A Method of Extracting Prolog Programming Techniques*

Wamberto Weber Vasconcelos†
Dept. Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN — Scotland, Great Britain
wamb@aisb.ed.ac.uk

Technical Paper 27

Abstract

We present a method of extracting the programming techniques employed in Prolog programs. Techniques are *dynamic* entities consisting of the syntax of the program and how it is used. The method records how subgoals are employed and uses this, together with their syntax and other auxiliary information, to partition the program into single-argument procedures possibly sharing variables. A technique is formally characterised as a sequence of such single-argument procedures.

1 Introduction

The compactness of Prolog programs allows the easy detection of commonly occurring patterns in the code. These patterns are loosely named *techniques* [Bow92, Brn91, BBD+91]. Prolog programming techniques are not directly expressed through specific syntactic primitives (*e.g.* "while" and "do-until" loops), but by the sophisticated use of the comparatively simple syntax of Prolog.

In this report we describe an automated approach to extract the programming techniques of a given working Prolog program. A Prolog programming technique is viewed here as a *dynamic* entity: the syntax of the program is complemented with the manner it has been used. To illustrate this issue, let there be the following formulation for the *prefix/2* predicate [SS86, O'K90],

```
prefix([],_).
prefix([X|Xs],[X|Ys]):-
     prefix(Xs,Ys).
```

which holds if the first argument, a list, is a prefix of the second argument, another list. This predicate allows for different usages and if it is used with its first argument free and its second argument instantiated to a list, its first argument will be instantiated to all the prefixes (one at a time) of this list. In such a circumstance, a technique to build a list has been used in the first argument position, and a list decomposition technique is employed in the second argument position. If, on the other hand, the predicate is used with its two arguments instantiated to lists and it succeeds, then we have list decomposition techniques employed in both argument positions.

The same syntax gives rise to different techniques, depending on how the predicate has been used. The "usage" of a predicate will be depicted here as the instantiation status of each of its variables and how they change during conventional Prolog interpretation. In the example above,

*A shorter version of this report has been accepted at the XI Brazilian Symposium on Artificial Intelligence, reproduced in [Vas94b]

†On leave from State University of Ceará, Ceará, Brazil; sponsored by Brazilian National Research Council (CNPq), under grant no. 201340/91-7.

the technique of building a list uses all its variables, with the exception of `X`, free, whereas in the decomposition technique all variables, with the possible exception of `X`, are instantiated.

These techniques can also involve different argument positions: a program to sum the elements of a list employs a technique to decompose a list and another argument position to perform the actual sum. Our assumption is that each argument of a Prolog program should serve some identifiable purpose, otherwise it should not have been inserted. The overall behaviour of a predicate is achieved by having its arguments interact in this way.

We restrict our attention to programming techniques within a single procedure[1]. Techniques spread across more than one predicate (*e.g.*, initialisation calls and mutually recursive predicates) are outside the scope of this work.

In the next subsection we explain the importance of the work presented here and how it relates to other areas of research. In the following subsection some notation and conventions are laid out. The last subsection of this first section describes the syntactic constraints on those programs analysed.

The second Section brings an overview of the extraction process, showing how the techniques of the *prefix/2* program are extracted, illustrating each stage of the proposed method. Section 3 explains the first stage of the extraction process, the *mode-annotation* of procedures, and compares different ways of achieving it. Section 4 explains how a mode-annotated procedure can be partitioned into *argument slices*, single-argument mode-annotated procedures comprising the building blocks of the programming techniques; issues on the correctness of the results and termination of this stage are also addressed. Section 5 describes the last stage of the extraction process, where clause annotations (depicting the sharing of variables across different argument slices) are inserted into the argument slices. Section 6 explains a manner to formally represent the extracted techniques. The last section summarises the work presented here and discusses its limitations.

1.1 The Importance of Formalising Programming Techniques

In [BBD+91], Brna *et al* informally describe programming techniques as the common code patterns used by programmers in a systematic way, being independent of any particular algorithm or problem domain. Programming techniques are, however, specific to a particular programming language, Prolog, in our case. The concept of Prolog programming techniques has been developed and applied in a variety of contexts:

- *Editing* — techniques-based Prolog editors have been proposed in [Rob91] and [Bow94]. Techniques-based editors offer their users techniques with which programs can be written: the task of writing a program can be transferred to that of finding a technique (or a set of techniques) which solves a given problem.
- *Program Tracing* — by keeping track of the way in which techniques were applied in the construction of a program using a techniques-based editor, it is possible to enhance explanations of the behaviour of programs by using this additional information. A tracing system using this approach is described in [Gab92].
- *Program Transformation* — programs can be more efficiently combined if information concerning which techniques are responsible for the flow of control of each program is available. In [VVRV93], it is shown how programs obtained via a techniques-based editor can be combined together yielding more efficient combined versions.

[1] We shall employ Deville's [Dev90] definition of a *logic procedure* or simply *procedure* as the sequence of program clauses with the same predicate p^n (predicate symbol p with arity n) in the head of each of these clauses. The built-in predicates available in Prolog are also called procedures and a logic program is a set of logic procedures.

- *Automatic Program Analysis* — techniques have been employed to identify bugs in the Prolog programs of students using a Prolog Intelligent teaching system [Loo88] and to criticise design decisions [Ben94] in students' intermediate sized code.

In [BRV+93] these items are explained in more detail. All these applications, however, assume the techniques are somehow encoded and made available, but no further details as to how this is done are provided. The preparation of these techniques may require much labour and ingenuity, for no formalisation or methodology has been proposed as an aid. After studying patterns frequently found in programs and acclaimed techniques informally described in Prolog textbooks and papers, a number of techniques are chosen and manually encoded so that the application can use them.

In this report we propose a means of automatically extracting the programming techniques from working Prolog programs. The extracted techniques are formally represented as single-argument procedures, possibly sharing variables across their clauses. The notation employed is based on the formalism proposed in [Vas94a], adapted to suit our needs[2]; in [BV93] an informal presentation of a similar notation to describe techniques can also be found. The extracted techniques can be stored in a library and supplied to each of the applications above.

1.2 Syntax of Programs and Adopted Notation

Only pure Prolog programs, without cuts, disjunctions or if-then-else's, complying with the Edinburgh Prolog syntax, are our concern here. Moreover, no `assert`, `retract`, `abolish` or similar database-altering predicates can be used. The built-in predicates currently handled by our method are:

- the operators `=`, `=:=`, `=\=`, `==`, `\==`, `>`, `>=`, `<`, `=<` and `=..` (denoted by "$\diamond$");
- the arithmetic operator `is`;
- the tests `atom`, `atomic`, `float`, `integer`, `number`, `var` and `ground` (denoted by "¶");
- the input predicates `read` and `get` (denoted by "$\Im$"), and
- the output predicates `write` and `display`.

We assume, without loss of generality, that the programs are in a *normal form*, with all unifications explicitly made via calls to `=` or `=..`. Each clause must be of the form

$$p(x_1, \ldots, x_n) \texttt{ :- } S_1 \ldots S_n$$

where $x_1, \ldots, x_n$ must be variables. Each subgoal S_i must comply with the following restrictions:

1. If it employs "$\diamond$" then it must be of the form $x \diamond a$ (the constant on the right-hand side of the operator), or $x \diamond f(y_1, \ldots, y_m)$ (the term on the right-hand side of $\diamond$), or $x \diamond y$;
2. If it is of the form $p(x_1, \ldots, x_n)$ then all its variables must be distinct, that is, $x_j \neq x_k, j \neq k$.

Moreover, constants and terms can only appear in subgoals employing "$\diamond$", `write`, `display` or `is` and terms should not contain multiple occurrences of variables. In Figure 1 below we show two versions of predicate *prefix/2*, the normal form (left-hand side) and the conventional form. This normal form allows for a homogeneous treatment of unifications: those unifications in the head goal and in body goals are handled in the same fashion, and their descriptions are

[2] The distinction between *skeletons* and *techniques* proposed by [KLS89] and adopted in [Vas94a] is deliberately blurred in this work.

```
prefix(A,B):-
     A = [].
prefix(A,B):-
     A = [X|Xs],
     B = [Y|Ys],
     X = Y,
     prefix(Xs,Ys).
```

```
prefix([],_).

prefix([X|Xs],[X|Ys]):-
     prefix(Xs,Ys).
```

Figure 1: Normal Form (left) and Conventional (right) Versions of Procedure *prefix/2*

similar. Another advantage is that programs in a normal form provide a detailed account of the computations taking place, splitting complex operations (such as decomposing a data structure *and* testing parts of it in the same subgoal) into a sequence of simpler subgoals. The descriptions of the techniques of such programs are clearer, consisting of a sequence of simple subgoals.

In this work variables are denoted by u, v, w, x, y and z, possibly super- and subscripted; constants are denoted by a, b and c, possibly super- and subscripted; function symbols are denoted by f, g and h, possibly super- and subscripted, the superscript standing for the arity of the function symbol — f_i^0 also stands for a constant; predicate symbols are denoted by p, q and r, possibly super- and subscripted, the superscript standing for the arity of the predicate symbol. These are meta-symbols through which Prolog constructs can be generically referred. The construction $x = y$ stands for a test in which the actual Prolog variable symbol abstractly represented by x is the same variable symbol as that represented by y. Specific Prolog constructions will be in `this kind of font`.

2 Overview of the Extraction Process

Given a working Prolog program, complying with the syntactic restrictions above, the method described here extracts those programming techniques used in it. The proposed method carries out the analysis and extraction of the techniques of a procedure *with respect to a query*, which defines how the predicate is to be used. In the first stage of our method, the procedure P of arity n, comprising clauses $C_1, \ldots, C_m$, is analysed with respect to a query Q and annotated with tokens describing the instantiation status of its variables before and after the execution of each subgoal. The outcome of this stage is the *mode-annotated* version of procedure P (wrt Q) denoted by $\widetilde{P}$, as shown in Figure 2.

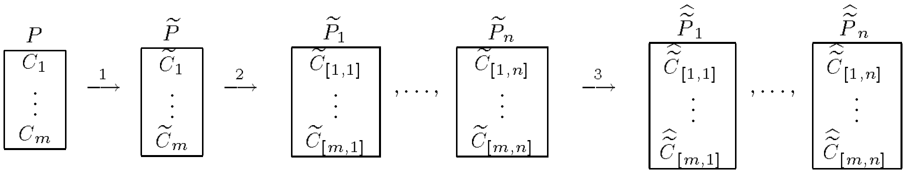


Figure 2: Stages of the Extraction Method and their Intermediate Results: 1 – Mode-Annotation; 2 – Argument Slicing; 3 – Clause Annotation

In the second stage $\widetilde{P}$ is partitioned into a sequence $\langle \widetilde{P}_1, \ldots, \widetilde{P}_n \rangle$ of single-argument procedures, its *argument slices*. Each argument position in the head goal of a clause has an argument slice consisting of those subgoals *relevant* to the argument position. The notion of *relevance* of a subgoal with respect to an argument slice is one of the contributions of this work.

The last stage of the method inserts *clause annotations*, place holders for variables referred across clauses of different argument slices, into the clauses of the mode-annotated argument slices, yielding the sequence $\langle \hat{\hat{P}}_1, \ldots, \hat{\hat{P}}_n \rangle$. Clause annotations state the required and offered resources (in the form of variables) of each clause of an argument slice. A technique is formally characterised as a sequence of argument slices sharing variables.

2.1 Working Example

The first stage of our method collects information about how each subgoal is used. This information is in the form of the instantiation status of the variables and how they change through the subgoal execution. The variables of each subgoal are assigned *tokens* describing their instantiation status: the token "`g`" is associated with ground variables, token "`f`" is associated with free variables and token "`i`" is associated with instantiated variables (variables instantiated to non-free terms).

The tokens associated with the variables of a subgoal before and after its execution describe how the subgoal has been used — the set of pairs x/T where x is a variable of a subgoal in the clause and T is the token associated with it comprise our *mode-annotations*. Each subgoal in the body of a clause has two mode-annotations corresponding to the instantiation status of the variables *before* and *after* its execution. The head goal has only one mode-annotation, corresponding to the instantiation status of the variables in the clause after it is matched. The mode-annotated version of *prefix/2*, given query

```
?- prefix(Pref,[foo1,foo2,foo3,foo4])
```

is shown in Figure 3, with only one copy of each clause. The mode-annotations of the first

```
                              prefix(A,B):-      {A/f,B/g}
                  {A/f,B/g}     A = [].          {A/g,B/g}
                              prefix(A,B):-      {A/f,B/g,X/f,Xs/f,Ys/f}
{A/f,B/g,X/f,Xs/f,Ys/f}         A = [X|Xs],      {A/i,B/g,X/f,Xs/f,Ys/f}
{A/i,B/g,X/f,Xs/f,Ys/f}         B = [X|Ys],      {A/i,B/g,X/f,Xs/f,Ys/g}
{A/i,B/g,X/g,Xs/f,Ys/g}         prefix(Xs,Ys).   {A/g,B/g,X/g,Xs/g,Ys/g}
```

Figure 3: Mode-annotated Version of Procedure *prefix/2*

clause show that after the head goal is matched, variable `A` is free (its associated token is `f`) and `B` is ground (its token is `g`); the execution of subgoal `A = []` instantiates `A` to the empty list, and causes the mode-annotation to be updated with `g` as the new token associated with `A`. Mode-annotated procedures provide us with a "picture" of how each clause was used during its execution. The mode-annotations supply the method with important information for the following stages. Different techniques can be used to mode-annotate procedures. In Section 3 they are explained in detail and their advantages and drawbacks are listed.

In the second stage of the extraction method, the procedure is partitioned into a set of single-argument procedures, its *argument slices*. Each argument position in the head goal of a clause has an argument slice consisting of those subgoals *relevant* to the clause. The argument slices of *prefix/2* are shown in Figure 4. The notion of *relevance* of a subgoal with respect to an argument slice is one of the contributions of this work, and is formally stated in Section 4, together with a detailed account of how the argument slicing is performed.

During the last stage of the method the set of mode-annotated argument slices receive *clause annotations*. The clause-annotated argument slices of *prefix/2* are shown in Figure 5 (only those clause-annotations relevant to the example are shown here; as will be explained later, other clause-annotations are also inserted, providing complementary information): the variable `Y` in the first argument slice is required and it is also offered in the second argument slice. Our approach to isolate programming techniques across each argument position requires that those variable symbols supposed to be the same in distinct argument slices be explicitly linked. The

```
prefix(A):-                                  prefix(B).
   A = [].
prefix(A):-                                  prefix(B):-
   A = [X|Xs],                                  B = [X|Ys],
   prefix(Xs).                                  prefix(Ys).
```

Figure 4: Argument Slices of Procedure *prefix/2*

```
                     p(A):-                  p(B).
                        A = [].
                     p(A):-                  p(B):-
⟨⟨required({X})⟩⟩       A = [X|Xs],             B = [X|Ys],   ⟨⟨offer({X})⟩⟩
                        p(Xs).                  p(Ys).
```

Figure 5: Clause-Annotated Argument Slices of Procedure *prefix/2*

idea of a technique as being the syntax of the procedure plus how it is used underlies the whole method, the mode-annotations being used to partition the procedure and to help insert the clause-annotations appropriately.

