# Extracting Prolog Programming Techniques*

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

**Research Paper 715**

**Abstract**

We present a method of extracting the programming techniques employed in Prolog programs. Our method records the manner each subgoal has been used and employs this, together with its 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

In this work we describe an automated approach to extracting the programming techniques of Prolog programs. A Prolog programming technique is viewed here as the syntax of the program and the manner it has been used: the same constructions in a predicate give 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 program fragment below, for instance,

```
p(... [] ...).
p(... [X|Xs] ...):-
     p(... Xs ...).
```

the argument position shown may be either a technique to build a list or to decompose it, depending on how the predicate is used. 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. These techniques can also involve different argument positions: a program to sum the elements of a list employs in one argument position a technique to decompose a list and in another argument position another technique to perform the actual sum of the elements.

We restrict our attention to programming techniques within a single procedure[1]. Initialisation calls and techniques spread across more than one predicate (*e.g.* mutually

*This paper has been accepted for presentation and publication in the proceedings of the XI Brazilian Symposium on Artificial Intelligence, to be held in Ceará, Brazil, October, 1994. An extended version of it is found in [Vas94a].

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

[1]We shall employ Deville's [Dev90] definition of a *logic procedure* as the sequence of clauses with the same predicate $p^n$ (predicate $p$ with arity $n$) in the head of each of these clauses.

recursive predicates) are outside the scope of this work, as well as any other technique which spans more than one procedure.

In the next subsection we explain the importance of the work presented here and how it interrelates with other research areas. The following subsection describes the syntactic constraints on those programs analysed and lays out some notation and conventions. The second section explains the three stages of the extraction method. The third section shows a manner of formally representing the extracted techniques. The last section summarises the work presented here and discusses its limitations.

### The Importance of Formalising Programming Techniques

Brna *et al* [BBD+91] 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, such as techniques-based editors [Rob91], program tracing [Gab92], program transformation [VVRV93], and automatic program analysis [Loo88, Ben94] (see [BRV+93] for a survey on some of these research topics).

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. The person responsible for devising the set of available techniques chooses, after studying patterns frequently found in programs and acclaimed techniques informally described in Prolog textbooks and papers, a number of techniques and manually encodes them so that the application(s) can use.

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 sharing variables across their clauses. The notation employed to represent the extracted techniques is based on the formalism proposed in [Vas94b]. The extracted techniques can be stored in a library and supplied to each of the applications above.

### 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 `=..`. This normal form allows for the homogeneous treatment of unifications in the head and in body goals, and their descriptions are similar. Moreover, programs in a normal form provide a detailed account of the computations taking place, splitting complex operations into a sequence of simpler 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 by which Prolog constructs can be generically referred to. 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 Extraction of Prolog Programming Techniques

Given a working Prolog program complying with the syntactic restrictions above, our method extracts those programming techniques used in it. The 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 $n$, consisting of 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 1.

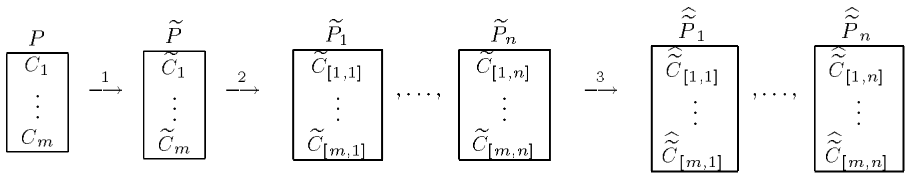

Figure 1: Stages of the Extraction Method and their Intermediate Results: 1 – Mode-Annotation; 2 – Argument Slicing; 3 – Clause-Annotation

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 to across clauses of different argument slices, into the clauses of the mode-annotated argument slices, yielding the sequence $\langle \widehat{\widetilde{P}}_1, \ldots, \widehat{\widetilde{P}}_n \rangle$. 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 Mode-Annotation of Procedures

The first stage carries out the *mode-annotation* of the procedure with the instantiation status of the variables in each subgoal. The mode-annotation collects information about the use of each variable 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.

