Extracting, Organising, Designing and Reusing Prolog Programming Techniques

Wamberto Weber M. P. de Vasconcelos

Ph.D.
University of Edinburgh
1995

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

Wamberto Weber M. P. de Vasconcelos
Edinburgh
August 9, 1995

Abstract

Software development is a knowledge-intensive activity in which records of previous similar experiences are in constant demand. The different tasks involved in software development would all benefit if tools embodying knowledge of that activity and of the application domain were made available. A number of such knowledge-based tools have been successfully developed.

The knowledge-based software development tools developed so far assume that the knowledge is somehow encoded and made available, but no further details as to how this is done is actually provided. The preparation of the knowledge may require much labour and ingenuity, for no automated support has been proposed.

We have implemented a semi-automated tool to support the management of the knowledge of a knowledge-based programming environment for Prolog. Our proposal incorporates the notion of *programming techniques*, a specific form of programming knowledge that has been closely connected with Prolog programming. This management tool provides a means to acquire and organise, in the form of a library of Prolog Programming Techniques, the programming knowledge of an expert programmer, making this knowledge accessible to different software development tools.

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Dave Robertson and Jane Hesketh, for their objective advice over the years of my research. Their support was crucial and is most warmly acknowledged.

Special thanks are due to my examiners, Dr. C. Mellish and Dr. N. E. Fuchs, for their useful suggestions, appropriate corrections and for a most exciting intellectual exercise during my Viva.

I owe a big thank-you to Andy Bowles, for sharing with me his expertise in the field and for the stimulating discussions over *cappucini* and *espressi* with *halbbitter* chocolate bars. Thanks are also due to the members of the Knowledge-Based Systems group, Maria Vargas-Vera and Diana Bental, for their invaluable suggestions and fruitful discussions.

Seumas Simpson greatly helped me with friendship and with the "prof-raeding" of my scribbles.

My family and friends back home supplied me with the much needed encouragement, for which I am most grateful. I owe special thanks to my sister Ana Cœli for looking after my affairs in Brazil so well in my absence.

Thanks to my friend Cristina Boeres for putting up with my grumpiest moods and being there when I needed it. A thank-you to all my colleagues in room E17 and E19, present and past incarnations: Alan Black, Alberto Castro, Alistair Knott, Carla Gomes, Chris Gathercole, David Moffat, Edjard Mota, Elena Perez-Miñana, Edward Carter, Flávio Correia da Silva, Ian Frank, Ian Lewin, Ion Androtsopoulos, Jeremy Crowe, Kazuhiro Kado, Keiichi Nakata, Kheyin-How, Kim Binstead, Matt Crocker, Namseog Park, Nelson Ludlow, Rob Scott, Rolando Carreras, Steffan Corley and Suresh Manandhar, for their companionship and help with Unix, Latex, English and Scots, British and Scottish laws and customs, pieces of trivia, and much more. Professor P. L. Broadhurst was a source of excellent advice in all areas of my life, for which a heartfelt thank-you is owed.

I would like to acknowledge the generous financial support provided by the Brazilian people through the Brazilian National Research Council (Conselho Nacional de Pesquisa, CNPq) under grant no. 201340/91-7. Thanks are also due to the staff and lecturers of the State University of Ceará, Department of Statistics and Computing, for backing me up during my leave of absence. I would also like to acknowledge the support provided by the Department of Artificial Intelligence (Student Travel Fund) which enabled me to attend the conferences LoPSTr'93 (Louvain-la-Neuve, Belgium) and XI SBIA (Fortaleza, Ceará, Brazil).

The hardships of bureaucracy in its many forms were soothed by the secretaries Jeannie Bunten, Janet Lee, Jo Clark, Maisha Henderson and Emma Lawson; the efficient library services provided by Olga Franks saved me much time and energy; the conspiracy of non-cooperative technology was effectively sorted out by the computing officers Craig Strachan and Mike Keightley — thanks a lot, folks.

Thanks to the people of Scotland for sharing their beautiful country with me and making me feel at home. Many thanks to the locals of Pease Bay for sharing their "cool" surf-scene, and also a thank-you to the surfers of Thurso Bay (Caithness, Sutherland) for the good vibes and most enjoyable waves.

para Heraldo (In Memoriam)

e

para S., C., T., D., etc.

Contents

Chapter 1

Introduction

Software development is a knowledge-intensive activity in which records of previous similar experiences are in constant demand. Knowledge-based software development tools have been proposed (*e.g.* [Wat92], [BGB91], [Rob91b] and [Bow94]) incorporating different formal and semi-formal methods for software development, all enjoying the common feature of possessing and employing a programming knowledge base.

One way to formalise the programming knowledge base is by means of *programming techniques*. Programming techniques, common code patterns of a programming language [BBD+91] have been extensively studied within the logic programming community. Techniques-based software development tools have been proposed (*e.g.* [Rob91b], [Bow94], [Gab92], [VVRV93], [Loo88b] and [Ben94]), but they all assume that the programming techniques are somehow encoded and made available, and no further details as to how this is done are provided. The preparation of these techniques may require much labour and ingenuity. Furthermore, there is not an agreement among researchers on a set of most generic and useful programming techniques nor how they should be represented.

In this dissertation we describe an implemented tool to manage a knowledge-base consisting of Prolog programming techniques. This tool is used by expert programmers to create and organise a repository of Prolog programming techniques, making this knowledge accessible to different software development tools. Our knowledge-management tool allows the automatic extraction of programming techniques from actual Prolog programs, and offers a natural way to organise these components and to reuse them to define new constructs.

This introductory chapter provides some background information and the context of our work. We propose a software development environment centred around the concept of programming techniques and explain how our tool fits into it.

1.1 Knowledge-based Software Development Tools

A long-term goal of Computer Science has been to determine formal and semi-formal approaches to software development. Tools incorporating these proposals would provide automated support to the development of systems enjoying the desired qualities of reliability, termination (whenever applicable), correctness, responsivity, and so on.

The different tasks involved in software development, such as requirements analysis, project management, implementation, debugging, testing, and so on, would all benefit if tools embodying knowledge of that activity and knowledge of the application domain were made available. The work described in this dissertation, however, is mainly concerned with the task of programming and how knowledge-based tools would enhance the quality of programs and improve the working conditions of programmers: during the implementation stage a large number of avoidable mistakes occur and much time is unnecessarily spent with trivial tasks that can easily be mechanised [Gol86, RF92].

1.1.1 Knowledge-based Programming

The task of programming can be greatly improved by knowledge-based tools. At a very simple level, the editing of a program can be given more support, for instance, by having shorthands given to larger chunks of constructs and allowing its users to economically refer to these more complex syntactic entities through a simple token. A knowledge-based software assistant can also warn its users about important missing details that should be supplied; it can fill in details that could be inferred from the context; it can report syntactic mistakes at the moment of their typing (not leaving them to be tackled during the compilation, thus preventing the cascading effect of some errors) [Gol86, Wat92].

In a more sophisticated level, knowledge about programming (and not simply about the programming language, as above) can be incorporated in the tool: a library of common programming practices (see Section 1.2) is made available to its users who can adapt these practices to suit their needs. Software development is a knowledge-intensive activity [Gol86, RF92], in which records of previous similar experiences are in constant demand. It is good practice among software engineers to adapt parts of or entire known solutions to new problems. The knowledge of standard solutions and the ability to recall and adapt them to new problems provide the programmer with a working template, progress (in the form of understanding the problem and devising a solution) being achieved in a faster pace than if the programmer had to devise solutions from scratch.

1.2 Representing Programming Knowledge

The programming knowledge to be used in a knowledge-based programming tool can be represented as *programming plans* or as *programming techniques*.

Programming plans, also called *clichés*, are standard algorithmic fragments [Wat92] represented in some formalism: they are formalisations of searching algorithms, sorting algorithms, operations on data structures, and so on. Different proposals have been made to represent programming plans, such as flow graphs [Wil92], plan calculus [RF92], plan diagrams [Wat92], and program transformations [Let88].

Programming techniques are, on the other hand, common code patterns used by programmers in a systematic way, being independent of any particular algorithm or problem domain, but specific to a particular programming language [BBD+91]. Programming techniques have been extensively studied within the logic programming community (*e.g.* [BRV+94, BV93, Bow92, Brn91, BBD+91]), and are more thoroughly explained in the following section.

Bowles and Brna in [BB93] argue that programming plans are inappropriate to be used by novice programmers. They justify this by pointing out the psychological evidence against the idea that programming plans capture the necessary information, and the difficulty novices would have in memorising the large number of existing plans.

Programming plans, depending on the chosen representation, can be rather complex to understand, and alternative more user-friendly formats, such as natural-language explanations [Wat92] or box-diagrams [Wil92] have to be employed.

In this dissertation we adopt the programming techniques approach with respect to Prolog programming. Programming techniques are more intuitive and simpler to grasp, requiring a less elaborate representation than those formalisms proposed for programming plans: the Prolog programming techniques shown in [BB93] and [BRV$^+$94] are all explained using Prolog's syntax itself, plus some auxiliary information. Furthermore, the advantages of declarative programming [Llo94] are, to a large extent, maintained.