A technique is a sequence of related argument slices, that is, argument slices sharing variables via *required* and *offer* annotations. The argument slices of the *prefix/2* procedure displayed above define two techniques: one decomposing a list (second argument slice) and another building a list with those values `Y` obtained as another list is decomposed (combination of first and second argument slices).

3 Mode-Annotation of Procedures

The first stage in the extraction of programming techniques concerns the *mode-annotation* of the procedure with the instantiation status of the variables in each subgoal. The mode-annotation of a procedure collects information concerning how each variable was used during the execution of a procedure. This can be achieved by the concrete (actual) or the abstract interpretation of the procedure being analysed. Both alternatives have been implemented and each one has its advantages and disadvantages.

Nothing essentially original is being proposed for this task of mode-annotation. A straightforward enhancement of the "vanilla" meta-interpreter [SS86, SL88] gives rise to a concrete interpreter collecting the semantics of the procedure (see below). Classic abstract interpretation techniques were employed with some adaptation for our particular needs. The purpose of this section is to briefly describe each approach, explain some implementational issues and try to justify decisions taken. The advantages and deficiencies of each alternative are listed and compared and samples of their outputs are shown.

3.1 Mode-Annotated Procedures

The analysis of the programming techniques of an executable Prolog program is carried out with respect to a given query. Only that procedure defining the query (*i.e.*, those clauses whose head goal matches the query) will be analysed. A *mode-annotated* procedure consists of those clauses whose head goal matches the query annotated with *tokens* associated with the variables of each subgoal. These tokens convey information on the instantiation status of the variable, *i.e.*, if the variable is free, instantiated, ground, etc.

The mode-annotated clauses are obtained by inserting a simplified form of substitution after the head goal unification and before and after each subgoal in the body of the clause. A mode-annotated clause is of the form

$$H \text{ :- } \theta_0 \ \ \theta_1 \, S_1 \, \theta'_1, \ldots, \theta_n \, S_n \, \theta'_n$$

where H and S_i are subgoals and the θ_i and θ'_j are simplified substitutions in which the actual values associated with variables are replaced by tokens representing their instantiation status. The simplified substitutions contain all the variables of the clause, and their associated tokens change to reflect how the execution of each subgoal alters their instantiation status. The mode-annotation θ_0 contains the status of the variables immediately after the head goal is matched. It is obvious that $\theta_0 = \theta_1, \theta'_i = \theta_{i+1}$ but this replication is made necessary because during the argument slicing stage (Section 4) mode-annotated subgoals can be removed and the absence of θ's would render the mode-annotated clause inaccurate.

A mode-annotated procedure can be seen as a description of the execution of a procedure in terms of the changes in the instantiation status of the variables of its clauses. Instead of considering particular executions, with actual values assigned to variables, entire *classes* of executions can be depicted by means of mode-annotated procedures. Mode-annotated procedures do not have repeated clauses, that is, only one copy of each mode-annotated clause is considered. Those clauses in the original procedure which may be used more than once (in the example above, the recursive clauses can be used many times) have only one mode-annotated version of them in the mode-annotated procedure.

In this Section, we shall employ the Prolog procedure *collect/2* shown in Figure 6, in a normal form and conventional versions, to illustrate different aspects of the mode-annotation process. The *collect/2* predicate holds if its first argument is a list whose integer elements (if any) are

```
collect(A,B):-
   A = [],
   B = [].
collect(A,B):-
   A = [X|Xs],
   B = [X|Ys],
   integer(X),
   collect(Xs,Ys).
collect(A,B):-
   A = [X|Xs],
   collect(Xs,B).
```

```
collect([],[]).

collect([X|Xs],[X|Ys]):-
   integer(X),
   collect(Xs,Ys).

collect([X|Xs],Y):-
   collect(Xs,Y).
```

Figure 6: Normal Form (left) and Conventional Versions (right) of Procedure *collect/2*

to be found, in the same order, in the list comprising the second argument. It provides the mode-annotated version shown in Figure 7, with respect to query

```
?- collect([foo1,foo2,3,foo4,foo5,6],L).
```

For the sake of brevity, when displaying mode-annotated procedures we shall omit repeated mode-annotations, θ_0 being reused as θ_1 and θ'_i as θ_{i+1}.

3.2 The Set of Tokens

The following tokens are available for the mode-annotations:

- "`f`" — token associated with *free* variables;
- "`g`" — token associated with *ground* variables, *i.e.* variables bound to constants or (composed) terms with ground subterms only;

```
collect(A,B):-        {A/g,B/f}
   A = [],            {A/g,B/f}
   B = [].            {A/g,B/g}
collect(A,B):-        {A/g,B/f,X/f,Xs/f,Ys/f}
   A = [X|Xs],        {A/g,B/f,X/g,Xs/g,Ys/f}
   B = [X|Ys],        {A/g,B/i,X/g,Xs/g,Ys/f}
   integer(X),        {A/g,B/i,X/g,Xs/g,Ys/f}
   collect(Xs,Ys).    {A/g,B/g,X/g,Xs/g,Ys/g}
collect(A,B):-        {A/g,B/f,X/f,Xs/f}
   A = [X|Xs],        {A/g,B/f,X/g,Xs/g}
   collect(Xs,B).     {A/g,B/g,X/g,Xs/g}
```

Figure 7: Mode-Annotated Version (Normal Form) of Procedure *collect/2*

- "`i`" — token associated with *instantiated* variables, *i.e.* variables *not free.*

We need token "`i`" to represent the instantiation mode of variables bound to composed terms with at least one free variable, that is, "partially ground/partially free" structures (*e.g.* a list with a free variable as its tail). Neither "`f`" nor "`g`" would accurately describe this partially ground/partially free status.

One is able to appropriately describe the instantiation status of any variable in a execution with this set of tokens. Token "`g`" is redundant in the sense that it is just a refinement of token "`i`": any variable whose mode is "`g`" is also "`i`", but the converse is not always true. The distinction between these modes, however, is an important one: in order to extract programming techniques it is essential to detect the changes in the contents of the variables. Nothing accurate can be stated about the content of a variable which remains with token "`i`" associated with it throughout a subgoal execution.

These tokens are an attempt to move from particular values of variables to a more abstract setting in which *sets of values* can be considered instead. Entire *classes of executions* are represented when tokens are used to express sets of actual values of variables, and hence more generality is accorded to those techniques extracted.

3.3 How to Mode-Annotate a Procedure

A very simple approach to mode-annotate a clause is to run the program (say, by using a Prolog interpreter) with the initial query, collecting information about the variables of each subgoal before and after its execution, and storing this information for those clauses used in the execution.

Another way is by making use of *abstract interpretation*, in which computations are carried out abstractly. Instead of using the concrete domain of Prolog (*i.e.* the set of all constants and terms syntactically correct) and carrying out computations on these values, a very small *abstract domain* is employed and the computations are abstracted in terms of this abstract domain.

A mode-annotated procedure provides a concise account of how each variable was used during the execution of a clause. By "execution" we mean the conventional Prolog interpretation. We have exploited the concrete and abstract ways to interpret Prolog programs. Both approaches have advantages and drawbacks. In the following subsections the ideas behind each approach are explained in more detail with illustrative examples. Some high-level implementational issues are also discussed.

3.4 Mode-Annotation of Procedures using Concrete Interpretation

A simple way to implement the mode-annotation process is to run the program with the given query using a concrete (actual) Prolog interpreter and to obtain the mode-annotations as the execution proceeds. This can be achieved by supplying the vanilla meta-interpreter with the following enhancements:

- after a clause is retrieved and its head goal is matched against another subgoal, the resulting substitution is simplified by having the actual content of each variable replaced by one of the tokens; this simplified substitution is used as θ_0, the first mode-annotation of the clause, after its head goal, and θ_1 the mode-annotation *before* the first subgoal S_1;
- after a subgoal S_i is executed, the resulting substitution is simplified as above; the simplified substitution is used as θ_i', the mode-annotation *after* the subgoal, and θ_{i+1}, the mode-annotation *before* the following subgoal S_{i+1}.

Given our *collect/2* example, its concrete mode-annotation with respect to the query

```
?- collect([1,2,foo,3,bar],L)
```

eventually causes the retrieval of the second clause (it firstly retrieves the first clause, but it fails and backtracks to try another clause) and matches the query with its head goal, binding `A` to `[1,2,foo,3,bar]` and `B` to `L`; all other variables in the clause are free at this point. We thus have

```
collect(A,B):-        {A/g,B/f,X/f,Xs/f,Y/f,Ys/f}
   A = [X|Xs],
   B = [X|Ys],
   integer(X),
   collect(Xs,Ys).
```

Although it is not shown (for the sake of brevity) the initial substitution θ_0 is reused as θ_1. The instantiation status of each variable can be obtained via system predicates `var` which succeeds if its argument is a free variable and `ground` which succeeds if its argument is ground; if a variable is neither `var` nor `ground` then it is assigned the "`i`" token. Following the conventional execution order of a meta-interpreter, the first subgoal is reached: since the actual bindings are kept, it is possible to concretely execute it, matching the content of `A` with the list pattern and bind `X` to `1` and `Xs` to `[2,foo,3,bar]`. After the subgoal is executed, the inspection of the instantiation status of the variables in the clause yields the mode-annotation θ_1' after S_1:

```
collect(A,B):-        {A/g,B/f,X/f,Xs/f,Ys/f}
   A = [X|Xs],        {A/g,B/f,X/g,Xs/g,Ys/f}
   B = [X|Ys],
   integer(X),
   collect(Xs,Ys).
```

The remaining subgoals are processed similarly: they are concretely executed with the actual contents of its variables and then their variables have their instantiation status analysed and recorded in a mode-annotation.

The concrete execution of a program may use the same clause many times, obtaining the same mode-annotated version. However, these repeated clauses should be discarded since they do not offer any additional information: the mode-annotated version of a procedure does not have repeated clauses. The adjustments performed in the meta-interpreter must account for this. The mode-annotation of *collect/2* wrt the previous would obtain 1 mode-annotated version of the first clause (empty list), followed by 3 copies of the mode-annotated version of the second clause (which caters for the `integer` elements of the list) and 2 copies of the mode-annotated version of the third clause. The information concerning the number of times a clause is used is not relevant to our characterisation of a programming technique; the usage (in the form of the mode-annotations) of a clause, however, is essential.

Advantages and Drawbacks

The mode-annotation of a procedure using an enhanced meta-interpreter provides an accurate account of a particular execution of that procedure: the instantiation status of each variable is always known and either "`f`", "`i`" or "`g`" is assigned to it. The enhancement of the "vanilla" meta-interpreter is a straightforward process and the resulting code is very simple. This approach, however, has two pitfalls:

- *Non-termination*: because we are actually running a program while mode-annotations are collected, if the program does not terminate neither does the mode-annotation.
- *Incompleteness*: there might be clauses which are not used in the execution of the procedure and hence will not have their mode-annotated versions collected.

The non-termination might not be an important issue, for two reasons: firstly, and more importantly, the extraction analysis relies on the participation of an expert programmer whose initiative in choosing the program and the query is essential – it would be expected that the expert programmer had chosen the program *because* it computed (hence terminated) some interesting result. Secondly, we could simply rule out non-terminating programs (and their techniques), considering them outside the scope of this work.

In an ideal setting, a mode-annotated procedure should contain every clause annotated with the tokens of each variable. The mode-annotated clauses would provide all the information one needed to analyse its techniques, according to our approach. However, it might be the case that not every clause is used in the execution (hence not every clause appears in the mode-annotated procedure) and in order to extract the techniques, one needs all the clauses with their mode-annotations. We could, here again, rely on the user's choice of an appropriate query to solve this problem: if a clause is left out of the mode-annotation the user would be warned about it and another query would be asked.

To illustrate the incompleteness issue, we shall consider the *collect/2* procedure above: its mode-annotated version provided by the enhanced meta-interpreter of Prolog, given query

```
?- collect([foo1,foo2,foo3,foo4],L).
```

is:

```
collect(A,B):-       {A/g,B/f}
   A = [],           {A/g,B/f}
   B = [].           {A/g,B/g}
collect(A,B):-       {A/g,B/f,X/f,Xs/f}
   A = [X|Xs],       {A/g,B/f,X/g,Xs/g}
   collect(Xs,B).    {A/g,B/g,X/g,Xs/g}
```

The second clause of *collect/2*, in which the elements satisfying *integer/2* are used to build the list comprising the second argument, is left out. The missing clause turns out to be an essential feature of the technique employed in the second argument, seriously affecting any attempts to recognise it.

3.5 Mode-Annotation of Procedures using Abstract Interpretation

An alternative approach is to use *abstract interpretation* [CC92, KK87] to obtain the desired annotated procedure. Instead of using the actual domain of Prolog (constants and terms) and carrying out computations on these values, we employ an *abstract domain* upon which abstract computations take place. The main benefit of using abstract interpretation is to address the incompleteness problem mentioned above: within an abstract interpreter particular values are not regarded as important and more clauses can be analysed. However, one main drawback of this approach, to be exemplified later, is the inaccuracy of the information obtained.

Some assumptions are made concerning the program and the query initially supplied to the abstract interpreter: there must be no syntactic errors in either of them and there must be no run-time errors in the (concrete) execution of the program with respect to the query. These are not stringent constraints: we are assuming that the program was chosen by an expert programmer to be analysed because it computed some interesting result. Hence it is fair to assume that the program had no errors (syntactic or run-time) and eventually terminated.

Abstract interpretation can be seen as executing a program with an abstract description of a goal and obtaining an abstract description of a set of possible answers [Wae88]. The execution of a program is simulated in an abstract domain of approximations to those values found in concrete computations. The quality of the information obtained about a program by means of its abstract interpretation is a function of the abstract descriptions (the *abstract domain*) used.

3.5.1 Abstract Domain

An abstract domain consists of a set of tokens each of which represents a possibly infinite set of actual values. We shall define an abstract domain in which each token describes an instantiation status of a variable. The same tokens "`f`", "`i`" and "`g`" used in the mode-annotations can be employed for this purpose. However, due to limitations inherent in abstract interpretation techniques, an extra token "`?`", standing for any of the other tokens, has to be included. Having the token "`?`" assigned to a variable means that the variable may be free, instantiated or ground, but nothing more specific can be said.

Each token of the abstract domain represent an infinite set of concrete values. Token "`g`", for instance, stands for an abstraction of Prolog constructs such as "`100 + 2`", "`[bar,foo,baz]`", "`baz`", "`foo(bar,baz,foo(bar))`", etc. Token "`i`" stands for an abstraction of Prolog constructs such as "`f(X,a)`", "`[foo|X]`", etc. (where `X` is a free variable), and is also an abstraction of the token "`g`". The sets described by these tokens form a partial ordering, according to the reflexive set inclusion relation "$\subseteq$", graphically depicted in Figure 8. The abstract interpretation

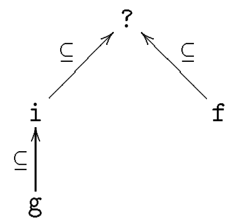


Figure 8: Set Inclusion Relationship between Tokens

functions simulating concrete Prolog computations are also naturally defined; the essential one, the binding of variables, is explained below.