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., as will be seen below. 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 $H$ :- $\theta_0\ \theta_1\ S_1\ \theta'_1, \ldots, \theta_n\ S_n\ \theta'_n$ where $H$ and $S_i$ are subgoals and the $\theta_j$ and $\theta'_i$ 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 $\theta_0$ contains the status of the variables immediately after the head goal is matched. It is clear that $\theta_0 = \theta_1$ and $\theta'_i = \theta_{i+1}$, but this replication is made necessary because during the argument slicing stage (subsection 2.2) mode-annotated subgoals can be removed and the lack of substitutions would render the mode-annotated clause inaccurate.

During the mode-annotation, the tokens "`f`" (associated with *free* variables), "`g`" (associated with *ground* variables, *i.e.* variables bound to constants or composed terms with ground subterms only), "`i`" (associated with *instantiated* variables, *i.e.* variables *not free*) and "`?`" (associated with variables whose status is unknown) are available. 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. Due to limitations inherent in abstract interpretation techniques, the token "`?`" 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.

### Mode-Annotation via Concrete Interpretation

A very simple approach to mode-annotate a clause is by concrete interpretation: run the program (say, by using an enhanced Prolog meta-interpreter) with the initial query, and collect information before and after the execution of each subgoal. The concrete interpretation of a program may use the same clause many times, obtaining the same mode-annotated version. Repeated clauses, however, are not relevant to the adopted view of techniques: they will be discarded, and only one mode-annotated version of the clause will actually be employed.

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. However, because we are actually running a program while mode-annotations are collected, if the program does not terminate neither does the mode-annotation. Another problem is that 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: the extraction process relies on the participation of a user whose initiative in choosing the program and the query is essential — it would be expected that the user had chosen the program *because* it computed (hence terminated) some interesting result. The incompleteness issue can be circumvented by again relying on the user's choice of an appropriate query: 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 a normal form of the *collect/2* procedure which holds if its first argument is a list whose integer elements (if any) are to be found, in the same order, in the list comprising the second argument. Its (concrete) mode-annotated version wrt query `collect([foo1,foo2,foo3,foo4],L)` is shown in the left-hand side of Figure 2: the second clause of *collect/2* (in which the elements satisfying *integer/2* are used to build the list comprising the second argument) was left out. For the sake of brevity, in Figure 2 we have omitted repeated mode-annotations, $\theta_0$ being reused as $\theta_1$ and $\theta_i'$ as $\theta_{i+1}$:

```
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}
```

```
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}
```

Figure 2: Concrete (Left) and Abstract (Right) Mode-Annotated Versions of procedure *collect/2* with respect to query `collect([foo1,foo2,foo3,foo4],L)`

### Mode-Annotation via Abstract Interpretation

An alternative approach to mode-annotate a procedure is to use *abstract interpretation* [CC92, KK87], and to simulate the actual computations of Prolog in terms of the tokens describing the instantiation of each variable. The mode-annotation based on abstract interpretation eventually terminates, 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. Furthermore, if the same query is supplied to both concrete and abstract interpreters, there is a guarantee that the clauses obtained in the latter form a superset of those obtained in the former.

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

A major deficiency of abstract interpretation in comparison with concrete interpretation is the lower quality of its mode-annotations themselves. During the abstract interpretation the sharing of variables within terms is not recorded and changes in their instantiation status are not propagated. This causes the inaccuracy of the mode-annotations given by the abstract interpreter. We show in the right-hand side of

Figure 2 the mode-annotated version of procedure *collect/2* obtained via abstract interpretation, with respect to the same query used in the concrete interpretation — the last annotation of the third clause is of less quality: `B` ends up associated with "`i`" rather than with "`g`", as in the right-hand side version, but all three clauses of the procedure are considered.

## 2.2 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 clause. This notion of relevance is formally stated in this section.

The analysis performed takes into account 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 fulfil to be included in an argument slice. Different mode-annotations in a procedure may yield different argument slices and hence different techniques. The outcome of this analysis is highly dependent on the quality of the mode-annotations: the more accurate these are, the better the outcome is. The more tokens "`i`" or "`?`" in the mode-annotations, the less accurate is the argument slicing.