1.3 Prolog Programming Techniques

The compactness of Prolog programs allows the easy detection of commonly occurring patterns in the code. These patterns are loosely named *techniques* [BBD$^+$91] and together with knowledge of *when* and *where* to use them, provide a useful account of the body of knowledge necessary for the systematic development of correct Prolog programs. However, 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.

Let there be the following formulation for the *prefix/2* predicate [SS86, O'K90], for instance, which holds if the first argument, a list, is a prefix of the second argument, another list:

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

This predicate is such that in each argument position a technique to manipulate a list is being employed. In the recursive clause the heads of both lists are also bound together as the lists are manipulated. The techniques have, however, different base-case patterns: the empty list in the first position and the catch-all anonymous variable in the second position. As another example, let there be the following formulation of the *reverse/2* predicate [SS86, O'K90] which holds if its arguments are lists such that one is the reverse of the other:

```
reverse(List,Rev):-
      reverse(List,[],Rev).

reverse([],Rev,Rev).
reverse([X|Xs],Rev0,Rev):-
      reverse(Xs,[X|Rev0],Rev).
```

The underlined argument positions comprise an instance of the *accumulator pair* technique, explained in Prolog textbooks [SS86, Ros89, O'K90] as a major efficiency-related Prolog programming technique. Communities using the same programming language eventually build their own folklore of programming practices which are then taught (formally or otherwise) and passed on to future generations: in the Prolog community this phenomenon has been well noticed and documented. Other examples of Prolog programming techniques of public domain are "difference structures" and "failure driven loops" [Ros89, O'K90].

Brna *et. al.* [BBD$^+$91] motivate the study of Prolog programming techniques with theoretical and empirical reasons. Among others, they list the following motivations:

- Programming techniques provide ready-made solutions for subtasks of programming; they provide more abstract pieces of Prolog code that can be used to implement part of a specification. The largely informal and error-prone task of writing a program can be converted into that of finding a technique or a combination of techniques which solves a given problem.
- A set of programming techniques would have uses in Prolog teaching, in that the techniques-acquiring process, only previously achievable by years of programming experience, could be formally carried out and consequently gauged.
- Novice programmers with previous experiences in different programming paradigms could take advantage of a programming techniques compendium which would help them to write programs in Prolog-style.

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 [Rob91b] 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. Programming techniques can be profitably used to support the programming part of problem-solving.

- *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. 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 conjoined yielding more efficient combined versions.

- *Automatic Program Analysis* — techniques have been employed to identify bugs in the Prolog programs of students using a Prolog Intelligent teaching system [Loo88b] and to criticise design decisions [Ben94] in students' intermediate sized code.

Other uses of Prolog programming techniques in intelligent teaching systems, debugging tools, and when learning to program are investigated in [BBD+91].

1.4 A Knowledge-Management Tool and a Techniques-based Software Development Environment

The applications of Prolog programming techniques listed above all assume that 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: 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.

We have implemented a tool to manage a repository (a *library*) of Prolog programming techniques. This tool allows the automatic extraction of programming techniques from actual Prolog programs, offering also a sensible way to organise these components and to reuse them to define new constructs. This management tool also enables its users to define Prolog programming techniques from scratch, via the specialisation of abstract templates. The library of programming techniques managed by our tool can be offered to any of the techniques-based applications listed above, thus defining a software development environment built around a library of programming techniques, as illustrated in Figure 1.1 below. This management tool provides a means to acquire

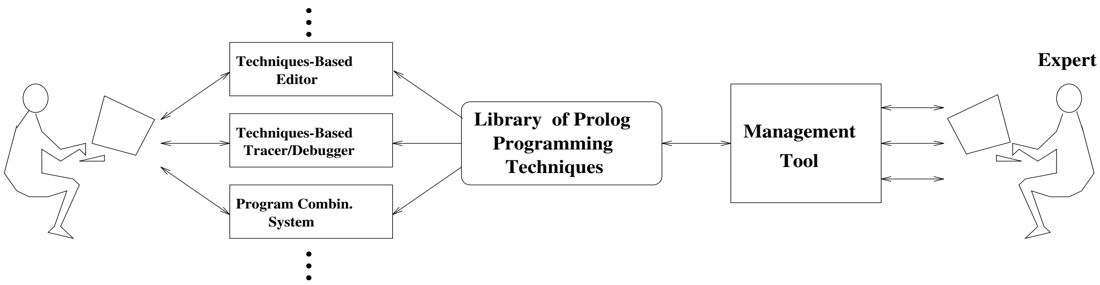


Figure 1.1: A Techniques-Based Software Development Environment

and organise, in the form of a library of Prolog Programming Techniques, the programming knowledge of an expert programmer, making this knowledge accessible to different software development tools. Our tool enables an expert to manipulate the knowledge of a knowledge-based software development environment.

1.5 Prolegomena: Syntax of Programs and Adopted Notation

Only pure Prolog programs, without cuts, disjunctions or if-then-else's, complying with the Edinburgh Prolog syntax [CM87], are our concern here. Moreover, no `assert`, `retract`, `abolish` or similar database-altering predicates can be used. The set of built-in predicates to be employed by the programs are:

- the operators `=`, `=:=`, `=\=`, `==`, `\==`, `>`, `>=`, `<`, `=<` and `=..` (denoted by "$\diamond$");
- the arithmetic operator $is/2$;

- the system predicates *atom*/1, *atomic*/1, *call*/1, *float*/1, *integer*/1, *number*/1, *var*/1 and *ground*/1 (denoted by "¶");
- the input predicates *read*/1 and *get*/1 (denoted by "$\Im$"), and
- the output predicates *write*/1 and *display*/1.

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; for instance, 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`.

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*/1, *display*/1 or *is*/2. In Figure 1.2 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

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

Figure 1.2: Normal Form (left) and Conventional (right) Versions of Procedure *prefix/2*

in body goals are handled in the same fashion, and their descriptions are 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.

1.6 Structure of the Present Thesis

The remainder of the present thesis is divided as follows:

Chapter II: We survey the existing work on Prolog programming knowledge and Prolog programming techniques. We also explain the adopted view of Prolog programming techniques: they are seen as relationships between argument positions of a predicate, with a dynamic aspect which must also be taken into account.

Chapter III: Our adopted view of Prolog programming techniques requires that the execution of the program be somehow represented. In this chapter we propose a means to describe the execution of a procedure and discuss different alternatives to achieve it.

Chapter IV: We describe a method for the semi-automatic extraction of techniques from actual Prolog programs. The outcome of this process is the set of programming techniques employed in the analysed program.

Chapter V: The set of techniques provided by the extraction method of the previous chapter is used by our management tool to create and upgrade the library of programming techniques. In this chapter we describe a suggestion to organise the library of programming techniques by abstracting particular design decisions and creating a hierarchy of more generic constructs.

Chapter VI: We show how the components of the library of programming techniques can be reused to define new techniques: the operators used to abstract the design decisions can be used in a reverse fashion to specialise the generic constructs, giving rise to new components. Abstract templates can also be provided as a starting point to define new techniques.

Chapter VII: This chapter lists some conclusions drawn from the material explained in the previous chapters. We also discuss the contributions made, the applicability of our ideas to other programming languages and paradigms and the directions for future work.

Appendix A: Examples of extracted Prolog programming techniques are given.

Appendix B: δ-Relations used in the abstraction, reimplementation and design of programming techniques are listed, an informal proof for the convergence of the abstraction process is given and examples of Prolog programming techniques being abstracted are shown.

Appendix C: Examples of Prolog programming techniques being designed are shown.

Appendix D: A working example of a complete Prolog program in which its programming techniques are extracted, abstracted and reimplemented.

Appendix E: Glossary with a list of notational shorthands and adopted conventions.

1.7 Summary

In this introductory chapter we have given the context of our work, that is to address the problem of knowledge acquisition and organisation in knowledge-based software development tools. We have also informally described the notion of Prolog programming techniques (which is further refined in Chapter II), explained our proposed knowledge-management tool, introduced the adopted notation and listed the content of each following chapter of this dissertation.

Chapter 2

Prolog Programming Knowledge

2.1 Introduction

In this chapter we survey recent work proposing means to formalise Prolog programming knowledge. This programming knowledge has been put to use in different contexts, such as intelligent tutoring systems, automatic or semi-automatic program development tools and program analysis systems. The different approaches are explained and relevant comments are made. Our adopted view of Prolog programming knowledge, in the form of programming techniques, is also introduced here and compared with the surveyed approaches.

Various proposals have been made towards the formalisation of the Prolog programming knowledge experienced programmers employ when they manipulate programs. The formalised programming knowledge can be embedded in automated tools and put to use to support programming development activities. These approaches differ greatly, but all of them share the common goal of trying to make explicit and/or formalise the programming practices experienced Prolog programmers use when writing reliable programs.