3.5.2 Abstract Interpretation and Mode-Annotations

The mode of each variable is given by the token the abstract interpreter associates with its content: the concrete domain of Prolog is now abstracted to four values, upon which all the computations are simulated. In the abstract interpretation there is no need to analyse the content of a variable in order to obtain its mode and associate a token with it because the content of a variable is already such a token.

The syntactic constraints of the programs to be analysed, where only explicit unifications are permitted, enable us to view computations as the effect of these explicit unifications over abstract substitutions. The unification of variables is abstracted in terms of changes made to their associated tokens. A token T associated with x can only be changed into another token T' following the arrows in the graph of Figure 9. That is, a variable whose associated token is `f`

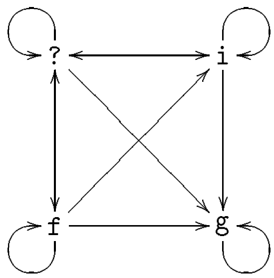


Figure 9: Possible Changes to Tokens

(a free variable) can be eventually associated with tokens `g`, `i` or `?`. Variables associated with token `i` (instantiated variables) can be associated, as a result of some computation, with `g` or `?`. Variables associated with `g` never have their tokens changed. Variables associated with `?` can become associated with `f`, `i`, or `g`. The unification between two variables x and y can be described by means of Table 1. The first column shows the possible values of x and the first line

$x \backslash y$	f	g	i	?
f	f	g	i	?
g	g	g	g	g
i	i	g	i	i
?	?	g	i	?

Table 1: Abstract Unification of Variables x and y

the possible values of y; each entry in the table depicts the token associated with *both* x and y after their abstract unification. For instance, entry (3,3) states that the unification of x/`i` with y/`i` does not alter their tokens.

During the abstract interpretation the sharing of variables within subterms is not recorded. If, for example, we are given a substitution {`X/f`, `Y/f`} and a subgoal `X = f(Y)` to interpret abstractly, then its outcome is {`X/i`, `Y/f`}, without any links whatsoever between `X` and `Y`, as there would be in the concrete interpretation of Prolog. If `Y` later becomes ground the status of `X` does not change; if `X` becomes ground the status of `Y` does not change either. This limitation is responsible for the inaccuracy of the mode-annotations given by the abstract interpreter.

3.5.3 Abstract Interpretation with Extension Tables

In order to mode-annotate a procedure via abstract interpretation we need to define the rules for the abstract computation of each subgoal and the mode-annotation process. As in conventional Prolog meta-interpreters, the subgoals of a clause will be grouped as:

- *System built-in procedures* — those predicates available in most Prolog interpreters and such that no definition (in the form of a sequence of clauses) is needed. The built-in procedures handled by our abstract interpreter are those of Subsection 1.2.
- *User-defined procedures* – those predicates representing calls to user-defined procedures, *i.e.*, procedures for which a sequence of clauses is found in the program. Recursive subgoals fall into this group.

System predicates are abstractly computed by updating the associated tokens of those variables involved: the meaning of a system predicate is known without any reference to a sequence of clauses defining it. For instance, the abstract computation of subgoal `X = [Y|Z]`, given an abstract substitution $\{X/g, Y/f, Z/f\}$, yields substitution $\{X/g, Y/g, Z/g\}$. The abstract interpreter defines the meaning of each Prolog system predicate in terms of changes in the tokens of the variables involved.

In the abstract computation of user-defined procedures either an analysis of the success patterns of previous clauses has to be carried out (in the case of recursive calls) or a complete analysis of the success patterns of auxiliary procedures must be performed. A success pattern is a subgoal together with a pair of abstract substitutions representing the status of the variables before and after the subgoal abstract computation — success patterns are stored in an *extension table*. As the abstract interpretation of the program proceeds the success patterns of all user-defined predicates employed are stored in an extension table.

Extension tables are an attempt to simulate the conventional Prolog interpretation of user-defined predicates. By keeping a record of call and exit patterns of each clause in a procedure, it is possible to infer, from these records, an exit pattern which subsumes all others, and *safely* abstracts the computations likely to be carried out in the actual execution of that procedure. Each entry in the extension table represents the result of the abstract computation of a user-defined subgoal on a substitution yielding another substitution. Our implementation of the abstract interpretation of recursive clauses exploits dynamic programming in order to find the mode-annotated clauses and the extension table: whenever the extension table is updated with a new entry, the abstract interpretation of the recursive clauses starts again from the very beginning, using the newly updated extension table.

3.5.4 Advantages and Drawbacks

There are two main advantages when using abstract interpretation in the mode-annotation of procedures:

- *Termination* — the abstract interpretation of programs eventually terminates. This is true even for non-terminating programs. The program has its execution *simulated* by having each clause separately interpreted, but no potentially non-terminating flow of control is actually established.
- *Completeness* — the supplied query is abstracted in terms of its modes and the analysis is carried out with respect to the modes of an abstract query. If the same query is supplied to both concrete and abstract interpreters, there is a guarantee that the clauses obtained in the latter are a superset of those obtained in the former.

These features overcome the disadvantages of the concrete interpretation pointed out previously — the system needs not rely on the user's appropriate choice of a query to produce good quality mode-annotated procedures, since the process always stops and supplies a mode-annotated version of every reachable clause in conventional Prolog execution.

During the abstract interpretation the query supplied by the user is abstracted to its modes, and all the computations are also abstracted in terms of modes. Instead of considering particular values as in the concrete interpretation only modes are employed, and hence more clauses can be reached and abstractly interpreted with the same initial query.

A major deficiency of abstract interpretation in comparison with the concrete interpretation is the lower quality of its mode-annotations themselves. During the subsumption of success patterns there might appear "`i`" or "`?`" tokens associated with variables and this does not tell much about the mode of the variables themselves. There is also the problem of instantiated variables whose free subterms are eventually ground — in the abstract interpretation those variables will remain associated with "i" whereas in the concrete interpreter they will be appropriately updated and have a "g" token associated with them.

To illustrate the points above, we shall get the abstractly interpreted mode-annotated version of procedure *collect/2* with respect to the same query "`collect([foo1,foo2,foo3,foo4],L)`" used previously (highlighting the incompleteness issue of the concrete interpretation). This query is abstracted as `collect(X/g,Y/f)` and is used to obtain the following mode-annotated procedure:

```
collect(A,B):-        {A/g,B/f}
   A = [],            {A/g,B/f}
   B = [].            {A/g,B/g}
collect(A,B):-        {A/g,B/f,X/f,Xs/f,Ys/f}
   A = [X|Xs],        {A/g,B/f,X/g,Xs/g,Ys/f}
   B = [X|Ys],        {A/g,B/i,X/g,Xs/g,Ys/f}
   integer(X),        {A/g,B/i,X/g,Xs/g,Ys/f}
   collect(Xs,Ys).    {A/g,B/i,X/g,Xs/g,Ys/i}
collect(A,B):-        {A/g,B/f,X/f,Xs/f}
   A = [X|Xs],        {A/g,B/f,X/g,Xs/g}
   collect(Xs,B).     {A/g,B/i,X/g,Xs/g}
```

It is worth noticing the mode-annotations on the head goals of the second and third clauses: they are of "less" quality than those obtained in the concrete interpretation (Section 3.4). This happens because in the second clause when subgoal `B = [X|Ys]` is abstractly computed there is simply an assignment of tokens to `B`, `X` and `Ys`, but no relationship is kept between these variables. If tokens `g` are associated with both `X` and `Ys`, then `B` should also have token `g` associated with it, but this does not happen because no variable dependency is recorded.

3.6 Final Remarks on the Mode-Annotation of Procedures

The mode-annotation of procedures can be achieved by concrete or abstract interpretation. Both approaches have been implemented and the following stages of our proposed method can use either of them.

Although the mode-annotations obtained in the abstract interpretation are less accurate than those obtained in the concrete interpretation, all the clauses of the procedure are taken into account. In the concrete interpreter, depending on the query, important clauses may be left out. The inaccuracy of mode-annotations of the abstract interpreter can be coped with in latter stages of the analysis, degrading gradually as it gets more inaccurate. The potential absence of clauses in the output of the concrete interpretation seriously impairs later stages of our analysis, but this can be automatically detected and a human intervention requested (to supply another query).

The mode-annotation θ' of auxiliary user-defined predicates may depend on the actual contents of its variables upon their call. The concrete interpreter yields only the specific mode-annotation provided by the actual contents of the variables in the auxiliary predicate If the auxiliary predicate is such that different values of variables with the same instantiation status provide different mode-annotations after its concrete interpretation then the resulting mode-annotated procedure is incomplete, because alternatives were not considered. This situation is hard to detect in the general case, so that the user could be asked for another query. The abstract interpreter, on the other hand, interprets auxiliary predicates abstractly, yielding mode-annotations covering any actual values, because only the modes are used instead.

A mode-annotated procedure can be seen as a static description of a class of executions. Particular (concrete) values of variables are abstracted to a token, representing a *set of values*, thus abstracting from a particular execution and giving generality to the mode-annotated procedure.

If the actual content of each variable were used in the annotations instead of the tokens, then those clauses used repeatedly in the concrete interpretation would have many copies with different annotations. Depending on the queries posed, this approach would yield a rather verbose account of the execution, in which similar computations would only differ in the actual contents of their parameters. By representing the actual values of variables as tokens, the the distinct versions of a clause (differing only in the actual contents of its variables) would be merged together into a single abstract construct which would stand for all of them.

4 Argument-Slicing of Mode-Annotated Procedures

In this stage the mode-annotated procedure obtained previously is partitioned into a sequence of distinct *argument slices*, *i.e.* single-argument mode-annotated procedures comprising the "building blocks" of more complex programming techniques. Each argument position in the head goal of a mode-annotated clause has an argument slice consisting of those subgoals *relevant* to the argument position. This notion of relevance is formally stated in this section.

In this stage we carry out an analysis of each subgoal in order to decide whether or not it belongs to a particular argument slice. The same subgoal may belong to more than one argument slice and a subgoal may not be relevant to any argument slices. The analysis performed here takes into account existing relationships between the variables of the subgoal and the changes in their modes. The mode-annotations play an essential role in the definition of the conditions a subgoal must fulfill to be included in an argument slice. Different mode-annotations in a procedure may yield different argument slices and hence different techniques. This means that different executions of the same procedure may yield different techniques.

The adopted criterion of the partitioning proposed here is Prolog itself: if a subgoal neither *interferes with* nor *contributes to* the computations of the argument slice, then it is not relevant and should not be included in the argument slice. More specifically, a subgoal is relevant to a clause in an argument slice if:

- it supplies a value or values used by the recursive call(s) or by the argument in the head of the clause;
- it employs variables of recursive call(s) or of the head goal, interfering with the flow of control.

The mode-annotations are used in the definitions below to distinguish between the different cases. A formalisation of this notion of relevance is the main contribution of this section. In this section we also discuss the accuracy and termination of this stage in our method.

We shall illustrate this stage with the *sum/2* procedure [SS86] whose mode-annotated (normal form) version w.r.t. to query `sum(X/g,Y/f)` is shown in Figure 10, together with its two mode-annotated argument slices. The *sum/2* procedure holds if the second argument is the sum of the elements of the list comprising the first argument. One should notice that in the argument slices of Figure 10 the subgoal "`B is E + C`" is considered as part of slice `B` and not as part of slice `A`, in spite of variable `C` being the head of list `A`. This decision was made by analysing how variables have their modes changed before and after each subgoal — variable `C` in subgoal "`B is E + C`" is "donating" its value to the computation of `B`.

The method proposed here detects cases in which it is possible to infer, with certainty, that a subgoal is relevant to an argument slice. If a subgoal is not one of these cases then it will be considered irrelevant to the argument slice being prepared. These cases are attempts at formalising the components of a programming technique. The quality of the outcome of this analysis is highly dependent on the quality of the mode-annotations: the better these are, the better the outcome is. The more tokens "`i`" or "`?`" in the mode-annotations, the less accurate is the argument slicing.

```
sum(A,B):-          {A/g,B/f}
      A = [],       {A/g,B/f}
      B = 0,        {A/g,B/g}
sum(A,B):-          {A/g,B/f,C/f,D/f,E/f}
      A = [C|D],    {A/g,B/f,C/g,D/g,E/f}
      sum(D,E),     {A/g,B/f,C/g,D/g,E/g}
      B is E + C,   {A/g,B/g,C/g,D/g,E/g}
```

```
sum(A):-
      A = [].
sum(A):-
      A = [C|D],
      sum(D).
```

```
sum(B):-
      B = 0.
sum(B):-
      sum(E),
      B is E + C.
```

Figure 10: Mode-Annotated (Normal Form) Version of *sum/2*, and its Argument Slices

In the definitions of this section $\widetilde{C}$ stands for a mode-annotated clause of the form

$$\begin{array}{l} p(x_1, \ldots, x_n) \text{ :- } \theta_0 \\ \quad \vec{S}_0 \\ \quad \theta_0^p\, p(x_{[0,1]}, \ldots, x_{[0,n]})\, {\theta'}_0^p \\ \quad \vec{S}_1 \\ \quad\ \vdots \\ \quad \vec{S}_r \\ \quad \theta_r^p\, p(x_{[r,1]}, \ldots, x_{[r,n]})\, {\theta'}_r^p \\ \quad \vec{S}_{r+1}. \end{array}$$

where $\vec{S}_i, 0 \leq i \leq r+1$, are possibly empty vectors of non-recursive mode-annotated subgoals, each of the form

$$\theta\, q(\ldots\ y_1\ \ldots\ y_m\ \ldots)\, \theta'$$

and denoted by $\widetilde{S}$. Sometimes mode-annotated subgoals will be shown enclosed in boxes to make it easier to read.

4.1 Relationships between Variables

In order to decide whether or not a subgoal is relevant we must find out about which computations take place in each subgoal and how important each subgoal is for its variables. This is done by studying the mode-annotations of each subgoal, and deciding which variables had their contents changed and which variables were simply used in the subgoal execution, without having their contents altered.

The actual content of a variable is abstracted as a token and this has to be considered during the analysis of this stage. If a variable has associated tokens "`f`" (or "`g`") before and after a subgoal it is correct to assume that its content did not change. If a variable has tokens "`f`" and "`g`" (or "`f`" and "`i`") associated with it before and after (respectively) the subgoal execution, it is correct to assume that its actual content did change. However, for tokens "`i`" and "`?`", representing supersets of values of other tokens, it is not possible to say with accuracy when changes take place. If a variable has token "`i`" associated to it before and after a subgoal it might have been the case that the variable remained with its content unaltered or its content changed to a "more instantiated" value. If, for instance, variable `X` is bound to `f(A,B)`, where `A` and `B` are free variables, it has token "`i`" associated with it; if `A` or `B` gets instantiated, then the token associated to `X` is still "`i`", in spite of its content having changed. Similar situations may arise for token "`?`".

We define three relationships, *fixed*, *change* and *unknown*, of a variable x with respect to the mode-annotations θ and θ' of a subgoal, which hold if it is safe to assume that the content of x, abstracted by its tokens in θ and θ', has remained fixed, has changed or is unknown, respectively:

Definition 4.1 $\mathit{fixed}(x, \theta, \theta')$ holds if $x/T \in \theta$ and $x/T' \in \theta'$ and $T = T', T \in \{\mathtt{g}, \mathtt{f}\}$.