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

$$p(x_1, \ldots, x_n) \text{ :- } \vec{S}_0\, \theta^p_0\, p(x_{[0,1]}, \ldots, x_{[0,n]})\, \theta^{p\prime}_0\, \vec{S}_1 \ldots \vec{S}_r\, \theta^p_r\, p(x_{[r,1]}, \ldots, x_{[r,n]})\, \theta^{p\prime}_r\, \vec{S}_{r+1}$$

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 $\widetilde{S}$. Mode-annotated subgoals will sometimes be shown enclosed in boxes to facilitate their visualisation.

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 also play an important role in the analysis of relevance of a subgoal: if the subgoal does not affect (either directly or indirectly) these variables then it should be considered irrelevant. A subgoal affects these variables if it a) *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 $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$, or b) *tests* a variable whose value was obtained, directly or not, from one of the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$. This notion of relevance uses Prolog's own execution model as a criterion: if a subgoal neither *interferes with* nor *contributes to* the argument slice, then the subgoal is not relevant and should not be included in the argument slice.

### Relationships between Variables

In order to decide whether or not a subgoal is relevant we must find out which computations take place in each subgoal and how important each subgoal is for its variables. This is done by studying the mode-annotations before and after 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 the content of the variable did not change. If a variable has tokens "f" and "g" (or "f" and "i") associated with it respectively before and after the subgoal execution, it is correct to assume that its actual content did change. However, for tokens such as "i" and "?", representing supersets of values of other tokens, it is not possible to say with accuracy when changes take place.

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

**Definition 2.1** $\mathit{fixed}(x, \theta, \theta')$ holds if $x/T \in \theta, x/T' \in \theta', T = T', T \in \{\mathtt{g}, \mathtt{f}\}$.

**Definition 2.2** $\mathit{change}(x, \theta, \theta')$ holds if $x/T \in \theta, x/T' \in \theta', T = \mathtt{f}, T' \in \{\mathtt{g}, \mathtt{i}\}$.

**Definition 2.3** $\mathit{unknown}(x, \theta, \theta')$ 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. To formalise 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 such pair for each subgoal and hence a set has been employed to store them:

**Definition 2.4** The set $\rho_{\widetilde{S}}$ of $\lhd$-related pairs (or simply $\lhd$-pairs) of $\widetilde{S}$ is comprised of elements of the form $y_i \lhd y_j$ such that *i)* $\mathit{change}(y_i, \theta, \theta')$ and $\mathit{fixed}(y_j, \theta, \theta')$, or *ii)* $\mathit{unknown}(y_i, \theta, \theta')$ and $\mathit{fixed}(y_j, \theta, \theta')$, or *iii)* $\mathit{change}(y_i, \theta, \theta')$ and $\mathit{unknown}(y_j, \theta, \theta')$, or *iv)* $\mathit{unknown}(y_i, \theta, \theta')$ 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 of representing 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.

A variable may be indirectly related, via an intermediate subgoal, to another variable. To deal with these situations, we extend the definition of $\lhd$-pair 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 2.5** The set $\rho_{\vec{S}}$ of $\lhd$-pairs of $\vec{S} = \widetilde{S}_0, \ldots, \widetilde{S}_n$ is $\rho_{\vec{S}} = \bigcup\limits_{i=0}^{n} \rho_{\widetilde{S}_i}$.

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 2.6** The set $\rho_{\widetilde{C}}$ of $\lhd$-pairs of $\widetilde{C}$ is $\rho_{\widetilde{C}} = \bigcup_{i=0}^{r+1} \rho_{\vec{S}_i}$.

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 pairs (edges).