An altogether different sort of Prolog programming knowledge is required for the *transformation* and *combination* of programs. A number of approaches for automatic and semi-automatic program transformation/combination have been proposed (*e.g.* [VF95], [VV95], [VVRI93a], [VVRI93b], [FF92], [LS90], [NN90]). We concentrate our survey, however, on work concerning program development and analysis.

2.2 Prolog Recursive Techniques Editor

Bundy and colleagues [Bun88, BGB91] propose a structure editor which presents the user with a basic schema or a previously generated procedure and a menu of editing commands. The user is to use this and only this repertoire of commands to conduct a series of syntactic transformations upon the primitive schema or a previously generated procedure which leads to the intended procedure.

The basic schema is either the *primitive recursion schema*

$$\begin{array}{rcl} \mu(\flat, \Phi) & \rightleftharpoons & \nu(\Phi) \\ \mu(\sharp(\Psi, \Theta), \Phi) & \rightleftharpoons & \xi(\Psi, \Theta, \Phi, \mu(\Theta, \Phi)) \end{array}$$

where μ is the recursive predicate being defined; ν and ξ are auxiliary predicates in terms of which μ is being defined; Θ is the recursion variable; Φ is the vector of parameters; $\flat$ and $\sharp$ are the constructor functions; Ψ is the constructor parameter; $\rightleftharpoons$ is the equivalence relation, or the *non-recursive schema* which is of form

$$\alpha(\Phi) \rightleftharpoons \beta(\Phi)$$

where α is the predicate being defined, β is some previously defined predicate and Φ is as above. An example of a command of the Recursive Editor is removing a parameter: the user indicates the position of the argument (parameter or constructor parameter) to be removed and all its occurrences are removed.

This approach is based upon theoretical work on recursive functions. Apart from ensuring the syntactic correctness of the programs, the Recursive Techniques Editor also ensures the correct use of recursion. By "correct" it is meant that the recursive procedure is guaranteed to: *i)* terminate: the execution of the procedure takes a finite amount of time; *ii)* be well-defined: the program is neither over- nor under-defined. A program is over-defined when some inputs have more than one output supplied and it is under-defined when some inputs have no output supplied, that is, a legitimate combination of inputs/outputs has not been considered.

The Recursive Techniques Editor was designed to help programmers devise terminating and well-defined programs: these restrictions on the code to be obtained leave out many interesting and useful non-terminating over- or under-defined programs. Even within the class of terminating and well-defined programs, the repertoire of commands is insufficient to produce them all. Some examples of terminating procedures (non-structural recursions, general course of values recursion, nested recursion) which are left out by the Editor are provided in [Bun88].

The commands offered by the Recursive Techniques Editor are syntactic transformations of low granularity [Gol86]. They represent a very simple form of knowledge describing correctness-preserving syntactic manipulations carrying out small, local changes of the initial templates. Although the commands stand for the basic syntactic manipulations programmers have to carry out in their endeavour to obtain correct programs, knowing when and where to apply which command is of paramount importance. This form of "meta-level" knowledge, described in terms of commands, is not in the scope of the Recursive Techniques Editor. However, [Bun88] recognises this and suggests some ways to help the user with which command to choose next: some questions would be asked to help the user think about the task in the proper way and hence call the appropriate commands in the correct order. No formalism is explicitly used in the Recursive Techniques Editor to formalise the programming knowledge employed.

The Recursive Techniques Editor offers a fixed set of commands. It is not possible for an experienced programmer to devise a new command and add it to the Editor or to combine existing commands in a more complex and useful sequence of low-level instructions thus capturing more sophisticated forms of programming knowledge.

2.3 Skeletons and Additions

Kirschenbaum and colleagues [SS86, KLS89, Lak89, SK93] advocate a structured approach to constructing Prolog programs. They formalise the programming knowledge by identifying a set of syntactic entities, called *skeletons*, which describe the flow of control of the most commonly found Prolog programs, and a set of syntactic additions[1] which are applied to the skeletons yielding complete programs. The additions convey the programming practices (*e.g.*: adding parameters, binding them properly, adding goals or clauses) carried out by experienced programmers upon very simple programs

[1] Kirschenbaum and his colleagues use the term "techniques" for these additions. As will be shown, our concept of techniques encompasses their notions of skeletons and additions.

processing data structures (*e.g.*: traversing or manipulating lists) in a straightforward way.

The *modus operandi* of this approach is a stepwise enhancement procedure in which programmers isolate, first of all, the flow of control needed to solve the problem. Then, they choose amongst the skeletons that one which best suits their needs. After having chosen one skeleton, the programmer has to customise it to obtain the final program. This customisation process is achieved via the application of the additions upon the skeleton yielding an *enhancement* (*i.e.*, the original skeleton enhanced with extra arguments, goals and/or clauses). The enhancements can be re-used as skeletons allowing one to repeat the process of application of additions until the final program is obtained. This approach has been incorporated in a software development tool to support the development of Prolog programs, named a *Prolog Techniques Editor* [Rob91c, Rob91b].

Skeletons are basic syntactic constructs depicting the flow of control of a Prolog program. They are programs themselves processing data structures in a very simple and straightforward way. In their work, Kirschenbaum and colleagues suggest the existence of two main trends in simple Prolog programming, the traversal and the manipulation of data structures, and they restrict their work to the former kind of programs. In the following examples $\mathtt{a}_i, \mathtt{b}_i$ and $\mathtt{c}_i$ are case analysis. Although the examples below resemble each other the way the execution ends (base cases) differs:

1. *Skeleton for searching a list*: the flow of control traverses the input list until a certain condition $\mathtt{a}_n$ describing what is actually being searched for is met:

```
search_n([X|Xs]):- a1(X), search_n(Xs).
search_n([X|Xs]):- a2(X), search_n(Xs).
        ⋮
search_n([X|Xs]):- an-1(X), search_n(Xs).
search_n([X|Xs]):- an(X).
```

2. *Skeleton for traversing a list*: the flow of control goes through the input list until it becomes empty; the empty list as the base case of the recursion ensures that the entire list will always be processed:

```
traverse_n([X|Xs]):- b1(X), traverse_n(Xs).
traverse_n([X|Xs]):- b2(X), traverse_n(Xs).
        ⋮
traverse_n([X|Xs]):- bn(X), traverse_n(Xs).
traverse_n([]).
```

3. *Skeleton for short-traversing a list*: the flow of control goes through the input list until an element satisfying c_n is found or the list becomes empty; the empty list as the base case of the recursion ensures that the entire list will be searched for the element satisfying c_n:

```
short_traverse_n([X|Xs]):- c1(X), short_traverse_n(Xs).
short_traverse_n([X|Xs]):- c2(X), short_traverse_n(Xs).
        ⋮
short_traverse_n([X|Xs]):- cn(X).
short_traverse_n([]).
```

Additions are represented as selection maps from a skeleton to one of its extensions. They act upon skeletons, adding arguments to goals or heads, adding goals to clauses or clauses to the skeleton. Skeletons can be thought of as the mechanism controlling the program in terms of the order in which a data structure is processed (the flow of control) and additions determine what is to be done with the data while the control flow is followed. For example, addition *collect* is applicable to skeletons whose clauses are mutually exclusive (at least one of the input arguments will be used for determining which clause to take) and of such a kind that there is a special data item to be collected. This addition inserts an extra argument for each clause (case) in the skeleton, and as the program traverses the data, the argument associated with the clause which is used collects the object in question in a list. The application of *collect* to `traverse_n` is:

$$collect\left(\begin{array}{l}\texttt{traverse\_n([X|Xs]):- b}_1\texttt{(X), traverse\_n(Xs).}\\ \texttt{traverse\_n([X|Xs]):- b}_2\texttt{(X), traverse\_n(Xs).}\\ \qquad\vdots\\ \texttt{traverse\_n([X|Xs]):- b}_n\texttt{(X), traverse\_n(Xs).}\\ \texttt{traverse\_n([]).}\end{array}\right) =$$

$$\begin{array}{l}\texttt{traverse\_n\_coll([X|Xs],[X|Ys}_1\texttt{],Ys}_2\texttt{,...,Ys}_n\texttt{):-}\\ \qquad\qquad\texttt{b}_1\texttt{(X),traverse\_n\_coll(Xs,Ys}_1\texttt{,Ys}_2\texttt{,...,Ys}_n\texttt{).}\\ \texttt{traverse\_n\_coll([X|Xs],Ys}_1\texttt{,[X|Ys}_2\texttt{],...,Ys}_n\texttt{):-}\\ \qquad\qquad\texttt{b}_2\texttt{(X),traverse\_n\_coll(Xs,Ys}_1\texttt{,Ys}_2\texttt{,...,Ys}_n\texttt{).}\\ \qquad\vdots\\ \texttt{traverse\_n\_coll([X|Xs],Ys}_1\texttt{,Ys}_2\texttt{,...,[X|Ys}_n\texttt{]):-}\\ \qquad\qquad\texttt{b}_n\texttt{(X),traverse\_n\_coll(Xs,Ys}_1\texttt{,Ys}_2\texttt{,...,Ys}_n\texttt{).}\\ \texttt{traverse\_n\_coll(}\underbrace{\texttt{[],[],...,[]}}_{n+1}\texttt{).}\end{array}$$

Other examples are the *count* addition which adds one to the value returned from the recursive call (in order to be applied, there must be a numeric argument being returned from a recursive call) and the *sum* addition, similar to the *count* one, however instead of always adding one to the argument being returned, it adds the value of some other argument.