Definition 4.2 $\mathit{change}(x, \theta, \theta')$ holds if $x/T \in \theta$ and $x/T' \in \theta'$ and $T = \mathtt{f}, T' \in \{\mathtt{g}, \mathtt{i}\}$.

Definition 4.3 $\mathit{unknown}(x, \theta, \theta')$ holds if $\neg \mathit{fixed}(x, \theta, \theta')$ and $\neg \mathit{change}(x, \theta, \theta')$.

Variables whose associated tokens satisfy the *fixed* relation will be called *fixed variables*. Variables whose associated tokens satisfy the *change* relation will be called *changing variables*.

The mode-annotated subgoals provide us with the description of the relations between their variables: those fixed variables supply their values to compute the changes in the contents of the changing variables. A mode-annotated subgoal of the form

$$\{\ldots \mathtt{X/g}, \mathtt{Y/f}, \mathtt{Z/f} \ldots\}\ \mathtt{X = f(Y,Z)}\ \{\ldots \mathtt{X/g}, \mathtt{Y/g}, \mathtt{Z/g} \ldots\}$$

for example, defines relationships between `X` and `Y` and between `X` and `Z`. `X` *provides* the values for both `Y` and `Z`. Variables whose tokens do not satisfy the relations *fixed* or *change* pose a serious problem, because nothing accurate can be stated about their relationships. A mode-annotated subgoal of the form

$$\{\ldots \mathtt{X/i}, \mathtt{Y/?} \ldots\}\ \mathtt{p(X,Y)}\ \{\ldots \mathtt{X/i}, \mathtt{Y/i} \ldots\}$$

for instance, can be seen as defining a relation between `X` and `Y`, with `X` providing its value to compute the possible changes in the content of `Y` and vice-versa. In order to capture these relationships, we define the possibly empty set of pairs of $\lhd$-related variables of a subgoal: a variable x is $\lhd$-related to y, $x \lhd y$, if its content may have been changed employing the content of y. There might be more than one of such pairs for each subgoal and hence a set has been employed to record them:

Definition 4.4 The set $\rho_{\widetilde{S}}$ of $\lhd$-related pairs (or simply $\lhd$-pairs) of $\widetilde{S} = \theta\, S\, \theta'$ is comprised of elements of the form $y_i \lhd y_j$ such that one of the cases below holds

1. $\mathit{change}(y_i, \theta, \theta')$ and $\mathit{fixed}(y_j, \theta, \theta')$,
2. $\mathit{unknown}(y_i, \theta, \theta')$ and $\mathit{fixed}(y_j, \theta, \theta')$,
3. $\mathit{change}(y_i, \theta, \theta')$ and $\mathit{unknown}(y_j, \theta, \theta')$,
4. $\mathit{unknown}(y_i, \theta, \theta')$ and $\mathit{unknown}(y_j, \theta, \theta')$.

If $\widetilde{S}$ is of the form $\theta\ x \mathtt{=} y\ \theta'$ where $x/\mathtt{f} \in \theta, y/\mathtt{f} \in \theta$ and $x/\mathtt{f} \in \theta', y/\mathtt{f} \in \theta'$, a special set $\rho_{\widetilde{S}}$ of $\lhd$-pairs is defined as $\rho_{\widetilde{S}} = \{x \lhd y, y \lhd x\}$.

We are proposing a manner to represent the relationships between variables using their modes and how they change or remain constant during the clause execution. Subgoals of the form $x \mathtt{=} y$, where the modes of the variables remain unchanged as "`f`", deserve special attention for, in spite of the contents of x and y not having changed, the variables were definitely related to each other by means of the subgoal, this relation being useful in the relevance analysis explained below.

Example: Given a mode-annotated subgoal $\widetilde{S}$ of the form

$$\{\ldots \mathtt{A/g}, \mathtt{C/f}, \mathtt{D/f} \ldots\}\ \mathtt{A = [C|D]}\ \{\ldots \mathtt{A/g}, \mathtt{C/g}, \mathtt{D/g} \ldots\}$$

then $\rho_{\widetilde{S}} = \{\mathtt{C} \lhd \mathtt{A}, \mathtt{D} \lhd \mathtt{A}\}$.

A variable may be indirectly related, via an intermediate subgoal, to another variable. To deal with such a situation, we extend the definition of $\lhd$-pairs sets to cover whole clauses. The set of $\lhd$-pairs of a clause is built in a piecemeal fashion, each non-recursive subgoal at a time. $\vec{S}$, a vector of mode-annotated subgoals, has its set $\rho_{\vec{S}}$ of $\lhd$-pairs defined as the union of the sets of $\lhd$-pairs of its constituent subgoals:

Definition 4.5 The set $\rho_{\vec{S}}$ of $\lhd$-pairs of $\vec{S} = \widetilde{S}_1, \ldots, \widetilde{S}_n$ is $\rho_{\vec{S}} = \bigcup\limits_{i=1}^{n} \rho_{\widetilde{S}_i}$.

The set of $\lhd$-pairs of a mode-annotated clause is the union of the sets of $\lhd$-pairs of each vector of non-recursive mode-annotated subgoals — recursive subgoals are not considered in this analysis:

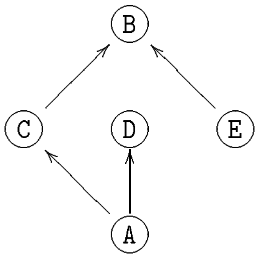
Definition 4.6 The set $\rho_{\widetilde{C}}$ of $\lhd$-pairs of $\widetilde{C}$ is $\rho_{\widetilde{C}} = \bigcup\limits_{i=0}^{r+1} \rho_{\vec{S}_i}$.

Example: The argument slicing of the mode-annotated *sum/2* procedure shown previously is performed by first building the set $\rho_{\widetilde{C}_i}$ of each clause. The first clause $\widetilde{C}_1$ has an empty set of $\lhd$-pairs. The second clause has the following set of pairs:

$$\rho_{\widetilde{C}_2} = \{\mathtt{C} \lhd \mathtt{A}, \mathtt{D} \lhd \mathtt{A}, \mathtt{B} \lhd \mathtt{E}, \mathtt{B} \lhd \mathtt{C}\}$$

the first and second pairs are from subgoal `A = [C|D]`; the third and fourth pairs are from subgoal `B is E + C`.

The set of $\lhd$-pairs provides an account of the dependency between the variables of a clause and can be seen as a *dependency graph*: any indirect relationship between two variables can be found by analysing the paths defined by the $\lhd$-pairs. The set of $\lhd$-pairs of the previous example can be represented as the following graph:



The analysis carried out in the slicing of a mode-annotated procedure may involve indirect $\lhd$-relationships. Indirect $\lhd$-relationships can be represented by a sequence of pairs $x_1 \lhd x_2, x_2 \lhd x_3, \ldots, x_{n-1} \lhd x_n$, stating that x_1 is indirectly $\lhd$-related to x_n. The definition below employs the set of $\lhd$-pairs as the description of a directed graph, and an element of the form $x \lhd y$ is considered as a directed edge linking variable y to x:

Definition 4.7 Given a set ρ of $\lhd$-related pairs and two variables x and y, the relation $x \lhd^*_\rho y$ holds if

1. $x \lhd y \in \rho$, or
2. $z \lhd y \in \rho$ and $x \lhd^*_\rho z$.

The relation $x \lhd^*_\rho y$ conveys the idea that the content of x may have been changed employing, possibly indirectly, the content of y. The problem of deciding whether $x \lhd^*_\rho y$ holds is similar to that of deciding if two nodes in a graph are connected — standard search algorithms, such as breadth-first or depth-first, can be employed here.

Example: Given the set $\rho_{\widetilde{C}_2}$ of the previous example, it is the case that $\mathtt{B} \lhd^*_{\rho_{\widetilde{C}_2}} \mathtt{A}$, that is, variable `B` is indirectly related to variable `A` (via intermediate variable `C`).

The definition of $\lhd^*_\rho$ above can be extended to cope with sets of variables: $V \lhd^*_\rho W$ holds if there is at least one variable $x \in V$ and at least one $y \in W$ such that the relation $\lhd^*_\rho$ holds:

Definition 4.8 Given a set ρ of $\lhd$-related pairs, a non-empty set of variables $V = \{x_1, \ldots, x_n\}$ is $\lhd^*_\rho$-related to the non-empty set of variables $W = \{y_1, \ldots, y_m\}$, $V \lhd^*_\rho W$, if there are i, j such that $x_i \lhd^*_\rho y_j$.

Example: A mode-annotated clause $\widetilde{C}$ (obtained through abstract interpretation), is shown below with the gradual preparation of its set $\rho_{\widetilde{C}}$ — the entries appear next to the subgoal introducing them:

```
collect(A,B):-          {A/g,B/f,X/f,Xs/f,Ys/f}
     A = [X|Xs],        {A/g,B/f,X/g,Xs/g,Ys/f}      ρ = {X ◁ A, Xs ◁ A,
     B = [X|Ys],        {A/g,B/i,X/g,Xs/g,Ys/f}           B ◁ X, B ◁ Ys}
     integer(X),        {A/g,B/i,X/g,Xs/g,Ys/f}
     collect(Xs,Ys).    {A/g,B/i,X/g,Xs/g,Ys/f}
```

$$\rho_{\widetilde{C}} = \{\mathtt{X} \lhd \mathtt{A}, \mathtt{Xs} \lhd \mathtt{A}, \mathtt{B} \lhd \mathtt{X}, \mathtt{B} \lhd \mathtt{Ys}\}$$

This set $\rho_{\widetilde{C}}$ is such that $\mathtt{B} \lhd^*_{\rho_{\widetilde{C}}} \mathtt{A}$.

4.2 Relevance of Subgoals

Analysing the relevance of a subgoal requires that we find out what computation(s) the subgoal is actually performing. The computations carried out by system predicates are easy to deduce because it is known what their meanings are and what behaviours are expected. Subgoals employing the operator `>`, for instance, are always tests, whatever their mode-annotations are. On the other hand `=` and `=..` can be used either as tests or to change the contents of variables: in such circumstances the mode-annotations may help in finding out what computations are taking place. The accuracy of the mode-annotations is crucial at this point.

The procedure to check the relevance of a subgoal to an argument slice relies on the following information:

1. *the syntax of predicate*: it may be enough to tell exactly what computations are being performed in the subgoal;
2. *the mode-annotations*: in some circumstances, they may provide extra information in finding out what computations are being performed;
3. *the set* $\rho_{\widetilde{C}}$: the relationships between the variables of different subgoals of the mode-annotated clause $\widetilde{C}$ support the recognition of *chains* of subgoals which are indirectly relevant to an argument slice;
4. *a possibly empty list of mode-annotated user-defined predicates that may fail*: it provides extra information to find out what user-defined predicates compute.

Recursive subgoals are sliced simply by restricting their arguments to the variable $x_{[j,i]}$ occupying position i in the j-th recursive call. The head goal is sliced by restricting its variables to x_i, occupying position i. The variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$ of the argument-sliced head and recursive subgoals play an important role in the analysis of relevance of a subgoal: if the subgoal does not affect directly or indirectly these variables, it should be considered irrelevant. A subgoal affects these variables if it

- *changes* the content of a variable (either by assigning a value to it or by instantiating parts of it) employed, directly or not, to change the contents of one of the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$;

- *tests* a variable whose value was obtained, directly or not, from one of the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$;

These two broad cases are formalised and exemplified below.

4.2.1 Dealing with Inaccurate Mode-Annotations

The successful argument-slicing of mode-annotated procedures relies on the ability to deduce, given the mode-annotations and the syntax of the subgoal, what computations are being performed. This ability underlies the whole process of the extraction of programming techniques. The mode-annotation stage is merely an initial step where more resources (*i.e.* how the predicate has been employed in terms of the instantiation status of its variables) are collected to help this decision.

Subgoals employing those "$\diamond$" system predicates different from `=` and `=..` (`=:=`, `>`, `>=`, etc.) are always tests, the mode-annotations not being necessary to state this. The same is true for the ¶ built-in tests (`atom`, `var`, etc.). In order to find out what computations those subgoals employing other system predicates are performing, the mode-annotations have to be used. The `=` system predicate, for instance, could either assign a value to a variable, that is,

$$\{\ldots \mathtt{X/f}, \mathtt{Y/g} \ldots\}\ \mathtt{X = Y}\ \{\ldots \mathtt{X/g}, \mathtt{Y/g} \ldots\}$$

or test whether two variables have the same contents, that is,

$$\{\ldots \mathtt{X/g}, \mathtt{Y/g} \ldots\}\ \mathtt{X = Y}\ \{\ldots \mathtt{X/g}, \mathtt{Y/g} \ldots\}$$

The mode-annotations provide us with the extra information we need to differentiate the two ways to employ the `=` system predicate. If the mode-annotations are accurate then they provide us with the precise information about the ongoing computations in that subgoal. However, they may be inaccurate, and the analysis carried out employing them inherits this imprecision.

Inaccurate mode-annotated subgoals contain variables satisfying the *unknown* relation (Def. 4.3). In these circumstances, it cannot be stated whether or not the content of this variable has remained fixed or been changed by means of the subgoal execution. These conditions arise when an abstract interpreter is employed during the previous mode-annotation stage.

There is no guaranteed way to decide precisely, given the available information (*viz.*, mode-annotations and syntax of subgoal), what actually happens to a variable satisfying the *unknown* relation. Some system predicates, whose effects on their variables are known, allow the ruling out of certain possibilities. The system predicate `is`, for instance, must be used with its right-hand side variables fully ground (and numeric) and it does not change their contents. Whenever these variables have inaccurate associated tokens (and bearing in mind that only programs without run-time errors are supplied to our analysis here) we can safely assume that these variables are all ground. User-defined predicates and system predicates `=` and `=..` however, do not pose such restrictions and hence we cannot discard the different possibilities for what happened to their variables.

The ideal solution would exploit separately both possibilities for each *unknown* variable within a subgoal: each variable would be considered both to have its content changed and to have remained fixed throughout the subgoal execution. This might turn out to be computationally expensive: a subgoal with n *unknown* variables yields 2^n different possibilities. This result gets even worse when whole clauses are taken into account, for the number of possibilities of a clause is the product of the number of possibilities of each subgoal. The total number of possibilities of the procedure would still have to compute the different possibilities and combinations of each distinct clause.

At the other extreme, another solution would be to simply discard all those variables with inaccurate associated tokens, and only use those variables with accurate tokens. This solution would provide unique but extremely poor results, leaving prospective subgoals out of the relevance analysis.

In between the two extreme solutions above, we could establish an *a priori* policy towards inaccurate mode-annotations. This policy states how a variable with inaccurate associated tokens will always be viewed: as a variable changing its content, as a variable whose content remained fixed, or as both. The first two options are completely arbitrary and bear no plausible reason for either choice. The third option should not be taken as being similar to the first computationally expensive solution presented above: here the different possibilities are considered simultaneously during the relevance analysis, and not pursued separately, as the initial solution above.

We adopt the last option here: a variable with inaccurate associated tokens will be considered *both* as if it had its content changed *and* as if it remained with its content unchanged. The decision procedure proposed here for the relevance of a subgoal has some overlapping between the cases of variables changing and remaining fixed, making the adopted policy less arbitrary. In some cases, it does not matter if a variable is either changing or fixed (one of which must be the case): if one of the relationships described below holds then the subgoal is relevant to the argument slice.