**Example:** The second clause $\widetilde{C}$ of the right-hand side program of Figure 2 yields $\rho_{\widetilde{C}} = \{\mathtt{X} \lhd \mathtt{A}, \mathtt{Xs} \lhd \mathtt{A}, \mathtt{B} \lhd \mathtt{X}, \mathtt{B} \lhd \mathtt{Ys}\}$

The analysis carried out in the slicing of a mode-annotated procedure may involve indirect $\lhd$-relationships. A variable $y$ is (possibly indirectly) related to $x$ via a set of $\lhd$-pairs if there is a sequence of pairs such that the first pair is of the form $x_1 \lhd y$, any two consecutive elements of the sequence are of the form $x_i \lhd x_j, x_j \lhd x_k$ and the last element is of the form $x \lhd x_n$:

**Definition 2.7** Given a set $\rho$ of $\lhd$-related pairs and two variables $x$ and $y$, the relation $x \lhd^*_\rho y$ holds if *i)* $x \lhd y \in \rho$, or *ii)* $z \lhd y \in \rho$ 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 finding out whether $x \lhd^*_\rho y$ 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. The definition of $\lhd^*_\rho$ above can be extended to cope with sets of variables $V_i$:

**Definition 2.8** $V_1 \lhd^*_\rho V_2$ holds if there is at least one variable $x \in V_1$ and at least one variable $y \in V_2$ such that $x \lhd^*_\rho y$ holds.

### 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 in 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 2.9** 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. $\widetilde{S} = \boxed{\theta\ x \diamond \ldots\ \theta'}, \diamond \in \{\mathtt{=}, \mathtt{=..}\}, \neg fixed(x, \theta, \theta')$;
2. $\widetilde{S} = \boxed{\theta\ \ldots \diamond x\ \theta'}, \diamond \in \{\mathtt{=}, \mathtt{=..}\}, \neg fixed(x, \theta, \theta')$;
3. $\widetilde{S} = \boxed{\theta\ \Im(x)\ \theta'}, \neg fixed(x, \theta, \theta')$;
4. $\widetilde{S} = \boxed{\theta\ x\ \mathtt{is}\ f_i^j(\ldots)\ \theta'}, \neg fixed(x, \theta, \theta')$;
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 fourth case above, for instance, considers those subgoals making use of `is`, such that the variable $x$ on its left-hand side is not fixed, to be changing the content of $x$.

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 2.10** If a subgoal $\widetilde{S}$ in $\widetilde{C}$ has a variable $x$ with its content (possibly) being changed, $change(x, \widetilde{S})$, then it is $c$-relevant to argument slice $i$, $relevant^c_i(\widetilde{S}, \widetilde{C})$, if *i)* $x \in \{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\}$, or *ii)* 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 $x$, that is, $y \lhd x \in \rho_{\widetilde{C}}$, $\{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\} \lhd^*_{\rho_{\widetilde{C}}} \{y\}$.

The first condition depicts those subgoals potentially changing the content of one of the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$. Such subgoals may be providing a technique with its final result or computing the value of its recursive calls. The second condition describes those subgoals computing intermediate values $x$ employed to change the value of $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$. 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 $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$.

### 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. It is not too difficult a task to verify if a system predicate is a test. However, finding out if a user-defined procedure may fail, possibly interfering with the flow of control, is a complex issue, harder than checking if the execution of a procedure terminates. We employ a *shallow* analysis in which the definition of user-defined predicates is not taken into account. Instead, the user should 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. If a mode-annotated subgoal satisfies the relation *user-test* then it is considered a user-defined test. The cases below provide a precise characterisation of those subgoals which may have been used as tests, potentially changing the flow of control of the procedure:

**Definition 2.11** 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. $\widetilde{S} = \boxed{\theta\ x\ \texttt{is}\ f^j_i(\ldots)\ \theta'}, \neg change(x, \theta, \theta')$;
9. $\widetilde{S} = \boxed{\theta\ y\ \texttt{is}\ f^j_i(\ldots x \ldots)\ \theta'}, \neg change(x, \theta, \theta')$ (same as above, but here $x$ is one of the variables in the expression);

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

Inaccurate mode-annotations satisfy the $\neg \textit{change}$ relationship, and thus may introduce their imprecision into the slicing stage. The list of cases above helps us find out which variables in a subgoal are being tested. For instance, the third case above 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 $f_i^j(\ldots)$. 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 2.12** A subgoal $\widetilde{S}$ in $\widetilde{C}$ testing a variable $x$, $\textit{test}(x, \widetilde{S})$, is $t$-relevant to argument slice $i$, $\textit{relevant}_i^t(\widetilde{S}, \widetilde{C})$, if *i)* $x \in \{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\}$; or *ii)* 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, $y \lhd x \in \rho_{\widetilde{C}}, \{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\} \lhd^*_{\rho_{\widetilde{C}}} \{y\}$; or *iii)* 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. 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 $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$.