Skeletons may be seen as an "essential" description of a whole class of programs. Conversely, after having collected a number of programs which share the same flow of control, one may write a skeleton which could be enhanced to produce those programs. In this fashion, skeletons could be written representing all sorts of classes of programs. Furthermore, any programming practice that could be described in syntactic terms, *i.e.*, as a syntactic operation carried out upon skeletons, may, in principle, be expressed through an addition. Any program could be written using this approach, provided there is a skeleton which could be used as a basic framework and there are appropriate additions. Skeletons are simple Prolog programs the correctness of which is guaranteed solely by the expertise of the person who provides them. A misconceived skeleton (*e.g.* a non-terminating loop) would cause those programs devised using it to inherit its problems.

Skeletons are represented as actual Prolog programs. In order to be manipulated by an automated tool, a way to store, retrieve and alter them is necessary — such an issue is not envisaged by Kirschenbaum and colleagues; Robertson, in [Rob91c, Rob91b], suggests the use of DCGs for this purpose. Additions, on the other hand, are described informally in English with a sample of an application in a skeleton: this is not precise enough to allow its incorporation to program development systems. Robertson also suggests a means to represent additions.

The approach of Kirschenbaum and his colleagues does not offer a method either to devise skeletons and additions or to compare them; they are obtained through the experience of the programmers who have no method to help them in this task. Skeletons and additions have, however, been used in program development tools [Rob91a, Rob91c], program combination systems [VVRV93, VVRI93a] and debugging tools [Gab92]. This approach has recently been suggested more generally to logic programming languages (sequential or parallel) [KMS94] and to constraint logic programming [Mic94].

2.4 Gegg-Harrison's Prolog Schemata

Gegg-Harrison [GH89, GH91, GH93] proposes the use of *schemata* of Prolog programs in an *Intelligent Tutoring System* (ITS, for short) which introduces the student to recursion via a schema-based instructional approach.

A schema is a generalisation of common Prolog programs employing second-order (predicate) variables, optional, arbitrary and permutable arguments and goals and schema variables (variables which stand for a finite unknown number of arguments — constants, variables or structures) besides the usual Prolog constructs.

The schemata, stored in the ITS, are offered to the user who inputs allowed parameters in order to fill slots and customise the generic schemata to suit his/her needs. Fourteen basic schemata, designed to capture the majority of simple recursive list processing Prolog programs, are the underlying knowledge structure of the proposed ITS. The ITS could also be used as a software development tool with the schemata being used by experienced programmers to help them in the development of more reliable programs. For instance, the list traversal[2] schema below

```
schema_A([],≪&1≫).
schema_A([X|Xs],≪&2≫):-
        <pre_pred(≪&3≫,X,≪&4≫),>
        schema_A(Xs,≪&5≫)
        <,post_pred(≪&6≫,X,≪&7≫)>.
```

may have its schema variables `&1`,...,`&7` instantiated to any number of data arguments. The `pre_pred` and `post_pred` components may be instantiated to either the null string or a single invocation of the goal. The recursion pattern, given by the terms `schema_A`, is such that all elements of a list are processed.

In [GH89] a more formal definition of a schema can be found, in which it is said that "... a Prolog program (written in the proposed extension) is a schema of a set of programs if each of the programs can be derived from it by a series of substitutions on the variables, inclusion/deletion of optional terms, instantiation of arbitrary terms, and permutations of arguments/goals of the schema." There is also an algorithm, the *Most Specific Generalisation Algorithm for Program Schemata*, for finding the most specific general schema for a set of comparable Prolog programs. This algorithm is supplied with a set of programs and a schema which is the most specific generalisation of them will be found.

The schemata can be organised hierarchically based upon relevant common character-

[2] Gegg-Harrison does not name it as such; the above name was a deliberate attempt to show the similarities among the distinct proposals.

istics (*e.g.*: syntax, termination, execution) giving place to *Prolog Schema Hierarchies*. Gegg-Harrison shows 14 basic construct schemata which comprise the core knowledge for the recursive programming segment of the proposed ITS for Prolog. These schemata, believed to represent a vast number of common simple recursive list processing programs, can be organised in different ways according to different features such as termination, data structures employed, and so on.

One remarkable feature of this proposal is that the production of the schemata is automated: a set of similar (*comparable*) conventional Prolog programs are input to the most specific generalisation algorithm and a schema which is the most specific generalisation of the input programs is output. Schemata themselves can be submitted to this algorithm which would then give a generalisation to the schemata. The process of searching for even more generalised schemata naturally defines a *hierarchy of schemata*. This facility saves the hard work of conceiving generalisations: if, by chance, a new, unexpected program is found, all one has to do is to input it to the algorithm mentioned, together with schemata which could possibly generalise it and check whether the output is a different schema. If that is so, then the program gives rise to a new schema.

Because of the resemblance between schemata and the syntax of Prolog, schemata may be devised to represent all "pure" recursive list-processing Prolog programs. However, Gegg-Harrison leaves out other recursive data structures (*e.g.*: binary trees, natural numbers represented as the nesting of a functor `succ`). Prolog programs incorporating these different data structures cannot be used as inputs to the algorithm and hence have no schema. No schemata other than list processing ones are devised.

The only programming knowledge to be found in Gegg-Harrison's work is that of filling in slots of previously obtained schemata and customising them. The task of filling in the slots (slots of schema variables may be filled in by any number of elements) is not given assistance and problems (*e.g.*: accidental binding of variables and inconsistency of number and/or order of parameters) may arise. The specialisation process, though, is left entirely to the user of the ITS who may be an inexperienced novice.

The process of production of schemata, automatised by the Most Specific Generalisation Algorithm, is an example of meta-level reasoning. The actual knowledge of the

ITS, *i.e.*, the schemata, is used as an object of an automated procedure which yields more knowledge. As mentioned previously, this process may also be used to organise the knowledge in hierarchies, which constitutes another meta-level issue. However, no details are given regarding the existence of an environment in which one could devise his/her own schemata and hierarchies, altering the knowledge base used in the ITS.

2.5 Barker-Plummer's Programming Clichés

Barker-Plummer [BP90] describes an approach to construct Prolog programs by instantiating existing generalised programs, called *clichés*. Barker-Plummer proposes an extension to the syntax of Prolog so as to enable the definition of very general predicates, the clichés.

For instance, the following list traversal[3] cliché,

```
$P/n := list_traversal($Q/n).

$P([],&Aux1).
$P([X|Xs],&Aux2):-
      $Q(X,&Aux3),
      $P(Xs,&Aux4).
$end_cliche$.
```

is such that the symbols prefixed with `$` are cliché parameters, with the exception of `$end_cliche$` which is the cliché end-marker. These parameters are to be supplied by the user and must be fully instantiated in order to yield the actual procedure. The first line of the `list_traversal` cliché is the header, bearing the name of the main procedure being defined, `P`, the name of the cliché, `list_traversal`, and the parameters required to define the procedure, `Q`. The next lines are the Horn clauses which define `P`. The head of each clause contains a fixed argument, which is part of the cliché, and an `&Aux` component to be used as a slot for additional arguments. The `&Aux`'s are consistently substituted throughout the cliché and their primary use is to hold Prolog variables or terms, but they may also be empty.

The `list_traversal` cliché can be used to define the class of programs traversing a list and checking each head element against a condition `Q` until the empty list is found. If the user instantiates `P` to `all_integers`, `Q` to `integer` and disposes (*e.g.*, by

[3] The name of the cliché was deliberately changed from `universal` to `list_traversal` to suggest the similarities to other approaches ([KLS89]).

instantiating to blank) of `&Aux`, the following program is obtained:

```
all_integers([]).
all_integers([X|Xs]):-
     integer(X),all_integers(Xs).
```

which would check whether a list is made up of integers only.

Barker-Plummer's idea consists of a restricted second-order language with implicit quantification intended to represent generic programs templates, the *clichés*. The clichés describe generically the flow of control of a procedure acting upon specific data structures and slots are offered to the users so that they may customise the cliché and devise the actual Prolog predicate.