4.2.2 Dealing with User-Defined Predicates

When user-defined predicates are employed, it becomes impossible to state, in general and with certainty, what computations are being carried out. Accurate mode-annotations reveal whether the contents of variables have changed, and hence whether the user-defined predicate was used to obtain new values. However accurate the mode-annotations are, they are of little use when we are concerned with finding out if the subgoal may fail; that is, if the employed user-defined predicate, together with its call mode, is a *test*.

Accurate mode-annotations tell us when the content of a variable has remained fixed. It is, however, impossible to say if fixed variables should be considered as variables being tested, possibly causing the failure of the user-defined predicate. One should notice that we are not referring to failures caused by an aborted execution: our assumption is that this analysis is supplied with correct (no run-time errors) Prolog code. We are concerned with those user-defined predicates that may fail, potentially altering the flow of control of a procedure.

The problem of deciding if a user-defined predicate may fail is harder to solve than the termination problem. Here we not only are interested in finding out whether the execution of the predicate terminates (*i.e.*, reaches a point in its execution where no further progress takes place) but whether it terminates as a consequence of a failure. Techniques such as *unfolding* the definition(s) of the user-defined predicate [TS84] onto the top-most procedure would tackle some cases where system predicates are encapsulated within a user-defined predicate, but this approach is not general enough to cope with recursive calls within the user-defined predicate. Abstract interpretation could also be used to infer the termination of these auxiliary predicates. This alternative, supplied with type information, would cover a range of cases, but it would not be guaranteed to work in general.

Our approach adopts a *shallow* analysis in which the definition of user-defined predicates is not taken into account. The solution to the problem of finding out whether or not a user-defined predicate may fail is solved by having the user of our system supplying this information: the user must provide all those non-system predicates (and their call modes) which might fail. Only these predicates (with the specified modes) will be considered to be user-defined tests. This might prove to be too heavy a burden for the users, but they can always run these predicates to find out about their termination or failure. A hybrid approach would use both the unfolding and the abstract interpretation techniques mentioned above, and only if they do not perform satisfactorily would the user be asked.

When a user-defined predicate is found during the argument slicing stage it is tested, together with its mode-annotations, against the supplied set of user-defined predicates. If there is an entry whose call-mode is *subsumed* by the mode-annotation in the procedure, then the subgoal is considered to be a test. To illustrate this point, let us consider the following alternative definition *sum2/2* for the procedure *sum/2* shown previously:

```
sum2(A,B):-
     A = [],
     B = 0.
sum2(A,B):-
     A = [C|D],
     sum2(D,E),
     plus(B,C,E).

plus(A,B,C):-
     A is B + C.
```

whose mode-annotated version, w.r.t. query `sum2(X/g,Y/f)` is:

```
sum2(A,B):-        {A/g,B/f}
     A = [],       {A/g,B/f}
     B = 0.        {A/g,B/g}
sum2(A,B):-        {A/g,B/f,C/f,D/f,E/f}
     A = [C|D],    {A/g,B/f,C/g,D/g,E/f}
     sum2(D,E),    {A/g,B/f,C/g,D/g,E/g}
     plus(B,C,E).  {A/g,B/g,C/g,D/g,E/g}
```

The *plus/3* user-defined predicate never fails (it may abort, but this possibility is ruled out from those programs presented to us) if its first variable is free, but it may fail if all its arguments are ground. The definition of *plus/3* could be unfolded into *sum2/2* yielding the *sum/2* code shown previously, and the computation being carried out by *plus/3* becomes clear. This approach however, as pointed out above, does not work in the general case. The adopted solution would check if the mode-annotated subgoal

$$\{\ldots \mathtt{B/f}, \mathtt{C/g}, \mathtt{E/g} \ldots\}\, \mathtt{plus(B,C,E)}\, \{\ldots \mathtt{B/g}, \mathtt{C/g}, \mathtt{E/g} \ldots\}$$

subsumes one of the user-defined predicates supplied by the user as being a test. In fact, if the user had provided the list of tests of our system with the mode-annotated subgoal

$$\{\mathtt{X/g}, \mathtt{Y/g}, \mathtt{Z/g}\}\, \mathtt{plus(X,Y,Z)}\, \{\mathtt{X/g}, \mathtt{Y/g}, \mathtt{Z/g}\}$$

(stating that those calls to *plus/3* employing its three arguments ground may fail) then the use of *plus/3* in *sum2/2* would not be considered a test, because the call mode "`f`" of the first variable in the mode-annotated subgoal does not subsume the call mode `g` of the first variable of the subgoal in the supplied list.

We define here the set τ of user-defined tests:

Definition 4.9 The set of user-defined predicates which may fail (tests), τ, is a possibly empty set of mode-annotated subgoals.

To check whether a mode-annotated subgoal $\widetilde{S}$ may fail (whether S is a user-defined test) amounts to finding out if there is an element $\widetilde{U}$ of τ with the same predicate symbol and arity of $\widetilde{S}$ and whose mode-annotation θ_S is subsumed by the mode-annotation θ_U of $\widetilde{U}$. The subsumption of each token follows the set inclusion relation $\subseteq$ as explained in Subsection 3.5.1:

Definition 4.10 A token T subsumes another token T', $subsume_t(T,T')$, if one of the cases below holds

1. $T = T'$,
2. $T = \mathtt{i}, T' = \mathtt{g}$,

3. $T = ?$.

The subsumption between tokens faithfully emulates the relation between the sets of values represented by each token — if $subsume_t(T, T')$ then $T \supseteq T'$. The subsumption of tokens extends to cope with mode-annotations:

Definition 4.11 $\theta = \{x_1/T_1, \ldots, x_n/T_n\}$ subsumes $\theta' = \{y_1/T'_1, \ldots, y_m/T'_m\}$, $subsume_a(\theta, \theta')$, if for every y_j there is one x_i such that $x_i = y_j$ and $subsume_t(T_i, T'_j)$.

One should notice that the above definition allows for variables x_i in θ which are not found in θ'. We extend the subsumption relationship to cover subgoals:

Definition 4.12 $\widetilde{S} = \theta_S\, p(y_1, \ldots, y_n)\, \theta'_S$ subsumes $\widetilde{U} = \theta_U\, p(z_1, \ldots, z_n)\, \theta'_U$, $subsume_s(\widetilde{S}, \widetilde{U})$, if $subsume_a(\theta_S, \theta_U)$ and $subsume_a(\theta'_S, \theta'_U)$.

A mode-annotated subgoal $\widetilde{S}$ is a user-defined test if it subsumes an element of τ:

Definition 4.13 Given a set τ of mode-annotated subgoals which may fail (supplied by the user), a mode-annotated subgoal $\widetilde{S}$ is a user test, $\textit{user-test}(\widetilde{S}, \tau)$, if there is an element $\widetilde{U}$ of τ such that $subsume_s(\widetilde{S}, \widetilde{U})$.

4.2.3 Relevance of Subgoals Changing the Contents of Variables

Subgoals which change the contents of a variable are important to our notion of a programming technique because they define the computations through which values are obtained — these values may help defining the flow of control of the program or may be the final values computed by the programming technique. The definition below lists those cases when we can infer, by means of the syntax and mode-annotations of the subgoal $\widetilde{S}$, that a value is possibly being assigned to variable x:

Definition 4.14 A variable x has its content possibly changed in subgoal $\widetilde{S}$, $change(x, \widetilde{S})$, if, and only if, one of the cases below holds:

1. $\widetilde{S} = \boxed{\theta\ x\ \diamond\ \ldots\ \theta'}, \diamond \in \{\texttt{=}, \texttt{=..}\}, \neg fixed(x, \theta, \theta')$;
2. $\widetilde{S} = \boxed{\theta\ \ldots \diamond\ x\ \theta'}, \diamond \in \{\texttt{=}, \texttt{=..}\}, \neg fixed(x, \theta, \theta')$;
3. $\widetilde{S} = \boxed{\theta\ \Im(x)\ \theta'}, \neg fixed(x, \theta, \theta')$;
4. $\widetilde{S} = \boxed{\theta\ x\ \texttt{is}\ f_i^j(\ldots)\ \theta'}, \neg fixed(x, \theta, \theta')$;
5. $\widetilde{S} = \boxed{\theta\ p(\ldots x \ldots)\ \theta'}, \neg system(p(\ldots x \ldots)), \neg fixed(x, \theta, \theta')$.

In this definition, a variable is considered to have its content possibly changed in $\widetilde{S}$ if it appears in one of the subgoals depicted above and does not remain fixed: it can either satisfy the *change* or the *unknown* relationships. The first and second cases represent those subgoals employing operators `=` or `=..` with a variable x on the left (first case) or right (second case) of the operator, such that its tokens do not remain fixed. The third case represents those subgoals employing a system input predicate `read` or `get` (denoted by $\Im$) such that its variable x is not fixed. The fourth case depicts those subgoals making use of `is` such that the variable x on its left-hand side is not fixed. The last case depicts those user-defined predicates with at least one variable not fixed.

The cases above provide our method with a precise yet general characterisation of the assignment of a value to a variable, using the available information (syntax of subgoal and mode-annotations). The definition lists all system predicates in which assignments (represented as changes in the instantiation status of variables) can be reported, as well as a formalisation of those user-defined predicates which may be assignments. The list of cases above is comprehensive: only those mode-annotated subgoals described will be considered as changing the contents of variables. All other system predicates (such as `>`, `<`, etc.) and user-predicates which do not match one of the cases of the above list will not be considered as subgoals instantiating variables.

The definition above uses the syntax of the system predicates and their meaning to overcome the limitations of inaccurate mode-annotations. By means of it, we can correctly infer that in a subgoal of the form

$$\{\texttt{X/f}, \texttt{Y/i}, \texttt{Z/i}\}\ \texttt{X is Y + Z}\ \{\texttt{X/g}, \texttt{Y/i}, \texttt{Z/i}\}$$

in spite of the low accuracy of its mode-annotations, `Y` and `Z` do not have their contents changed, because this is how the system predicate `is` works. The system predicates `=` and `=..` (used both as comparators and to assign values to variables) are not always guaranteed to work. The adopted policy in such circumstances is that if there is a possibility that a change might have happened, then we shall assume it did happen. For instance, a subgoal of the form

$$\{\texttt{X/i}, \texttt{Y/i}\}\ \texttt{X =..Y}\ \{\texttt{X/i}, \texttt{Y/i}\}$$

will be considered to have changed the contents of *both* `X` and `Y`. This liberal approach, of course, may incorrectly consider a subgoal to be changing the content of a variable, when in fact it is not.

One could argue that the last case of the previous definition is general enough (if the predicate is not restricted to be a non-*system* one) to cover all the previous cases and any other missing ones. However, we characterise, in the most precise manner, the cases where changes might have occurred. If only the last case was employed (without the non-*system* restriction), the `is` sample subgoal shown above would be considered as if changes had happened in the right-hand side of the operator; this is not possible, given the possible behaviours of `is`. All the other system predicates left out of the list above (*e.g.*, `>`, `<`, `write`, etc.) never perform changes in the contents of their variables.

One should notice that changes in variables within terms due to the matching of the term with another variable are not covered by the definition above. The first case of Def. 4.14 only addresses the situation in which a term is assigned to a variable x. The lack of a case covering changes within terms is in accordance with our proposed view of a technique which, together with the syntactic restrictions of the normal form of the procedures being analysed, disregards the matching between variables and terms as relevant to the technique.

Having established the conditions a subgoal must fulfil to ensure it is an instantiation, we now proceed to formalise when such subgoals are relevant to an argument slice. A subgoal is considered a relevant computation if one of its variables with non-fixed tokens is related to the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$ of the i-th argument slice:

Definition 4.15 If a subgoal $\widetilde{S}$ in $\widetilde{C}$ has a variable x possibly being changed, $change(x, \widetilde{S})$, then it is c-relevant to argument slice i, $relevant_i^c(\widetilde{S}, \widetilde{C})$, if one of the conditions below holds:

1. x is one of the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$, that is, $x \in \{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\}$;
2. one of the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$ is $\lhd$-related to each variable y $\lhd^*_{\rho_{\widetilde{C}}}$-related to x:

$$y \lhd x \in \rho_{\widetilde{C}}, \{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\} \lhd^*_{\rho_{\widetilde{C}}} \{y\}$$

The first condition depicts those subgoals potentially instantiating one of the variables $x_i, x_{[0,i]}$, $\ldots, x_{[r,i]}$. Such subgoals may be providing a technique with its final result (x_i is being instantiated) or computing the value of its recursive calls ($x_i^0, \ldots$ or x_i^r are being instantiated).

The second condition describes those subgoals computing intermediate values x employed to instantiate $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$. These intermediate values are relevant to the argument slice if they are employed by $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$. In other words, the variables y to which x "donates" its value, on their turn, "donate" their value to $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$.

Example: Given the mode-annotated *sum/2* procedure shown before (presented here with the sets $\rho_{\widetilde{C}_i}$ of each clause):

`sum(A,B):-`	`{A/g,B/f}`	$\rho_{\widetilde{C}_1} = \emptyset$
`    A = [],`	`{A/g,B/f}`	
`    B = 0,`	`{A/g,B/g}`	
`sum(A,B):-`	`{A/g,B/f,C/f,D/f,E/f}`	
`    A = [C|D],`	`{A/g,B/f,C/g,D/g,E/f}`	$\rho_{\widetilde{C}_2} = \{\texttt{C} \lhd \texttt{A}, \texttt{D} \lhd \texttt{A},$
`    sum(D,E),`	`{A/g,B/f,C/g,D/g,E/g}`	
`    B is E + C,`	`{A/g,B/g,C/g,D/g,E/g}`	$\texttt{B} \lhd \texttt{E}, \texttt{B} \lhd \texttt{C}\}$

Then it is the case that:

1. for $\widetilde{S} = \boxed{\texttt{\{A/g,B/f\} A = [] \{A/g,B/f\}}}$, $\neg relevant^c_i(\widetilde{S}, \widetilde{C}_1), i = 1, 2$: this mode-annotated subgoal does not perform any change in the content of `A`, hence there is no variable x, $change(x, \widetilde{S})$ and thus $\widetilde{S}$ is not c-relevant to any argument slice.

2. for $\widetilde{S} = \boxed{\texttt{\{A/g,B/f\} B = 0 \{A/g,B/g\}}}$, we have that $\neg relevant^c_1(\widetilde{S}, \widetilde{C}_1)$ (`B` is the only variable of this subgoal, and it is such that $change(\texttt{B}, \widetilde{S})$, however `B` does not satisfy any of the cases of Def. 4.15 for the first argument slice, hence the negative result) and $relevant^c_2(\widetilde{S}, \widetilde{C}_1)$ (for the second argument slice `B` satisfies the first case of Def. 4.15, and hence the result).

3. $\neg relevant^c_i(\boxed{\texttt{\{...,A/g,C/f,D/f,...\} A = [C|D] \{...,A/g,C/g,D/g,...\}}}, \widetilde{C}_2), i = 1, 2$: this subgoal describes a data structure decomposition and is not covered by the cases of Def. 4.14, hence the subgoal is not c-relevant to any argument slice.