The two argument slices of the mode-annotated *collect/2* (Figure 2) are shown in Figure 3: the mode-annotations are shown before and after each subgoal, generically represented as $\theta_i$ and $\theta'_j$.

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

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

Figure 3: Mode-Annotated Argument Slices of Procedure *collect/2*

### Argument Slices of Mode-Annotated Procedures

A mode-annotated procedure with arity $n$ yields a sequence of $n$ 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. 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 2.13** 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 $relevant^c_i(\widetilde{S}_j, \widetilde{C})$ or $relevant^t_i(\widetilde{S}_j, \widetilde{C})$; otherwise $\widetilde{S}_{[j,i]} =$ `true`.

The insertion of `true` predicates is due to a notational convenience. We shall assume that these `true` predicates are eliminated.

**Definition 2.14** Given $\widetilde{C}$, its $i$-th mode-annotated argument slice, $\widetilde{C}_i$, is of the form $p(x_i)$ :- $\vec{S}_{[0,i]}\, \theta^p_0\, p(x_{[0,i]})\, \theta^{p\prime}_0\, \vec{S}_{[1,i]} \ldots \vec{S}_{[r,i]}\, \theta^p_r\, p(x_{[r,i]})\, \theta^{p\prime}_r\, \vec{S}_{[r+1,i]}$..

**Definition 2.15** Given a mode-annotated procedure $\widetilde{P} = \widetilde{C}_1, \ldots, \widetilde{C}_m$ with arity $n$, its $i$-th mode-annotated argument slice $\widetilde{P}_i, 1 \leq i \leq n$, is comprised of clauses $\widetilde{C}_{[1,i]}, \ldots, \widetilde{C}_{[m,i]}$.

### Termination and Correctness of the Argument Slicing Stage

In order to obtain the $i$-th argument slice of $\widetilde{P}$ each mode-annotated subgoal must be checked for its $c$- or $t$-relevance (Defs. 2.10 and 2.12, 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 which guarantee that this analysis always terminates for finite sets $\rho_{\widetilde{C}}$ and 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 is also clearly finite, given a finite set of user-defined tests. Thus the argument slicing of a mode-annotated clause always terminates, and since there is only a finite number of clauses in a procedure, the argument slicing of a mode-annotated procedure always terminates.

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 and the policy adopted 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. 2.12 dealing with inaccurate mode-annotations (when unknown variables are considered to be fixed). The other cases overlap with those cases of Def 2.10, and since an unknown variable is either fixed or changing, no mistakes are introduced. However, if we assume that an unknown variable of $\widetilde{S}$ is fixed and the third case of Def. 2.12 causes $\widetilde{S}$ to be considered $t$-relevant, a mistake may be introduced.

## 2.3 Clause-Annotation of Mode-Annotated Argument Slices

Mode-annotated argument slices may have references to variables of other slices. This variable sharing between different argument slices is also part of the programming technique being extracted and must be explicitly represented. We employ, for this purpose, place holders for variables referred to across clauses of different argument slices, the *clause-annotations*. They are of the form "$\langle\!\langle required(V) \rangle\!\rangle$" stating that at that point in the clause the variable symbols in the set $V$ are required to be instantiated to variables of other argument slices, and of the form "$\langle\!\langle offer(V) \rangle\!\rangle$" indicating that from that point in the clause onwards the set of variable symbols in $V$ 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. In both cases $V$ may be empty.

Clause-annotations provide explicit links between argument slices. The sharing of variables between argument positions is another feature of a Prolog programming technique extracted and recorded during this stage. If a subgoal has inaccurate mode-annotations then it gives rise to more than one clause-annotated version, the different versions corresponding to distinct ways of viewing those variables with inaccurate mode-annotations. Here we shall not distinguish between recursive and non-recursive subgoals: $\widetilde{C}$ is of the form $p(x)$ :- $\widetilde{S}_0, \ldots, \widetilde{S}_n$ where each $\widetilde{S}_i$ is a mode-annotated subgoal, recursive or not. A variable may only appear once in the clause-annotations of a clause.