There is a more familiar reading of a cliché using conventional symbols of a second-order formalism. For example, the `list_traversal` cliché, without its additional parameters, could be understood as the following second-order formula [BP90]:

$$\forall\,\mathtt{Q}\,\exists\,\mathtt{P}[\mathtt{P([])} \land \forall\,\mathtt{X},\mathtt{Xs}((\mathtt{P(Xs)} \land \mathtt{Q(X)}) \rightarrow \mathtt{P([X|Xs])})]$$

The cliché specification language admits three kinds of variables: first-order (which will be instantiated to terms), second-order (which will be instantiated to a predicate symbol) and Prolog variables (which will not be instantiated at all, but will be maintained as such). The semantics of a cliché would be the set of all programs which could be devised by instantiating the first- and second-order variables. The concept of a semantics for a cliché is only briefly mentioned in [BP90], though.

In this approach, no guarantee is given regarding the properties of the programs obtained. These rely largely upon the user's skills in correctly instantiating the clichés. It is possible that non-terminating or ill-defined programs can be devised using these clichés.

Due to the similarity to actual Prolog code, the class of programs which can be written with the aid of clichés is, given an appropriate set of clichés, the entire set of Prolog programs. However, a large number of clichés may be necessary. Furthermore, the clichés may be used to represent virtually *any* kind of logic program, however complex they are and with any sort of extra-logical facilities.

The clichés are stored internally in the form of lists. For example, the `list_traversal` cliché is represented as

```
cliche(list_traversal(Q/N),P/N,[Base,(Head:-Perf, Rec)]):-
     n_vars(N,1,Aux),
     Base =..  [P,[]|Aux], Head =..  [P,[X|Xs]|Aux],
     Perf =..  [Q,X|Aux], Rec =..  [P,Xs|Aux].
```

Such a representation would allow a more convenient and efficient handling when being instantiated by the user yielding the desired code. The representation of clichés (user- and computer-oriented) was conceived with concerns such as efficiency and convenience of manipulation as well as to resemble Prolog. As far as the work described in [BP90] is concerned, no knowledge representation formalism is used.

A very basic programming skill, that of having an overall depiction of a class of predicates and having to have it specialised, is captured in this proposal. The allowed specialisations, however, are such that very closely related programs cannot be obtained from the same cliché. For example, a procedure to collect all the integer elements of a list of numbers, could not be devised from cliché `list_traversal`, although bearing remarkable similarities with the procedure `all_integers`.

No cliché editing facilities are supplied and no checks are made within the clichés. The only operations allowed to be performed on a cliché are its specialisation and translation into an actual Prolog program; if, for instance, an extra case `$Q2` was needed in the `list_traversal` cliché, thus providing us with the following construct

```
$P([],&Aux).
$P([X|Xs],&Aux):-
     $Q1(X,&Aux),$P(Xs,&Aux).
$P([X|Xs],&Aux):-
     $Q2(X,&Aux),$P(Xs,&Aux).
$end_cliche$.
```

it would not be possible to alter the previous cliché so as to obtain it. Additionally, the instantiations are entirely up to the user and no advice is given in this task. The clichés are static in the sense that nothing can be done to them after they are input to the system. If a slightly altered version of a cliché is needed, a distinct cliché has to be supplied.

2.6 APROPOS2 Program Analyser

Looi [Loo88b, Loo88a] proposes an approach to the debugging of novices' Prolog programs in a Prolog Intelligent Teaching System (PITS). This proposal is incorporated in a program analyser named APROPOS2: it employs the joint collaboration of multiple sources of programming expertise in order to spot bugs in programs and suggest ways to fix them.

The programming knowledge embedded in APROPOS2 consists of a library of algorithms, information about modes and types, predicate names, recursion types, number of clauses of each predicate, order of these clauses, programming techniques (as in Section 1.3) and the closeness of clause matching. These sources are used to select the algorithm the student had intended to implement and the mismatches are explained as being potential bugs. The programming knowledge is stored in *Prolog frames* (P-frames, for short), a structured representation. The P-frame representation for the naive reverse predicate is:

Task name: reverse/2

Algorithm name: naive-reverse

Likely predicate names: reverse, rev, rv

Invocation type of predicate: both arguments are lists

Programming techniques: naive-recursion

Recursion argument: 2

Clause 1:
TYPE: base
HEAD GOAL: reverse([],[])
PREFIX SUBGOALS: nil
RECURSIVE SUBGOALS: nil
SUFFIX SUBGOALS: nil
COMMENTARY: "This base case says that the reverse of the empty list is the empty list."

Clause 2:
TYPE: recursive
HEAD GOAL: reverse([H|T],Res)
PREFIX SUBGOALS: nil
RECURSIVE SUBGOALS: reverse(T,SoFar)
SUFFIX SUBGOALS: append(SoFar,[H],Res)
COMMENTARY: "This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and then appending a list consisting only of the first element of the original to the end of the reverse of the tail."

additional information on naive-reverse...

The P-frames provide a useful account of the programming knowledge required to implement a specific task. They offer knowledge about the algorithm used, number of clauses, likely names of predicates, and so on — some of these items may be somehow too detailed and of less importance, for instance, the name of a predicate is not essential, so long as there is consistency within its definition. P-frames are an attempt at explicitly representing various forms of programming knowledge involved in the preparation of a Prolog program given a specific task.

Since P-frames are automatically obtained from actual programs, we can use classic solutions to tasks to form the knowledge-base of APROPOS2. There are no references to the upgrading of the knowledge-base of P-frames or to the reuse of existing P-frames to define different constructs.

2.7 Bowles' Prolog Programming Techniques

Bowles [Bow92] proposes a representation for Prolog programming techniques (see Section 1.3) and a method for detecting them. The aim of his work is to enhance APROPOS2's P-frame language, adding to the purely syntactic account of programming techniques a dynamic aspect, *viz.* the effects of unifications which occur during program execution. Bowles' approach employs dataflow analysis, thus providing a more abstract account of programming techniques so as to allow the matching of different actual implementations in students' code. The programming techniques are described in terms of the dynamic effects of unification and the syntax of the constructs.

The description of techniques is based on two relationships between the variables in a clause, named *inclusion* and *sharing*. The inclusion relationship can be found in the subgoal "`X = [H|T]`": we say that `X` includes `H` and `X` includes `T`, and denote this as "$\mathtt{X} \gg \mathtt{H}$" and "$\mathtt{X} \gg \mathtt{T}$". The sharing relationship can be found in the conjunction "`X = [H|T], Y = [H|S]`": we say that `X` shares with `Y`, because `H` appears in both unifications, and we denote this by "$\mathtt{X} \sim \mathtt{Y}$".

Let there be the following formulation[4] of predicate *append*/3:

```
append(A,B,C):-
      A = [],
      B = C.
append(A,B,C):-
      A = [D|E],
      C = [D|F],
      append(E,B,F).
```

In order to detect the techniques employed in a predicate, a series of annotations is

[4] Bowles [Bow92] also assumes that the programs to be analysed are in a normal form, as in Section 1.5.

inserted in each clause. The third argument of the second clause in *append*/3, for instance, has the following annotations (underlined to facilitate their visualisation):

```
append(A,B,C:Up):-
      A = [D|E],
      C = [D|F]:C≫F,
      append(E,B,F).
```

The annotations are attached to variables or subgoals via the ":" symbol. The technique employed in that argument position can be stated as

> A head argument position is *Up* if
> X is the variable in that position
> there is a subgoal which makes X include another variable Y
> there is another subgoal involving Y

Or more economically, using the programming techniques' description language, as

```
head(X:Up):-
      subgoal:X≫Y &
      subgoal(Y).
```

Where "subgoal" is a generic reference to a subgoal, and the "&" stands for the joining of subgoals, without any ordering implied. The generic formalisation for technique *Up* above, using the proposed language to describe programming techniques, correctly identifies instances of its use in which the order of subgoals is not essential. This feature would improve the performance of APROPOS2 since a greater degree of variability is allowed in the programs to be analysed. In order to infer the information necessary to analyse the programming techniques of a given program, abstract interpretation has been used.

This work makes use of the basic unification mechanism of logic programming, restricted to those Prolog programs handling data structures. The Prolog programming techniques Bowles' work is geared at, *i.e.* data structures manipulation, have a clean declarative meaning. Those techniques employing more procedural aspects of Prolog are not addressed: Bowles' work is restricted to Prolog programming techniques manipulating data structures explicitly. Other important techniques such as those found in programs doing arithmetic via the `is` system predicate or handling data structures via auxiliary predicates (*e.g.*, a graph represented as a set of facts depicting its edges) are beyond the scope of his proposal.

The purely syntactic account of Prolog programming knowledge is complemented by information on more dynamic aspects of Prolog programs, the effect of unifications carried out by subgoals on the variables of each clause. This dynamic aspect is incorporated in the formalism to represent programming techniques. The proposed language offers a comfortable level of abstraction with which programming techniques can be represented, thus providing a means to economically refer to a potentially large class of actual constructs.

The language to describe programming techniques and the method of detecting them in actual programs are useful tools to create and manage a knowledge-base of programming techniques. Bowles, however, does not suggest their use for such activity, but restricts it to enhance APROPOS2 P-frame language. No tool or method is proposed for the disciplined use of the programming techniques' language so as to enable expert programmers to devise their own techniques or to modify existing ones.