4. for $\widetilde{S} = \boxed{\texttt{\{...,B/f,C/g,E/g,...\} B is E + C \{...,B/g,C/g,E/g,...\}}}$, $\neg relevant^c_1(\widetilde{S}, \widetilde{C}_2)$ (`B` is the only changing variable, but as the result 2 above, it does not satisfy any of the case of Def. 4.15 for the first argument slice, and hence this negative result) and $relevant^c_2(\widetilde{S}, \widetilde{C}_2)$ (as result 3 above).

4.2.4 Relevance of Tests

Test subgoals are important to our notion of a programming technique because they help to establish the flow of control of a procedure, or alternatively, they have the potential to interfere with it. When the content of a variable x may cause a test subgoal to fail, then, according to our view of a programming technique, this subgoal is relevant to those argument slices providing the value for x or employing x in their computations.

Definition 4.16 A variable x (possibly) has its content tested in subgoal $\widetilde{S}$, $test(x, \widetilde{S})$, if, and only if, one of the cases below holds:

1. $\widetilde{S} = \boxed{\theta\ x\ \diamond\ \ldots\ \theta'}$, $\diamond \not\in \{\texttt{=}, \texttt{=..}\}$;

2. $\widetilde{S} = \boxed{\theta\ \ldots \diamond\ x\ \theta'}$, $\diamond \not\in \{\texttt{=}, \texttt{=..}\}$;

3. $\widetilde{S} = \boxed{\theta\ x\ \diamond\ f^j_i(\ldots)\ \theta'}$, $\diamond \in \{\texttt{=}, \texttt{=..}\}$, $\neg change(x, \theta, \theta')$;

4. $\widetilde{S} = \boxed{\theta\ x\ \diamond\ y\ \theta'}$, $\diamond \in \{\texttt{=}, \texttt{=..}\}$, $\neg change(x, \theta, \theta')$, $\neg change(y, \theta, \theta')$;

5. $\widetilde{S} = \boxed{\theta\ y\ \diamond\ x\ \theta'}$, $\diamond \in \{\texttt{=}, \texttt{=..}\}$, $\neg change(x, \theta, \theta')$, $\neg change(y, \theta, \theta')$ (same as above, but x is now on the right-hand side of the operator);

6. $\widetilde{S} = \boxed{\theta\ \P(x)\ \theta'}$;

7. $\widetilde{S} = \boxed{\theta\ \Im(x)\ \theta'}$, $\neg change(x, \theta, \theta')$;

8. $\widetilde{S} = \boxed{\theta\ x\ \texttt{is}\ f_i^j(\ldots)\ \theta'}$, $\neg change(x, \theta, \theta')$;

9. $\widetilde{S} = \boxed{\theta\ y\ \texttt{is}\ f_i^j(\ldots x \ldots)\ \theta'}$, $\neg change(x, \theta, \theta')$ (same as above, but here x is one of the variables in the expression);

10. $\widetilde{S} = \boxed{\theta\ p(\ldots, x, \ldots)\ \theta'}$, $user\text{-}test(\widetilde{S}, \tau)$, $\neg change(x, \theta, \theta')$.

In this definition, a variable is considered to have its content unchanged in $\widetilde{S}$ if it appears in one of the subgoals depicted above and does not satisfy the relation *change*: it can either satisfy the *fixed* or the *unknown* relationships. The first and second cases represent those subgoals employing the $\diamond$ system operators different from `=` or `=..` (that is, `>`, `<`, etc.) with a variable x on the left (first case) or right (second case) of the operator: these subgoals are obviously tests. The third case represents those subgoals making use of `=` to carry out a data structure decomposition: if the variable x on its left-hand side does not change then the subgoal is testing the content of x against the pattern $f_i^j(\ldots)$. The fourth and fifth cases depicts the subgoals employing operators `=` or `=..` with a non-changing variable in each of its sides. The sixth case stands for those subgoals employing the system predicates $\P$ (*e.g.* `var`, `ground`, `integer`, etc.). The seventh case depicts subgoals using the system input predicates `read` or `get` with a non-changing variable. The eigth case represents those subgoals making use of `is` such that the variable x on its left-hand side does not change — the `is` operator is being employed to calculate the expression $f_i^j(\ldots)$ and test its final value against the content of x. The ninth case depicts the same subgoals as the previous case, but alternatively considers a variable x within the expression to be the the one being tested. The last case shows the user-defined test subgoals.

The cases above provide a precise characterisation of those subgoals which may have been used as tests, potentially changing the flow of control of the procedure. Inaccurate mode-annotations satisfy the $\neg$*change* relationship, this being in accordance with our adopted policy.

Test subgoals are relevant if at least one of these variables satisfies certain properties, given by the definition below:

Definition 4.17 A subgoal $\widetilde{S}$ in $\widetilde{C}$ testing a variable x, $test(x, \widetilde{S})$, is t-relevant to argument slice i, $relevant_i^t(\widetilde{S}, \widetilde{C})$, if one of the conditions below holds:

1. x is one of the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$, that is, $x \in \{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\}$,
2. one of the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$ is $\lhd^*_{\rho_{\widetilde{C}}}$-related to each variable y $\lhd$-related to x, that is
$$y \lhd x \in \rho_{\widetilde{C}}, \{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\} \lhd^*_{\rho_{\widetilde{C}}} \{y\}$$
3. all the variables y providing values to x are $\lhd^*_{\rho_{\widetilde{C}}}$-related to $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$, that is,
$$x \lhd y \in \rho_{\widetilde{C}}, y \lhd^*_{\rho_{\widetilde{C}}} \{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\}$$

The first condition addresses those cases where the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$ of the i-th argument slice is actually being used to perform a test — when this happens, the subgoal is relevant for it may interfere with the flow of control. The second case covers the situation when x is being tested and it provides values for $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$. The third case covers the situation when all those variables which contributed to the value of x had their values provided by $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$.

Example: Given the mode-annotated version of *sum/2* shown previously, with its sets $\rho_{\widetilde{C}_i}, i = 1, 2$, then it is the case that:

1. for $\widetilde{S}$ = $\boxed{\texttt{\{A/g,B/f\} A = [] \{A/g,B/f\}}}$, $relevant_1^t(\widetilde{S}, \widetilde{C}_1)$ (for $test(\texttt{A}, \widetilde{S})$ and the first condition of the previous definition holds) and $relevant_2^t(\widetilde{S}, \widetilde{C}_1)$ (there is only one x such that $test(x, \widetilde{S})$ but $x \in \{x_j, x_{[0,j]}, \ldots, x_{[r,j]}\}$ and hence the negative result).

2. for $\widetilde{S}$ = $\boxed{\texttt{\{A/g,B/f\} B = 0 \{A/g,B/g\}}}$, $\neg relevant_i^t(\widetilde{S}, \widetilde{C}_1), i = 1, 2$ (there is no variable x such that $test(x, \widetilde{S})$ and hence the subgoal is not t-relevant to any argument slice).

3. for $\widetilde{S}$ = $\boxed{\texttt{\{\ldots,A/g,C/f,D/f,\ldots\} A = [C|D] \{\ldots,A/g,C/g,D/g,\ldots\}}}$, $relevant_1^t(\widetilde{S}, \widetilde{C}_2)$ ($test(\texttt{A}, \widetilde{S})$ and the first condition of the previous definition holds) and $\neg relevant_2^t(\widetilde{S}, \widetilde{C}_2)$ (there is only one variable x satisfying the relation $test(x, \widetilde{S})$ but $x \in \{x_j, x_{[0,j]}, \ldots, x_{[r,j]}\}$).

4. for $\widetilde{S}$ = $\boxed{\texttt{\{\ldots,B/f,C/g,E/g,\ldots\} B is E + C \{\ldots,B/g,C/g,E/g,\ldots\}}}$, $\neg relevant_i^t(\widetilde{S}, \widetilde{C}_2), i = 1, 2$ (there is no variable x such that $test(x, \widetilde{S})$).

Some may object to the second condition of Def. 4.17 above, for it considers important for an argument slice (and hence for a technique) those subgoals that test intermediate values employed by the argument slice. The two argument slices of the mode-annotated *collect/2* (Figure 7) are shown in Figure 11. The `integer` system predicate (enclosed in a box) appears in both argument

```
         collect(A,B):-          θ[1,0]
θ[1,1]     A = [],               θ'[1,1]
θ[1,2]     B = [].               θ'[1,2]
         collect(A,B):-          θ[2,0]
θ[2,1]     A = [X|Xs],           θ'[2,1]
θ[2,2]     B = [X|Ys],           θ'[2,2]
θ[2,3]     integer(X),           θ'[2,3]
θ[2,4]     collect(Xs,Ys).       θ'[2,4]
         collect(A,B):-          θ[3,0]
θ[3,1]     A = [X|Xs],           θ'[3,1]
θ[3,2]     collect(Xs,B).        θ'[3,2]
```

```
         collect(A):-            θ[1,0]
θ[1,1]     A = [].               θ'[1,1]
         collect(A):-            θ[2,0]
θ[2,1]     A = [X|Xs],           θ'[2,1]
θ[2,3]     integer(X),           θ'[2,3]
θ[2,4]     collect(Xs).          θ'[2,4]
         collect(A):-            θ[3,0]
θ[3,1]     A = [X|Xs],           θ'[3,1]
θ[3,2]     collect(Xs).          θ'[3,2]
```

```
         collect(B):-            θ[1,0]
θ[1,2]     B = [].               θ'[1,2]
         collect(B):-            θ[2,0]
θ[2,2]     B = [X|Ys],           θ'[2,2]
θ[2,3]     integer(X),           θ'[2,3]
θ[2,4]     collect(Ys).          θ'[2,4]
         collect(B):-            θ[3,0]
θ[3,2]     collect(B).           θ'[3,2]
```

Figure 11: Mode-Annotated Argument Slices of Procedure *collect/2*

slices. In the second argument slice this subgoal does not alter the flow of control, but it works as a contraint on the variable `X` used to build the list. If we drop the second condition of Def. 4.17, then the `integer` subgoal would not appear in the second argument slice.

4.3 Argument Slices of Mode-Annotated Procedures

A mode-annotated procedure with arity n has n argument slices. In order to obtain argument slice i each clause is analysed separately: first its set $\rho_{\widetilde{C}}$ of $\lhd$-pairs is prepared and then each subgoal is checked for relationships between its variables and the argument slice i being built.

The slicing of mode-annotated recursive subgoals is straightforward. It consists of restricting the arguments to the variable x_i^j occupying position i in the j-th recursive call. A non-recursive subgoal may either be included or not be included in argument slice i, depending on the properties its variables possess and the set $\rho_{\widetilde{C}}$, as explained before.

The i-th argument slice of a vector of mode-annotated subgoals is obtained by checking the c- or t-relevance of each subgoal: if the subgoal is relevant, then it is put into the argument slice; if it is not relevant a `true` subgoal replaces it:

Definition 4.18 Given a vector $\vec{S} = \widetilde{S}_0, \ldots, \widetilde{S}_n$ in $\widetilde{C}$, the vector $\vec{S}_i = \widetilde{S}_{[0,i]}, \ldots, \widetilde{S}_{[n,i]}$, of subgoals relevant to argument slice i is such that $\widetilde{S}_{[j,i]} = \widetilde{S}_j$, if $relevant_i^c(\widetilde{S}_j, \widetilde{C})$ or $relevant_i^t(\widetilde{S}_j, \widetilde{C})$; otherwise $\widetilde{S}_{[j,i]} = \mathtt{true}$.

The insertion of `true` predicates is due to a notational convenience. We shall assume that these `true` predicates are discarded: if a vector $\vec{S}$ contains a subgoal `true`, that is, part of it is of the form "$\ldots, \widetilde{S}_k, \mathtt{true}, \ldots$" or "$\ldots, \mathtt{true}, \widetilde{S}_{k+1}, \ldots$", then this part is rewritten as "$\ldots, \widetilde{S}_k, \ldots$".

Definition 4.19 Given $\widetilde{C}$, its i-th mode-annotated argument slice, $\widetilde{C}_i$, is of the form

$$
\begin{array}{ll}
p(x_i) \;\text{:-} & \theta_0 \\
& \vec{S}_{[0,i]} \\
& \theta_0^p \; p(x_{[0,i]}) \; {\theta'_0}^p \\
& \vec{S}_{[1,i]} \\
& \vdots \\
& \vec{S}_{[r,i]} \\
& \theta_r^p \; p(x_{[r,i]}) \; {\theta'_r}^p \\
& \vec{S}_{[r+1,i]}.
\end{array}
$$

Here we shall also assume that spurious `true` predicates are eliminated, and if $\widetilde{C}$ is of the form "$\widetilde{H}$:- $\ldots, \widetilde{S}_k, \mathtt{true}, \widetilde{S}_{k+1}, \ldots$" it is rewritten as "$\widetilde{H}$:- $\ldots, \widetilde{S}_k, \widetilde{S}_{k+1}, \ldots$". If $\widetilde{C}$ is of the form $\widetilde{H}$:- `true` then it is rewritten as $\widetilde{H}$ only.

Definition 4.20 Given a mode-annotated procedure $\widetilde{P}$ with arity n and clauses $\widetilde{C}_1, \ldots, \widetilde{C}_m$, then its i-th mode-annotated argument slice $\widetilde{P}_i$, $1 \leq i \leq n$, is comprised of clauses $\widetilde{C}_{[1,i]}, \ldots, \widetilde{C}_{[m,i]}$.

4.4 Termination and Correctness of the Argument Slicing

In order to obtain the i-th argument slice of a mode-annotated procedure $\widetilde{P}$ each mode-annotated subgoal must be checked for its c- or t-relevance (Defs. 4.15 and 4.17, respectively). The c- and t-relevance checking relies on the $\lhd^*_{\rho_{\widetilde{C}}}$-relationships between the variables of the subgoal and those variables of the recursive calls and head goal of the argument slice. There are search procedures (depth- or breadth-first graph search) for checking $\lhd^*_{\rho_{\widetilde{C}}}$-relationships which always terminate for finite sets $\rho_{\widetilde{C}}$; since we are dealing with a finite number of subgoals in each clause this will always be the case here. The checking for a user-test (Def. 4.13) is also clearly finite, given a finite set τ of user-defined tests. Thus the argument slicing of a mode-annotated clause always terminate, and since there is only a finite number of clauses in a procedure, the argument slicing of a mode-annotated procedure always terminate.

The result of the relevance analysis of each subgoal is such that when a mode-annotated subgoal is considered not c- or t-relevant then, in spite of the accuracy of its mode-annotations, this result is correct in the sense that the subgoal does not contribute to the argument slice (as explained in the beginning of this section). If, however, the outcome of the relevance analysis is positive ($\widetilde{S}$ is c- or t-relevant) it might be the case that, due to inaccurate mode-annotations the adopted policy to cope with them, the subgoal does not really contribute to the argument slice; if the same analysis were performed with more accurate mode-annotations, the subgoal would not be considered relevant.

The source of this imprecision is the third case of Def. 4.17, and inaccurate mode-annotations when unknown variables are considered to be fixed. The other cases overlap with those cases of Def 4.15, and since an unknown variable is either fixed or changing, no mistakes are introduced. However, if we assume that an unknown variable of a mode-annotated subgoal $\widetilde{S}$ is fixed and the third case of Def. 4.17 causes $\widetilde{S}$ to be considered t-relevant, a mistake might have been introduced.