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

**Definition 2.16** Given a mode-annotated clause $\widetilde{C}$, its *clause-annotated version*, $\widehat{\widetilde{C}}$, is of the form $p(x)$ :- $\theta_0 \langle\!\langle \mathit{offer}(W_0) \rangle\!\rangle\, \widehat{\widetilde{S}}_0, \ldots, \widehat{\widetilde{S}}_n$, where $W_0 = \{x\}$ if $\neg \mathit{change}(x, \theta_0, \theta'_n)$; 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, when there are no mode-annotated subgoals in its body[2], its clause-annotated version is of the form $p(x)\ \theta\ \langle\!\langle \mathit{offer}(\{x\}) \rangle\!\rangle$.

**Definition 2.17** Given a mode-annotated subgoal $\widetilde{S}$ its *clause-annotated version* $\widehat{\widetilde{S}}_i$ is of the form $\langle\!\langle \mathit{required}(V_i) \rangle\!\rangle\ \theta_i\ q(\ldots, y_1, \ldots, y_m, \ldots)\ \theta'_i\ \langle\!\langle \mathit{offer}(W_i) \rangle\!\rangle$, where $y_l \in V_i$, $\neg \mathit{change}(y_l, \theta_p, \theta'_p), y_l \notin V_j, y_l \notin W_k, 0 < j < i, 0 \leq k < i, 1 \leq p \leq n$; and $y_l \in W_i$, $\neg \mathit{fixed}(y_l, \theta_i, \theta'_i), y_l \notin W_j, 0 \leq j < i, y_l \notin V_k, 0 < k \leq i$.

The set $V_i$ of required variables of $S_i$ is built 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 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.

An algorithm to obtain the clause-annotated version of a mode-annotated argument slice is given in [Vas94a]. In Figure 4 we show the clause-annotated versions of *collect/2*: the clause-annotations with empty sets are not shown, nor are the clause-annotations whose variables are not referred to in other argument slices.

# 3 Formalising Extracted Techniques

The clause-annotated argument slices obtained at the end of the previous stage are the basic components of a programming technique. A programming technique is the smallest sub-sequence of clause- and mode-annotated argument slices such that all *required* variables appear in an *offer* annotation of some other argument slice in that

[2]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.

```
collect(A):-                              
    A = [].                               
collect(A):-                              
    A = [X|Xs],        《offer({X})》      
    integer(X),                           
    collect(Xs).                          
collect(A):-                              
    A = [X|Xs],                           
    collect(Xs).                          
```

```
                      collect(B):-
                          B = [].
                      collect(B):-
《required({X})》          B = [X|Ys],
                          integer(X),
                          collect(Ys).
                      collect(B):-
                          collect(B).
```

Figure 4: Clause-Annotated Argument Slices of Procedure *collect/2*

sub-sequence. 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 concepts shown here can be understood, without significant changes, as employing clause- and mode-annotated constructs.

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 3.1** $\widehat{P}_i$ requires $\widehat{P}_j$, *requires*$(\widehat{P}_i, \widehat{P}_j)$, if there is a $k, 1 \leq k \leq m$, such that *i)* $\widehat{C}_{[k,i]} = H_{[k,i]}$ :- $\vec{G}_{[k,i]}\langle\!\langle required(V)\rangle\!\rangle\vec{G'}_{[k,i]}$, and *ii)* $\widehat{C}_{[k,j]} = H_{[k,j]}$ :- $\vec{G}_{[k,j]}\langle\!\langle offer(W)\rangle\!\rangle$ $\vec{G'}_{[k,j]}$, or $\widehat{C}_{[k,j]} = H_{[k,j]}\,\theta\,\langle\!\langle offer(W)\rangle\!\rangle$, and *iii)* $V \cap W \neq \emptyset$

The symbols $\vec{G}_i$ stand for a possibly empty sequence of subgoals. 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.