2.8 Bental's Tasks and Prototypes

Bental [Ben94] proposes an architecture to recognise the design decisions in Prolog programs, with the intention of criticising them. The programs analysed are not simple ones such as *append*/3, *prefix*/2, and so on, but more complex programming exercises such as the programs devised in later stages of a Prolog programming course.

The program analysis is performed with the intention of finding design flaws (bodges) in working Prolog programs of intermediate complexity. Two kinds of programming knowledge are represented in this approach: the knowledge about the specific problem domain, *viz.* a program to play the game of noughts and crosses; and the knowledge about possible Prolog implementations for the subtasks and data structures involved in the solutions to the problem.

The programming knowledge embedded in Bental's system is divided into *programming tasks*, which determine what is to be done (the problem domain), and the *prototypes*, which offer ways to implement the task in the chosen programming language, Prolog. A task definition consists of a task header and an unordered set of sub-tasks. The task header consists of a name plus the number of domain objects manipulated as input or output by the task. Each sub-task description consists of a role description and a body. The role description dictates how the body is matched into a prototype; the body is either a piece of Prolog code or a task specification which could match the header of another task. For instance, the following illustration[5]

find_empty_square /2

Sub-tasks	
Role	Body
base(Base)	`Base = 1`
increment(This,Next)	`Next is This + 1`
test(Square)	*empty_square*`(Square)`

Game objects: Board, Square

represents a simple task to find the position of an empty square in a list of squares: it consists of three sub-tasks, *base*, *increment* and *test*, which initialise a counter, increment that counter and perform a test, respectively. The first two of these sub-tasks map onto Prolog code; the third one maps onto another task definition to test if a square is empty. The system is supplied with a definition for each task it is expected to recognise.

A prototype represents how a task can be realised in terms of the programming language. Each prototype consists of one or more prototypical Prolog predicates, which in their turn consist of one or more prototypical Prolog clauses. Parts of the prototype stand for Prolog code used for all the tasks implemented using that prototype. Other parts of the prototype are slots referring to different sub-tasks for each task: the contents of a slot are specific to the task to be implemented. For instance, the following illustration[6]

TESTED_ELEMENT_POSITION_ACC

```
elem_pos(List,Res):-
      base(Base),
      elem_pos(List,Base,Res).

elem_pos([H|_],Base,Base):-
      test(H).
elem_pos([_|T],Acc,Res):-
      increment(Acc,Next),
      elem_pos(T,Next,Res).
```

represents a prototype to find the position of an element of a list which satisfies some test. The role slots are *base*, *test*, and *increment* and they can match to the role descriptions in the task definition *find_empty_square* shown above. Each prototypical predicate consists of one or more prototypical clauses, which are Prolog clauses with

[5] Bental [Ben94] graphically represents tasks in a slightly different way.

[6] Bental [Ben94] graphically represents prototypes also in a slightly different way.

special role slots (a special form of predicate) to be specified according to the task it implements. The objective of a prototype is to encapsulate a Prolog programming practice in an abstract way, allowing its specialisation to a given task.

Bental's system incorporates a clear-cut division between domain- and language-specific knowledge. The former provides an abstract account of the tasks involved in implementing a program to play a game of noughts and crosses. The latter provides a form of programming knowledge: the prototypes are standard programming practices which can be used for different purposes (tasks), once they are properly adapted.

The programming knowledge in the form of prototypes bears a strong resemblance with Gegg-Harrison's schemata (Section 2.4). It provides a complete predicate with slots to be filled in: the filling of these slots, is, however, by means of the tasks, which provide knowledge about the use of these programming practices to achieve specific ends.

2.9 The View of Prolog Programming Knowledge Adopted in this Thesis

We shall incorporate Prolog programming techniques, informally described in Section 1.3, as the programming knowledge of our knowledge-based software development environment (Section 1.4). In this section we provide more details about our adopted view of Prolog programming techniques.

Prolog programming skills consist, to a large extent, in the capacity to exploit the relatively simple syntax of Prolog in elaborate manners. Prolog program writing can be profitably seen as the proper assemblage of standard code patterns, the programming techniques [BRV+94, BBD+91]. A Prolog program can also be explained and studied in terms of the code patterns it employs.

Prolog programming techniques will be considered as being associated with the argument positions of a procedure[7]: each argument position is either a programming

[7] 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.

technique on its own or is part of a more complex technique together with other argument position(s). For instance, the following definition of the *count*/2 predicate [SS86, O'K90]

```
count([],0).
count([H|T],Count):-
      count(T,CountT),
      Count is 1 + CountT.
```

which holds if its second argument is the number of elements of the list comprising the first argument, is such that a technique to manipulate a list has been used in the first argument, that is

```
count([],...).
count([H|T],...):-
      count(T,...).
```

and another technique to count the iterations of a loop has been employed in the second argument position, that is

```
count(...,0).
count(...,Count):-
      count(...,CountT),
      Count is 1 + CountT.
```

Programming techniques may involve more than one argument position, as in the second and third argument positions of predicate *reverse*/3 (Section 1.3), that is,

```
reverse(...,Rev,Rev).
reverse(...,Rev0,Rev):-
      reverse(...,[X|Rev0],Rev).
```

The `X` variable in the second clause, being part of the list being manipulated in the first argument position (not shown), creates another dependency between the two argument positions pictured above and the first argument position.

Any Prolog procedure can be explained as the proper combination of programming techniques: each argument position contains a simple programming technique or is part of a more complex programming technique involving other argument positions. Alternatively, any Prolog procedure could be devised via the disciplined use and adaptation of programming techniques. One of our assumptions is that each argument position of a Prolog procedure should serve some identifiable purpose, otherwise it should not be present; each argument position contributes to the overall behaviour or computed results of the procedure: if this were not true, then that idle argument position should not have been present at all. The contribution of each argument position to a procedure is in the form of subgoals performing tests, calculations, assignments, and so on.

The contributions of an argument position in each one of the clauses of a procedure define the global contribution of an argument position for the procedure. This is to say, a Prolog programming technique "cuts across" a predicate definition, along each argument position, with components in every clause. In the *count*/2 example above we have, for each argument position, contributions in the first and second clauses, and they are closely related. This characteristic allows for a more controlled use of programming techniques to build programs because their application would appropriately perform the alterations (*i.e.* insertion of contributions) in each clause. For instance, if we were to enhance the following formulation of procedure *sum*/2 [SS86, O'K90]

```
sum([],0).
sum([H|T],Sum):-
      sum(T,SumT),
      Sum is H + SumT.
```

which holds if its second argument is the sum of the elements of the list of numbers comprising the first argument, with the programming technique used in the second argument position of *count*/2, thus obtaining the number of elements of the first argument in a third argument position, we would have

```
sum([],0,0).
sum([H|T],Sum,Count):-
      sum(T,SumT,CountT),
      Sum is H + SumT,
      Count is 1 + CountT.
```

The alterations to each clause were performed appropriately since they are all part of the programming technique. The contributions of an argument position to the clauses of a procedure define an *argument slice*. We restrict our attention to programming techniques within a single procedure. Techniques spread across more than one predicate (*e.g.*, initialisation calls and mutually recursive predicates) are outside the scope of this work. Moreover, those techniques used in procedures without arguments in their head goals do not fit into our proposal. For instance, given the procedure *calculator*/0 below

```
calculator:-
      read(X),
      X \== end,
      Y is X,
      write(Y),
      calculator.
calculator.
```

which reads in arithmetic expressions and evaluates and displays them, stopping when an `end` flag is typed in, its technique(s) cannot be captured by our proposed formalisation. This is due to our premise, stated in the beginning of this section, that programming techniques are associated with argument positions of a procedure.

2.10 Comparison between Approaches

In this section we compare our adopted view of Prolog programming techniques with the other approaches surveyed. More specifically we contrast the adequacy of our approach to the task at hand, *viz.* the management of the programming knowledge of the envisaged knowledge-based software development tool described in Section 1.4, with the existing proposals.

We propose an automated tool to help manage a knowledge base of Prolog programming practices. This tool supports the manipulation of the knowledge of a knowledge-based software development environment. The kinds of service we offer the users of such a tool are:

1. automatic acquisition of knowledge;
2. support in the preparation of new components to be added on to the knowledge base;
3. support in the reuse of existing components of the knowledge base to define new ones;
4. organisation of the knowledge in different manners;

The suitability of a formalisation of programming knowledge must account for this intended use.

The programming knowledge captured in the approach of Bundy and colleagues (Section 2.2) is in the form of commands offered to alter very generic program schemata. We pointed out above that this is a very finely-grained form of programming knowledge. The adopted concept of programming techniques can be seen as a sequence of commands of the Prolog Recursive Techniques Editor with a collective meaning. For instance, the appropriate insertion of an argument and subgoals in each clause, *i.e.* a programming technique, is performed as a single operation, instead of a sequence of lower-level syntactic alterations to the initial template.