4.5 Final Remarks on the Argument Slicing

The mode-annotations complement the syntax of the subgoal with the manner in which it has been used during the execution of the clause. It would still be possible to carry out the argument slicing without the mode-annotations, relying on the syntax of the subgoal only. Given a procedure to be argument-sliced, the worst scenario is that in which no mode-annotations are provided, and we can only rely on the syntax of the subgoals to carry out the analysis. Unrelated subgoals will be considered as part of the argument slice, compromising the extraction of the techniques in the procedure.

At the other extreme, the best scenario is that in which high-quality mode-annotations are provided, giving an accurate description of how each subgoal has been used. Accurate mode-annotations provide an account of changes in the contents of each variable and this information is essential when discarding subgoals which are irrelevant for the argument slice. Lower quality mode-annotations do not tell much about actual changes in the contents of variables and assumptions have to be made. The adopted policy to cope with inaccurate mode-annotations correctly infers the (non-)relevance of 4 (the overlapping cases of Defs. 4.15 and 4.17) of the 5 cases found.

5 Clause Annotation of Mode-Annotated Argument Slices

The argument slices of a procedure may share variables among themselves. This variable sharing is also part of the programming technique being extracted and must be explicitly represented. We employ, for this purpose, place holders for variables referred across clauses of different argument slices, the *clause annotations*. They are of the form

$$\langle\!\langle \mathit{required}(V) \rangle\!\rangle$$

stating that at that point in the clause the variable symbols in V are required to be instantiated to variables of other argument slices, and also of the form

$$\langle\!\langle \mathit{offer}(W) \rangle\!\rangle$$

indicating that from that point in the clause onwards the set of variable symbols in W is offered to be linked to variables in other argument slices.

Each subgoal of a mode-annotated argument slice receives one annotation of each of the forms above: the *required* annotation before it and the *offer* after it. The sets V and W may be empty. The variable symbols are offered as soon as they are obtained, *i.e.* immediately after the subgoal where they have their instantiation status changed, and each variable symbol is only offered once. Variables are requested, by appearing in a *required* annotation immediately before the first subgoal in which they are employed, and are only requested once. Within each clause a variable may be required or offered, but not both.

The clause annotations state the required and the offered resources, in the form of instantiated variables, of each clause of a mode-annotated argument slice. Due to the implicit quantification of the clauses of a Prolog program, the same symbol stands for different variables if it appears in different clauses. Our approach to isolating programming techniques across each argument position requires that those variable symbols supposed to be the same in distinct argument slices be explicitly linked. A clause- and mode-annotated clause is of the form

$$H \texttt{ :- } \theta_0 \, \gamma'_0 \;\; \gamma_1 \, \theta_1 \;\; S_1 \, \theta'_1 \gamma'_1, \ldots, \gamma_n \, \theta_n \;\; S_n \, \theta'_n \gamma'_n.$$

where γ_i are *required* annotations, γ'_j are *offer* annotations and θ_j and θ'_i are mode-annotations.

5.1 Clause-Annotated Versions of Mode-Annotated Argument Slices

A clause-annotated version of a mode-annotated argument slice is comprised of the clause annotated version of each clause. A mode-annotated clause has its clause-annotated version prepared by obtaining the clause-annotated version of each mode-annotated subgoal. The clause-annotation of a mode-annotated subgoal takes into account the clause annotations inserted so far, *i.e.* between the subgoal being analysed and the head of the clause, thus avoiding multiple occurrences of the same variable in different clause annotations.

The mode annotations play an essential role in the clause annotation. The decision as to whether a variable should be inserted in a clause annotation takes into account the changes (or the lack of changes) of its instantiation status represented by the associated tokens in the mode-annotations before and after the subgoal. If a subgoal has inaccurate mode-annotations (*i.e.*, it has variables whose associated tokens satisfy the relation *unknown*) then it gives rise to more than one clause-annotated version. The different versions correspond to the distinct ways to view those variables with inaccurate mode-annotations.

In the definitions of this section we shall not distinguish between recursive and non-recursive subgoals: $\widetilde{C}$ is of the form $p(x) \texttt{ :- } \theta_0, \widetilde{S}_1, \ldots, \widetilde{S}_n$ where each $\widetilde{S}_i$ is a mode-annotated (recursive or not) subgoal. Initially, we shall provide a definition for a clause-annotated subgoal, taking into account its original mode-annotations (which are preserved) and those clause annotations of previous subgoals. It should be noted that we assume an ordering among the subgoals, from left to right, the same order of the conventional execution of a clause in Prolog.

The clause-annotated version of a mode-annotated argument slice $\widetilde{P}$ is defined as the clause-annotated version of each of its mode-annotated clauses:

Definition 5.1 Given a mode-annotated argument slice $\widetilde{P}$, comprised of mode-annotated clauses $\widetilde{C}_1, \ldots, \widetilde{C}_n$, its *clause-annotated version*, $\widetilde{\widetilde{P}}$, is comprised of the clause-annotated version of each mode-annotated clause, $\widetilde{\widetilde{C}}_1, \ldots, \widetilde{\widetilde{C}}_n$.

The clause-annotated version of a mode-annotated clause is comprised of the clause-annotated version of each subgoal, plus an initial *offer* annotation:

Definition 5.2 Given $\widetilde{C}$, its *clause-annotated version*, $\widetilde{\widetilde{C}}$, is of the form

$$p(x) \texttt{ :- } \theta_0 \, \langle\!\langle \mathit{offer}(W_0) \rangle\!\rangle \; \widetilde{\widetilde{S}}_0, \ldots, \widetilde{\widetilde{S}}_n$$

where $W_0 = \{x\}$, if $\neg \mathit{change}(x, \theta_0, \theta'_n), 1 \leq i \leq n$; otherwise $W_0 = \emptyset$.

This clause annotation offers the non-changing variable in the head goal: from that point in the clause onwards x can be employed in the computations of other argument slices; otherwise W_0 is empty. When the mode-annotated clause $\widetilde{C}$ is a fact of the form $p(x)\,\theta$, that is, there are no mode-annotated subgoals in its body[3], its clause annotated version is of the form

$$p(x)\,\theta \; \langle\!\langle \mathit{offer}(\{x\}) \rangle\!\rangle$$

[3] This will only happen if the instantiation status of x does not change, otherwise the subgoal performing the change would have appeared in the body.

Definition 5.3 Given $\widetilde{S}_i$ its *clause-annotated version* $\widetilde{\widetilde{S}}_i$ is

$$\langle\!\langle \mathit{required}(V_i) \rangle\!\rangle \;\; \theta_i \;\; q(\ldots, y_1, \ldots, y_m, \ldots) \;\; \theta'_i \;\; \langle\!\langle \mathit{offer}(W_i) \rangle\!\rangle$$

where

1. $y_l \in V_i, \neg change(y_l, \theta_p, \theta'_p), y_l \notin V_j, y_l \notin W_k, 0 < j < i, 0 \leq k < i, 1 \leq p \leq n$;
2. $y_l \in W_i, \neg fixed(y_l, \theta_i, \theta'_i), y_l \notin W_j, 0 \leq j < i, y_l \notin V_k, 0 < k \leq i$.

The set V_i of required variables of S_i is built (first case above) by collecting all those variables whose associated tokens do not satisfy the *change* relation in any subgoal of that clause, if they are not already in a previous *required* or *offer* annotation. The set W_i of offered variables of S_i is comprised (second case) of those variables whose associated tokens do not satisfy the *fixed* relation, and are not already in any previous clause-annotation including the *required* annotation of S_i itself.

The definitions above provide the basis for the algorithm *ClauseAnnotate* which, given a mode-annotated argument slice $\widetilde{P}$, obtains a clause-annotated version $\widetilde{\widetilde{P}}$ of it:

Algorithm 5.1 *ClauseAnnotate*

input: Mode-annotated argument slice $\widetilde{P}$ consisting of clauses $\widetilde{C}_1, \ldots, \widetilde{C}_n$;

output: Clause-annotated version $\widetilde{\widetilde{P}}$ consisting of clauses $\widetilde{\widetilde{C}}_1, \ldots, \widetilde{\widetilde{C}}_n$

method: Obtain the clause-annotated version of each subgoal $\widetilde{\widetilde{S}}_j$, in the order they appear in each clause and use them to compose each clause-annotated clause $\widetilde{\widetilde{C}}_i$

begin
  for $i := 1$ **to** n **do**
    if $\widetilde{C}_i = p(x)\,\theta$
    then $\widetilde{\widetilde{C}}_i = p(x)\,\theta\,\langle\!\langle \mathit{offer}(\{x\}) \rangle\!\rangle$
    else if $\widetilde{C}_i = p(x)$:- $\theta_0\ \widetilde{S}_1, \ldots, \widetilde{S}_m$ **then**
      begin
        if $\neg change(x, \theta_0, \theta'_m)$ **then** $W_0 = \{x\}$
                        else $W_0 = \emptyset$
        for $j := 1$ **to** m **do**
        begin
          $V_j := \emptyset, W_j := \emptyset$
          if $\widetilde{S}_j = \theta_j\ q(\ldots, y_1, \ldots, y_p, \ldots)\ \theta'_j$ **then**
          begin
            for $k := 1$ **to** p **do**
              if $\neg fixed(y_k, \theta_j, \theta'_j)$
              then $W_j := W_j \cup \{y_k\}$
              else if $\forall l, 1 \leq l \leq m, \neg change(y_k, \theta_l, \theta'_l)$
                then $V_j := V_j \cup \{y_k\}$
            for $l := 1$ **to** $j - 1$ **do**
            begin
              $V_j := V_j - (V_j \cap V_l), W_j := W_j - (W_j \cap V_l)$
            end
            for $l := 0$ **to** $j - 1$ **do**
            begin
              $V_j := V_j - (V_j \cap W_l), W_j := W_j - (W_j \cap W_l)$
            end
            $W_j := W_j - (W_j \cap V_j)$
            $\widetilde{\widetilde{S}}_j := \langle\!\langle \mathit{required}(V_j) \rangle\!\rangle\ \theta_j\ q(\ldots, y_1, \ldots, y_p, \ldots)\ \theta'_j\ \langle\!\langle \mathit{offer}(W_j) \rangle\!\rangle$
          end
        end
        $\widetilde{\widetilde{C}} := p(x)$:- $\theta_0\ \langle\!\langle \mathit{offer}(W_0) \rangle\!\rangle\ \widetilde{\widetilde{S}}_1, \ldots, \widetilde{\widetilde{S}}_m$
      end
end

Line 4 of algorithm *ClauseAnnotate* copes with facts; clauses with non-empty bodies are dealt with in lines 5–28. The loop of lines 9–30 prepares a clause-annotated version of each mode-annotated subgoal in the body of a clause: firstly the algorithm obtains the sets of non-changing and non-fixed variables in the mode-annotated subgoals and assign these sets, respectively, to V_j, the set of required variables, and W_j, the set of offered variables (lines 14–18). After this the algorithm checks for intersections between the recently obtained sets V_j and W_j with the previous sets V_l and W_l (lines 19–26) and intersections between W_j and V_j (line 27). One should notice the different lower limit of the loops in 19–22 and 23-26: this is because there is a W_0 set, but not a V_0.

Example: The clause and mode-annotated argument slices of *collect/2* are shown in Figure 12 (for the sake of brevity the clause-annotations with empty sets are omited).

	`collect(A):-`	$\theta_{[1,0]}$	$\langle\langle$*offer*$(\{\texttt{A}\})\rangle\rangle$
$\theta_{[1,1]}$	`A = [].`	$\theta'_{[1,1]}$	
	`collect(A):-`	$\theta_{[2,0]}$	$\langle\langle$*offer*$(\{\texttt{A}\})\rangle\rangle$
$\theta_{[2,1]}$	`A = [X|Xs],`	$\theta'_{[2,1]}$	$\langle\langle$*offer*$(\{\texttt{X},\texttt{Xs}\})\rangle\rangle$
$\theta_{[2,3]}$	`integer(X),`	$\theta'_{[2,3]}$	
$\theta_{[2,4]}$	`collect(Xs).`	$\theta'_{[2,4]}$	
	`collect(A):-`	$\theta_{[3,0]}$	$\langle\langle$*offer*$(\{\texttt{A}\})\rangle\rangle$
$\theta_{[3,1]}$	`A = [X|Xs],`	$\theta'_{[3,1]}$	$\langle\langle$*offer*$(\{\texttt{X},\texttt{Xs}\})\rangle\rangle$
$\theta_{[3,2]}$	`collect(Xs).`	$\theta'_{[3,2]}$	

		`collect(B):-`	$\theta_{[1,0]}$	
	$\theta_{[1,2]}$	`B = [].`	$\theta'_{[1,2]}$	$\langle\langle$*offer*$(\{\texttt{B}\})\rangle\rangle$
		`collect(B):-`	$\theta_{[2,0]}$	
$\langle\langle$*required*$(\{\texttt{X}\})\rangle\rangle$	$\theta_{[2,2]}$	`B = [X|Ys],`	$\theta'_{[2,2]}$	$\langle\langle$*offer*$(\{\texttt{B}\})\rangle\rangle$
	$\theta_{[2,3]}$	`integer(X),`	$\theta'_{[2,3]}$	
	$\theta_{[2,4]}$	`collect(Ys).`	$\theta'_{[2,4]}$	$\langle\langle$*offer*$(\{\texttt{Ys}\})\rangle\rangle$
		`collect(B):-`	$\theta_{[3,0]}$	
	$\theta_{[3,2]}$	`collect(B).`	$\theta'_{[3,2]}$	$\langle\langle$*offer*$(\{\texttt{B}\})\rangle\rangle$

Figure 12: Clause- and Mode-Annotated Argument Slices of Procedure *collect/2*

6 Formalising Extracted Techniques

The clause-annotated argument slices obtained at the end of the last stage above are the building blocks of more complex programming techniques. A programming technique is a subsequence of clause- and mode-annotated argument slices *sharing* variables via clause annotations.

The definitions of this section require only that our constructs (subgoals, clauses and procedures) be clause-annotated. The mode-annotations are not essential in the formalisation proposed here. We shall denote this by dropping the "~" symbol representing the mode-annotations. This should not be taken as a restriction: the definitions and the algorithm shown here can be understood, without significant changes, as employing clause- and mode-annotated constructs.

The first important definition of this section establishes the conditions two clause-annotated argument slices must fulfill in order to be considered (part of) a technique. The definition below states that if there is a clause $\widehat{C}_{[k,i]}$ in $\widehat{P}_i$ requesting a variable (*i.e.* it has a *required* annotation with a non-empty set) which is offered by a clause $\widehat{C}_{[k,j]}$ in $\widehat{P}_j$, then we say that $\widehat{P}_i$ *requires* $\widehat{P}_j$:

Definition 6.1 $\widehat{P}_i$ requires $\widehat{P}_j$, *requires*$(\widehat{P}_i, \widehat{P}_j)$, if there is a $k, 1 \leq k \leq m$, such that

1. $\widehat{C}_{[k,i]} = H_{[k,i]}$:- $\vec{G}_{[k,i]}\langle\langle$*required*$(V)\rangle\rangle\vec{G}'_{[k,i]}$, and

2. $\widehat{C}_{[k,j]} = H_{[k,j]} \,:\!-\, \vec{G}_{[k,j]} \langle\!\langle \mathit{offer}(W) \rangle\!\rangle\ \vec{G}'_{[k,j]}$, or $\widehat{C}_{[k,j]} = H_{[k,j]}\ \theta\ \langle\!\langle \mathit{offer}(W) \rangle\!\rangle$, and

3. $V \cap W \neq \emptyset$

The symbols $\vec{G}_i$ stand for a (possibly empty) sequence of subgoals not relevant to the definition. The second condition shows two possible templates for the clause of $\widehat{P}_j$ because when a clause offers variables it can either have a non-empty body or be a fact.