Argument slices with empty sets of required variables are techniques on their own. Our *collect/2* procedure has two techniques: a technique $\mathcal{T}_1 = \langle\widehat{\widetilde{P}}_1\rangle$ corresponding to the first argument slice alone (with no *required* variables) and another technique $\mathcal{T}_2 = \langle\widehat{\widetilde{P}}_1, \widehat{\widetilde{P}}_2\rangle$ involving both argument slices.

**Example:** The following procedure *p/4*

```
p(A,Y,C,S):-
    A = [X|Xs],
    X = Y,
    C = 0,
    S = 0.
p(A,Y,C,S):-
    A = [X|Xs],
    X \== Y,
    p(Xs,Y,RC,RS),
    C is RC + 1,
    S is RS + X.
```

has the following sequence of clause-annotated argument slices (for the sake of brevity, only those clause-annotations cross-referred to in other argument slices are shown) wrt query `p([1,3,end],end,C,S)`:

$\widehat{P}_1 =$

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

$\widehat{P}_2 =$

```
p(Y).          《offer({Y})》
p(Y):-         《offer({Y})》
    p(Y).
```

$\widehat{P}_3 =$

```
p(C):-
  C = 0.
p(C):-
  p(RC),
  C is RC + 1.
```

$\widehat{P}_4 =$

```
                    p(S):-
                      S = 0.
                    p(S):-
                      p(RS),
⟨⟨required({X})⟩⟩     S is RS + X.
```

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 the execution, decomposing 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; and $\mathsf{T}_4 = \langle \widehat{P}_1, \widehat{P}_2, \widehat{P}_4 \rangle$ is a technique summing the elements provided by technique $\mathsf{T}_1$. Since the argument slices share the same relative ordering of the original procedure, more complex techniques (such as $\mathsf{T}_1$ and $\mathsf{T}_4$) may use other simpler techniques as part of their definitions.

In [Vas94a] we define an algorithm to partition the sequence of all clause-annotated argument slices of a procedure into a set of techniques. 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.

The predicate *path/2* shown below, which holds if its second argument is a list containing a path from the head element of the list in the first argument position to a *destination* node, can be enhanced by the appropriate combination of the argument slices comprising the techniques above. For instance, the combination of $P_3$ and *path/2* is straightforward, it only being necessary that the variables be renamed so as to avoid name clashes. The combination of $P_4$, on the other hand, demands the participation of a human user to inform which value the *required* variable `X` in $P_4$ should be bound to in the *path/2* predicate: in the example below the user has chosen for this purpose the cost `C` associated with each edge. The "$\cdot$" symbol stands for a restricted form of a program combination operator, employed by a Prolog Techniques Editor (such as the one described in [Rob91]):

```
path(P,S):-
  P = [N|_],
  destination(N),
  S = P.
path(P,S):-
  P = [N|_],
  edge(N,M,C),
  \+ member(M,P),
  NP = [M|P],
  path(NP,S).
```

$\cdot\ P_3 \cdot P_4 =$

```
path(P,S,C1,S1):-
  P = [N|_],
  destination(N),
  S = P,
  C1 = 0,
  S1 = 0.
path(P,S,C1,S1):-
  P = [N|_],
  edge(N,M,C),
  \+ member(M,P),
  NP = [M|P],
  path(NP,S,C2,S2),
  C1 is C2 + 1,
  S1 is S2 + C.
```

# 4 Conclusions

A method to extract programming techniques from Prolog programs is presented here. Programming techniques are analysed and extracted with respect to a query. The method employs information concerning the usage of the subgoal to complement its syntax and to partition the procedure into a set of argument slices. The argument slices comprise the building blocks of programming techniques. Some argument slices are techniques on their own, but they may also be part of 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 in 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 formalism proposed can be enhanced with more elaborate forms of representation, such as the one proposed in [Vas92].

**Acknowledgements:** Thanks are due to D. Robertson, J. Hesketh and A. Bowles for helpful discussions, to S. Simpson for proof-reading earlier drafts of this paper, and to the anonymous referees for their comments.

## 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.

[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.*

[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.

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

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

[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. A Method of Extracting Prolog Programming Techniques. Technical Paper 27, Department of Artificial Intelligence, University of Edinburgh, 1994.

[Vas94b] 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.

[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.