Our notion of programming technique encompasses the skeletons and additions of Kirschenbaum and colleagues (Section 2.3). The distinction between skeletons and additions is based on their usage in the construction of a program: a skeleton is chosen first as the definition for the flow of execution, and it is subsequently enhanced with additions. We consider both skeletons and additions as techniques, being able to characterise the conditions a technique must fulfil to be considered a skeleton or an addition.

Gegg-Harrison's Prolog Schemata (Section 2.4) concentrate on capturing list processing patterns. These patterns are represented as the first argument(s) of a Prolog predicate definition with slots to be filled in with variables or subgoals. This approach fails to represent the programming practices employed to fill in these slots which are as important as the patterns to establish the flow of control. The schemata also impose an unnecessary ordering in arguments in that the first argument position(s) always perform the list processing and the added bits perform secondary computations as the execution proceeds. Our adopted view of programming technique can be used to circumvent these problems: some programming techniques could be regarded as establishing the flow of execution (and these can be the processing of any data structure, not only lists) and other programming techniques could be regarded as the practices to fill in the slots of a schema. Barker-Plummer's Programming Clichés (Section 2.5) are much similar to Gegg-Harrison's Schemata and inherit the same problems.

Looi's (Section 2.6) and Bental's (Section 2.8) proposals pose difficulties for services 1, 2 and 3 expected from a programming knowledge management tool. Since both these approaches employ some form of knowledge about the programs to be built (or the intention of the programmer) it would be difficult to have this knowledge automatically extracted from programs, and then offered as an object to be edited. Bental's proposal also inherits the problems of Gegg-Harrison's schemata listed above.

Our work is closely related to that of Bowles (Section 2.7) in that we both use similar notions of programming techniques. Both our work and Bowles' employ extra information on the dynamics of Prolog behaviour to augment the syntax. Bowles' work, however, is geared towards detecting explicit data structure manipulating techniques in programs, and it exploits the working of Prolog's unification mechanism to analyse them. Our approach aims at a more comprehensive class of programming techniques, (*e.g.* data structures other than lists, arithmetic, and so on) and employs a more sophisticated representation of the execution of a Prolog program. Bowles' approach fails to detect those programming techniques which embed part of them in auxiliary procedures. The following example of one of such techniques, taken from [Bow92]

```
closure(A,A).
closure(A,B):- arc(A,C), closure(C,B).

arc(a,b).
arc(a,c).
   ⋮
```

illustrates a programming technique whose manipulated data is in the form of facts in the knowledge base: since no relationship between the variables of *closure*/2 can be drawn based on unification, this technique is not detected. Our approach models more procedural aspects of the execution of Prolog programs, being able to detect the technique above. In that sense, our work can be seen as an adaptation of Bowles' ideas to cope with a larger class of programming techniques, adopting a more procedural view of Prolog.

2.11 Detecting Prolog Programming Techniques

Being able to spot and extract programming techniques in working Prolog programs allows the automatic creation and upgrading of our programming knowledge base. Programming techniques extracted from real programs and stored in our knowledge base can be employed by all those knowledge-based tools of our environment (Section 1.4) and also be reused to define other programming techniques.

Our notion of programming techniques has to be sufficiently well-defined to be automatically detectable. We have devised a formalisation for programming techniques which allows their automatic recognition and extraction. This formalisation is explained in detail in Chapter 4; we shall briefly outline its intuition in the rest of this subsection.

The detection of programming techniques amounts to finding out which subgoals are relevant to which argument positions: given an argument position in a procedure, the subgoals of each clause which are related to it will comprise the programming technique (or part of the programming technique, if it involves more than one argument position) in that argument position. We provide a formal definition for this notion of relevance in Chapter 4.

For instance, if we were to detect the programming techniques of *count*/2 procedure shown above, it would be necessary to conclude that the subgoal "`Count is 1 + CountT`" is not related to the first argument position, but it is relevant to the second argument position. The syntax of a subgoal, that is, its predicate symbol and variables, sometimes is enough to tell us which argument position it is relevant to. However, the decision concerning the relevance of a subgoal with respect to an argument position is not always as trivial as in the *count*/2 procedure.

For example, let there be the following formulation of *sum_int*/2 predicate

```
sum_int([],0).
sum_int([H|T],Sum):-
      integer(H),
      sum_int(T,SumT),
      Sum is H + SumT.
sum_int([_|T],Sum):-
      sum_int(T,Sum).
```

which is similar to *sum*/2 above, but a test is carried out to make sure only the integers in the list comprising the first argument are actually summed up in the second argument (predicate `integer` in the second clause). The "`integer(H)`" subgoal is obviously related to the first argument position, but it is not all that clear if "`Sum is H + SumT`" is also related to it: both subgoals employ variable `H`, but in different contexts. If we want to distribute the subgoals of each clause in a procedure amongst the argument positions, we need to be able to detect the different contexts of the occurrence of variables.

The context of the occurrence of a variable may be given by the predicate employed in the subgoal. In the *sum_int*/2 example above, the "`integer(H)`" subgoal is a test employing the system predicate `integer`: this is an important subgoal for the programming technique of the first argument position because it defines a case for the manipulation of the list, thus providing a distinction between the two recursive clauses of the procedure. If we choose to see the procedure *sum_int*/2 from a procedural perspective, then the "`integer(H)`" is important for the list manipulation technique since it has the capacity to change the flow of execution.

However, it is not straightforward whether or not the "`Sum is H + SumT`" is relevant to the first argument position. This is due to the inability to determine the context of those variables appearing in subgoals employing the "`is`" system predicate: it may be used as a means to perform an arithmetic calculation on its right-hand side followed by the assignment of the result of this calculation to its left-hand side, or to perform a calculation on its right-hand side followed by a test of this result against the value associated with the variable in left-hand side. The *sum_int*/2 procedure can be used for calculating the sum of the integer elements of a list or to test if that sum is equal to a given figure. If "`Sum is H + SumT`" is used to assign a value to `Sum` then it is simply employing the value of `H` supplied by the list manipulation technique and hence is not relevant to the first argument position. If, on the other hand, it is used to perform a test, then it is as relevant to the first argument as the "`integer(H)`" subgoal.

This notion of "context" of occurrences of variables is fundamental to our definition of programming techniques. We need to know how each variable has been used in a subgoal so as to be able to decide if that subgoal is important or not to an argument position. For some subgoals this can be inferred just from the syntax of their predicates. Other subgoals are such that their meaning is not clear from the syntax only and auxiliary information must then be employed. Our notion of Prolog programming technique thus entails a *dynamic* aspect: the syntax of the procedure must be complemented with the manner in which it has been used. This view differs from the purely syntactic approaches of Kirschenbaum and colleagues (Section 2.3), Gegg-Harrison (Section 2.4), Barker-Plummer (Section 2.5, Looi (Section 2.6) and Bental (Section 2.8), and is related to that of Bowles' (Section 2.7).

We aim at detecting programming techniques in pure Prolog programs (Section 1.5) whose procedures may be used in different ways or directions [SS86]. We consider a subgoal relevant to an argument position if *i)* it provides values to that argument position; or *ii)* it performs a test which may interfere with that argument position.

We collect additional information on the usage of a subgoal by analysing the execution of the procedure. We make use of the sequential execution model of pure Prolog programs [SS86, Llo93], using SLD-refutation with a leftmost atom selection rule and depth-first search rule. The programming techniques of a procedure are always analysed *with respect to* a particular usage, as given by a query or set of queries. The same syntax may yield different programming techniques, depending on its usage. We adopt the view of a Prolog programming technique as consisting of the syntax of Prolog code and the way this is used in answering queries.

We have adopted a procedural view of Prolog programs when trying to detect their programming techniques because some constructs in Prolog programs have a strong procedural reading (*e.g.* arithmetic via the `is` predicate) and can most easily be understood if the programs they appear in are viewed procedurally. Programming techniques with a declarative reading (*e.g.* difference-lists, manipulation of recursive data structures, and so on) can also be understood procedurally whereas some procedural-oriented techniques do not have a declarative description.

We propose a means of economically representing the execution of a procedure using *mode-annotations*. A mode-annotation contains information on the instantiation of the variables of each clause. Each subgoal has a pair of mode-annotations, one *before* its execution and another *after* it. We associate with each variable a *token* which represents its *instantiation status*: for example, if a variable is free at one point during the execution, we associate token "**f**" with it; if it is ground we associate token "**g**". For instance, if subgoal "`Sum is H + SumT`" had token "**g**" associated to `H` and `SumT`, and "**f**" associated with `Sum` before its execution and "**g**" associated to all its variables after its execution, we could correctly infer that that subgoal had been employed to perform a calculation/assignment. Knowing which computation each subgoal is performing enables us to decide about its relevance to the argument positions. Further details on the mode-annotations, other alternatives to represent the execution of a procedure and the different ways to obtain the mode-annotations are explained in Chapter 3.