Each programming technique will be represented as a non-empty sequence of clause-annotated argument slices sharing variables. A sequence S of argument slices is defined recursively as:

1. $\mathsf{S} = \langle\,\rangle$ is a sequence (named the *empty* sequence);

2. $\mathsf{S} = \langle P_i, \mathsf{S}' \rangle$, where P_i is an argument slice, is a sequence, if S' is a sequence.

The results of the argument slicing and clause-annotation will be represented as sequences.

We describe below an algorithm to partition a sequence of clause-annotated argument slices into a set of programming techniques. A programming technique is a sub-sequence of the sequence of all argument slices obtained after the clause annotation stage. We define two auxiliary relations: one to obtain the position of an argument slice with respect to a sequence and another to obtain the result of inserting an argument slice into an existing sequence of argument slices respecting the original ordering.

Definition 6.2 Given a sequence of argument slices $\mathsf{S} = \langle P_1, \ldots, P_n \rangle$ and an argument slice P_j, the *order* of P_j with respect to S, $\mathit{order}(P_j, \mathsf{S}, n)$, is such that:

1. $\mathit{order}(P_j, \langle P_j, \mathsf{S}' \rangle, 1)$.

2. $\mathit{order}(P_j, \langle P_i, \mathsf{S}' \rangle, m+1) \leftarrow P_j \neq P_i \wedge \mathit{order}(P_j, \mathsf{S}', m)$.

The relation *insertion* defined below holds if N is a sequence comprised of those elements of T and the proper insertion (obeying the relative ordering of an initial sequence of which both N and T are sub-sequences) of a new element P_{new}:

Definition 6.3 Given a (possibly empty) sequence T and an argument slice P_{new}, the relation $\mathit{insertion}(P_{new}, \mathsf{T}, \mathsf{N})$ holds if

1. $\mathit{insertion}(P_{new}, \langle\,\rangle, \langle P_{new} \rangle)$.

2. $\mathit{insertion}(P_{new}, \langle P_{new}, T' \rangle, \langle P_{new}, T' \rangle)$.

3. $\mathit{insertion}(P_{new}, \langle P_i, \mathsf{T}' \rangle, \langle P_{new}, P_i, \mathsf{T}' \rangle) \leftarrow$
$$\begin{array}{l} \mathit{initial\text{-}sequence}(\mathsf{S}) \wedge \\ \mathit{order}(P_{new}, \mathsf{S}, n_{new}) \wedge \mathit{order}(P_i, \mathsf{S}, n_i) \wedge n_{new} < n_i. \end{array}$$

4. $\mathit{insertion}(P_{new}, \langle P_i, \mathsf{T}' \rangle, \langle P_i, \mathsf{N}' \rangle) \leftarrow$
$$\begin{array}{l} \mathit{initial\text{-}sequence}(\mathsf{S}) \wedge \\ \mathit{order}(P_{new}, \mathsf{S}, n_{new}) \wedge \mathit{order}(P_i, \mathsf{S}, n_i) \wedge n_{new} \geq n_i \wedge \\ \mathit{insertion}(P_{new}, \mathsf{T}', \mathsf{N}'). \end{array}$$

The definition of *insertion* compares the *order* n_i of the first element P_i of the sequence T (with respect to the initial sequence S) and the *order* n_{new} of P_{new}: if n_{new} is less than n_i (third case) the new sequence N is T with P_{new} inserted before P_i; otherwise (fourth case) P_i is the first element of N and the rest of the sequences must satisfy (recursively) the *insertion* relation. The first case caters for empty sequences and the second case for sequences in which P_{new} is already found.

Finally we define the algorithm to partition the sequence of all clause-annotated argument slices of a procedure into a set of techniques. Each technique is a sub-sequence of the original sequence of clause-annotated argument slices, where the relative ordering of the slices is maintained. The algorithm works by collecting the clause-annotated argument slices sharing variables, and assembling a sequence which preserves the original ordering of the argument positions:

Algorithm 6.1 *SeparateTechniques*

input: Initial sequence $\mathsf{S} = \langle \widehat{P}_1, \ldots, \widehat{P}_n \rangle$ of clause-annotated argument slices
output: Set $\mathcal{T} = \{\mathsf{T}_1, \ldots, \mathsf{T}_t\}$ of sub-sequences (techniques) of S
method: Obtain the sub-sequence of arg. slices in S with which each $\widehat{P}_i$ shares values, then obtain the arg. slices sharing values with each component of this subsequence

1 **begin**
2   **for** $i := 1$ **to** n **do**
3   **begin**
4     $\mathsf{T}_i := \langle \widehat{P}_i \rangle$
5   **for** $j := 1$ **to** n **do**
6   **begin**
7   **let** $\mathsf{T}_i = \langle \widehat{P}_{[i,1]}, \ldots, \widehat{P}_{[i,p_1]} \rangle$
8   **for** $k := 1$ **to** p_i **do**
9   **begin**
10   **if** $requires(\widehat{P}_{[i,k]}, \widehat{P}_j)$ **then**
11   **begin**
12           $insertion(\widehat{P}_j, \mathsf{T}_i, \mathsf{T}'_i)$
13           $\mathsf{T}_i := \mathsf{T}'_i$
14   **end**
15   **end**
16   **end**
17     $\mathcal{T} := \mathcal{T} \cup \mathsf{T}_i$
18   **end**
19 **end**

The *SeparateTechniques* algorithm initially sets the sub-sequence T_i to the clause-annotated argument slice $\widehat{P}_i$ (line 4). It then inserts all those argument slices $\widehat{P}_j$ from which the current components of T_i require values (relation *requires*, Definition 6.1) (lines 5–15); the newly obtained T'_i after the insertion of $\widehat{P}_j$ updates T_i and is tested against the remaining elements of S. An alternative declarative version of the *SeparateTechniques* can easily be implemented. The final set $\mathcal{T}$ contains the sub-sequences of S sharing values, without repeated elements.

The set $\mathcal{T}$ is the set of techniques of a procedure. The sequence of clause-annotated argument slices $\widehat{P}_{[i,1]}, \ldots, \widehat{P}_{[i,t]}$ of each element S'_i of $\mathcal{T}$ preserves the same relative ordering as in the initial complete sequence S provided by the argument slicing (stage 2, in Figure 2). By maintaining the original ordering of argument positions we avoid changing the relative position of subgoals (possibly introducing mistakes) when argument slices are joined together again, upon the application of the technique. This consistent ordering also allows for the reuse of common sub-parts of techniques and complex techniques can be defined in terms of simpler ones.

Examples: *a)* If we denote the left-hand side clause- and mode-annotated argument slice of procedure *collect/2* shown in Figure 12 by $\widehat{\widehat{P}}_1$ and the right-hand side by $\widehat{\widehat{P}}_2$, we have the set of techniques

$$\mathcal{T} = \{\langle \widehat{\widehat{P}}_1 \rangle, \langle \widehat{\widehat{P}}_1, \widehat{\widehat{P}}_2 \rangle\}$$

where the first technique is comprised of the first argument slice only (a list being decomposed), and the second technique is both the first and second argument slices (a list being built with those elements of the first list satisfying the *integer/1* predicate).

b) Given the following procedure *p/2*

```
p(A,Y,Count,Sum):-
   A = [X|Xs],
   X = Y,
   Count = 0,
   Sum = 0.
p(A,Y,Count,Sum):-
   A = [X|Xs],
   X \== Y,
   p(Xs,Y,RestCount,RestSum),
   Count is RestCount + 1,
   Sum is RestSum + X.
```

which counts and computes the sum of those elements of a list until a certain element `Y` is found, its clause-annotated argument slices, wrt query

```
?- p([1,2,3,end,4],end,C,S)
```

are (for the sake of brevity the clause annotations with empty sets of variables are omited)

- $\widehat{P}_1 =$

```
                      p(A):-           ⟨⟨offer({A})⟩⟩
                        A = [X|Xs],    ⟨⟨offer({X,Xs})⟩⟩
⟨⟨required({Y})⟩⟩       X = Y.
                      p(A):-           ⟨⟨offer({A})⟩⟩
                        A = [X|Xs],    ⟨⟨offer({X,Xs})⟩⟩
⟨⟨required({Y})⟩⟩       X \== Y,
                        p(Xs).
```

- $\widehat{P}_2 =$

```
p(Y).      ⟨⟨offer({Y})⟩⟩
p(Y):-     ⟨⟨offer({Y})⟩⟩
  p(Y).
```

- $\widehat{P}_3 =$

```
p(Count):-
  Count = 0.                ⟨⟨offer({Count})⟩⟩
p(Count):-
  p(RestCount),             ⟨⟨offer({RestCount})⟩⟩
  Count is RestCount + 1.   ⟨⟨offer({Count})⟩⟩
```

- $\widehat{P}_4 =$

```
                      p(Sum):-
                        Sum = 0.              ⟨⟨offer({Sum})⟩⟩
                      p(Sum):-
                        p(RestSum),           ⟨⟨offer({RestSum})⟩⟩
⟨⟨required({X})⟩⟩       Sum is RestSum + X.   ⟨⟨offer({Sum})⟩⟩
```

The set of techniques $\mathcal{T} = \{\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3, \mathsf{T}_4\}$ is such that

- $\mathsf{T}_1 = \langle \widehat{P}_1, \widehat{P}_2 \rangle$ is a technique defining the flow of execution (decompose a list until a certain element is found);
- $\mathsf{T}_2 = \langle \widehat{P}_2 \rangle$ is a technique carrying a value down the recursive call;
- $\mathsf{T}_3 = \langle \widehat{P}_3 \rangle$ is a technique counting the number of iterations of a loop;
- $\mathsf{T}_4 = \langle \widehat{P}_1, \widehat{P}_2, \widehat{P}_4 \rangle$ is a technique summing the elements provided by technique T_1.

Since the argument slices share the same relative ordering of the original procedure, more complex techniques (such as T_1 and T_4) may use other simpler techniques as part of their definitions.

7 Conclusions

A method to extract the programming techniques of Prolog programs is presented here. Programming techniques are *dynamic* entities characterised by the syntax of the program and how it has been used. They are analysed and extracted with respect to a query . The method employs information on how each subgoal has been used (the *mode-annotations*) to complement its syntax and to partition the procedure into a set of argument slices. The argument slices are the building blocks of programming techniques. Some argument slices are techniques on their own, but they may also be clustered into more complex constructs. The sharing of variables between argument slices is recorded and considered as part of the technique being extracted.

An inaccurate account of the changes in the instantiation status of the variables may have serious negative effects. The syntactic part of a technique is completed with the information concerning the changes in each of its variables. The proper identification of a technique relies strongly on the quality of the information gathered on the first stage, the mode-annotations. The other stages employ this information and will be affected if inaccuracies are present.

Programming techniques, once extracted and formalised as shown here, can be used as input to all those applications explained in the first section. The mode-annotations can enhance the check for correct use of each technique, rejecting its application when the variables do not have the appropriate instantiation status. The formalism presented here can be enhanced with more elaborate forms of representation, such as the one proposed in [Vas92].

References

[BBD+91] P. Brna, P. Bundy, T. Dodd, C. K. Eisenstadt, M. Looi, H. Pain, D. Robertson, B. Smith, and M. Van Someren. Prolog Programming Techniques. *Instructional Science*, 20(2):111–133, 1991.

[Ben94] D. Bental. *Recognising the Design Decisions in Prolog Programs as a Prelude to Critiquing.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.

[Bow92] A. W Bowles. *Detecting Prolog Programming Techniques Using Abstract Interpretation.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1992.

[Bow94] A. W. Bowles. A Techniques Editor for Prolog Novices. Internal software report, available from author., 1994.

[Brn91] P. Brna. Teaching Prolog Techniques. Research Paper 530, Department of Artificial Intelligence, University of Edinburgh, 1991.

[BRV+93] A. W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. Research Paper 641, Department of Artificial Intelligence, University of Edinburgh, 1993. To appear in the *International Journal of Human-Computer Studies.*

[BV93] A. W. Bowles and W. W. Vasconcelos. Characterizing Prolog Programming Techniques. Presented at the Fifth Workshop on Logic Programming Environments, held in conjunction with ILPS'93, Vancouver, British Columbia, Canada, 1993.

[CC92] P. Cousout and R. Cousout. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[Dev90] Y. Devilles. *Logic Programming: Systematic Program Development.* Addison-Wesley, 1990.

[Gab92] D. S. Gabriel. TX: a Prolog Explanation System. Msc dissertation, Department of Artificial Intelligence, University of Edinburgh, 1992.

[KK87] T. Kanamori and T. Kawamura. Analyzing Success Patterns of Logic Programs by Abstract Interpretation. Technical Report 279, ICOT, June 1987.

[KLS89] M. Kirschenbaum, A. Lakhotia, and L. Sterling. Skeletons and Techniques for Prolog Programming. Tr 89-170, Computer Engineering and Science Department, Case Western Reserve University, Ohio, U.S.A., 1989.

[Loo88] C.-K. Looi. *Automatic Program Analysis in a Prolog Intelligent Teaching System.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1988.

[O'K90] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[Rob91] D. Robertson. A Simple Prolog Techniques Editor for Novice Users. In *3rd Annual Conference on Logic Programming*, Edinburgh, Scotland, April 1991. Springer-Verlag.

[SL88] L. Sterling and L. Lakhotia. Composing Prolog Meta-Interpreters. In R. Kowalsky and K. Bowen, editors, *?th International Conference on Logic Programming*, pages 386–403, 1988.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[TS84] H. Tamaki and T. Sato. Unfold/fold Transformations of Logic Programs. In *2nd International Conference on Logic Programming*, 1984.

[Vas92] W. W. Vasconcelos. Formalising the Knowledge of a Prolog Techniques Editor. In *9th Brazilian Symposium on Artificial Intelligence*, Rio de Janeiro, Brazil, October 1992.

[Vas94a] W. W. Vasconcelos. Designing Prolog Programming Techniques. In *Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation (LoPSTr'93).* Springer-Verlag, 1994.

[Vas94b] W. W. Vasconcelos. Extracting Prolog Programming Techniques. Research Paper 715, Department of Artificial Intelligence, University of Edinburgh, 1994. To appear in the *Proceedings of the X Brazilian Symposium on Artificial Intelligence.*

[VVRV93] M. Vargas-Vera, D. Robertson, and W. W. Vasconcelos. Building Large-Scale Prolog Programs using a Techniques Editing System. Research Paper 635, Department of Artificial Intelligence, University of Edinburgh, 1993. Presented as a poster at the ILPS'93, Vancouver, Canada.

[Wae88] A. Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *5th Logic Programming Conference*, Seattle, Oregon, U.S.A., August 1988. MIT Press.