Our view of programming techniques takes into account the sharing of variables between the contributions of argument positions. We employ place holders in strategic points stating which variables are required or offered in the contributions of an argument position. For instance, the second argument position of procedure *sum*/2, has the following formulation for its contributions

```
                      sum(...,0).
                      sum(...,Sum):-
                          sum(...,SumT),
⟨⟨required({H})⟩⟩           Sum is H + SumT.
```

The *required* annotation states that at that point a variable `H` must be supplied by the contributions of another argument position. The contributions of the first argument position, if *sum*/2 is used to obtain the sum of the elements[8], are

```
sum([],...).
sum([H|T],...):-    ⟨⟨offer({H,T})⟩⟩
    sum(T,...).
```

Its *offer* annotation states that from that point onwards two variables `H` and `T` are at the disposal of the contributions of other argument positions. The use of these auxiliary annotations permits the convenient record of the sharing of variables among the contributions of different argument positions. It represents another aspect of programming knowledge, that of being able to properly relate the contributions of distinct argument slices.

2.11.1 Program Slicing

Our proposed method for detecting programming techniques is related to work carried out on *program slicing* within the procedural paradigm community [Wei82, Wei84, RW89, GL91, JR94, RHSR94]. Program slicing is a technique aimed at procedural languages to isolate portions of the code of a program that have some common functionality: it is a decomposition process based on data flow and control flow, restricting a program to a subsequence of its commands which are somehow related.

Weiser [Wei82, Wei84] proposed a method to automatically decompose a given program thus obtaining an executable subprogram, a slice, which reproduces a restricted form of the behaviour of the original program. The purpose of his work is to restrict a program to those commands relevant to particular variables and points of the code, breaking apart large programs into smaller coherent pieces.

To illustrate the program slicing method, we shall consider the C program shown in Figure 2.1, taken from [KR88] and reproduced in [GL91], a simplified version of the

[8] As opposed to testing if the sum is equal to a given value, in which case different contributions would be obtained.

Unix utility `wc`, to count the words, characters and lines of a given file. In order to

```
1    #define YES 1
2    #define NO 0
3    main()
4    {
5         int c, nl, nw, nc, inword;
6         inword = NO;
7         nl = 0;
8         nw = 0;
9         nc = 0;
10        c = getchar();
11        while (c != EOF) {
12             ++nc;
13             if (c == '\n')
14                  ++nl;
15             if (c == ' ' || c == '\n' || c == '\t')
16                  inword = NO;
17             else if (inword == NO) {
18                  inword = YES;
19                  ++nw;
20             }
21             c = getchar();
22        }
23        printf("%d '\n'",nl);
24        printf("%d '\n'",nw);
25        printf("%d '\n'",nc);
26   }
```

Figure 2.1: C Program to be Sliced

obtain the slices of a program, the proposed method requires that a *criterion* has to be provided. A criterion is a pair $\langle i, V \rangle$ where i is a line number of the program and V is a subset of the program variables. A program slice with respect to a criterion consists of all those commands up to line i which are relevant to the set of variables V. The criterion can be simplified, as done in [GL91], such that the line i is always considered to be the last line of the program to be sliced, and the criterion would only consist of the set V; in this case we have a *decomposition slice*. For instance, the following program slice obtained with respect to criterion $\langle 26, \{\texttt{nw}\} \rangle$ outputs the number of words in a file [GL91]; it contains those commands relevant to the word counter variable `nw`:

```
1    #define YES 1
2    #define NO 0
3    main()
4    {
5         int c, nw, inword;
6         inword = NO;
8         nw = 0;
10        c = getchar();
11        while (c != EOF) {
15             if (c == ' ' || c == '\n' || c == '\t')
16                  inword = NO;
17             else if (inword == NO) {
18                  inword = YES;
19                  ++nw;
20             }
21             c = getchar();
22        }
24        printf("%d '\n'",nw);
26   }
```

The following executable program comprises the program slice obtained with respect to criterion $\langle 26, \{\texttt{nc}\}\rangle$ [GL91], and is such that it counts the number of characters in the input text file; it contains those commands relevant to the character counter variable `nc`:

```
3    main()
4    {
5         int c, nl, nw, nc, inword;
9         nc = 0;
10        c = getchar();
11        while (c != EOF) {
12             ++nc;
21             c = getchar();
22        }
25        printf("%d '\n'",nc);
26   }
```

Psychological evidence for program slicing is reported in [Wei82]. In his work it is argued that when debugging a program, programmers mentally slice the program, starting from the command where the error is first reported and moving through the flow of control of the program backwards towards its initial command, checking for the relevance of each command until the place where the bug is introduced is found. It is thus argued that tools to support debugging could profit by embedding a program slicing method.

The use of program slicing in software maintenance is pursued in the work of Gallagher and Lyle [GL91]. A refinement of Weiser's definition of program slice, the decomposition slice, is given, which accounts for full programs. A discipline of software maintenance is developed employing the introduced concept of decomposition slice. Reps and Yang in [RW89] provide a semantics for program slicing.

We make a comparison between Weiser's work and our proposal in Section 4.6, after the presentation of our argument slicing method. A more detailed account of the slicing of procedural code is given there. We also discuss the positive features and drawbacks of each proposal.

2.12 Summary

In this chapter we have:

- surveyed existing proposals to represent Prolog programming practices;
- described informally our adopted view of Prolog programming knowledge, the programming techniques;
- compared the adequacy of our view of Prolog programming techniques against the surveyed proposals, for the task at hand, *i.e.* to provide an expressive formalisation of programming practices convenient for embedding in a knowledge managing tool.
- shown how programming techniques can be detected in programs and how this relates to work carried out within the procedural paradigm community.

Chapter 3

Representing the Execution of Prolog Programs

3.1 Introduction

We want to detect a large class of programming techniques, including those with a strong procedural "flavour", such as those performing arithmetic via the *is*/2 system predicate, or manipulating data structures via auxiliary procedures. In order to achieve this, we must be able to represent the procedural meaning of a program.

In this chapter we suggest that the procedural meaning of a program can be given by its execution. Different queries may yield different executions, and hence different meanings. This is in accordance with our view of programming techniques: the same procedure may contain different techniques depending on its usage. In the sections below we propose different ways to represent the execution of a program and describe manners to obtain them.

Ideally, we want a description of the usage of each clause of a procedure. The order in which the clauses are used and the number of times each clause is used is not relevant to our formalisation of programming technique. We want to supplement the syntax of the procedure with the manner it is used during conventional Prolog interpretation. The approaches shown in this chapter aim at this ideal scenario.

We propose an economic manner to depict the execution of a program using *mode-annotations*, a simplified form of substitution describing the instantiation status of each variable in the clauses of a procedure and how they change as the execution proceeds. We describe different ways of obtaining the mode-annotated version of a procedure and list their advantages and drawbacks. We claim that this proposal provides us with the information we need to detect programming techniques.

3.2 Execution of Prolog Programs

By *execution* of a Prolog program P with respect to a query Q we mean the construction of a SLDNF-refutation for $P \cup \{\neg Q\}$, that is, a proof that $P \cup \{\neg Q\}$ leads to a contradiction [Llo93] using SLDNF-resolution and Prolog's computation rule (*i.e.* leftmost goal) and search rule (*i.e.* depth-first). Only the *first* solution of a query is of importance for us: the different solutions obtained by backtracking will not be considered in our analysis.

Existing representations for the execution of programs, such as proof-trees [Llo93, SS86] and traces [SS86, CM87] concentrate on its dynamic aspects, showing the progress of the computation in terms of the unifications taking place.

We propose in the next section a representation for the execution of a program which relates its syntax to the dynamics of its computations. Then in the following sections we improve this first proposal, thus obtaining a more economic representation of the execution of a program.

The programming techniques of our concern are those restricted to a single procedure. Although the proposals to represent executions aim at complete programs, our focus will be a single procedure at each time, specified by the user of our knowledge management tool. We assume that the focused procedure is always the topmost predicate, *i.e.* that predicate which employs (calls) other subsidiary procedures.

3.3 Recording the Instantiation of Variables

Important aspects in the execution of a Prolog program can be described in terms of the instantiations performed on the variables in its clauses. Proof-trees and traces show these instantiations in the order they happen, and may also include information about the order of clauses employed, backtracking, and so on.

We propose here a means to incorporate the dynamics of the execution of a program to its syntax. This proposal consists of listing the clauses employed in the execution, in the order they were used: the clauses are shown as a one-piece construct, without any disruptions to show the execution of auxiliary user-defined predicates or recursive calls, as traces [SS86, CM87] conventionally do. This proposed representation could be seen as a "flattened" trace.

The instantiations taking place during the execution are incorporated in our representation as *annotations* strategically placed in each clause. These annotations represent the substitutions (*i.e.* the instantiation of variables) holding at that place in the clause where they appear. Each annotation contains all the variables of the clause and are placed after the head goal and after each subgoal in the body of the clause: the result of the execution of each subgoal is conveniently represented as the changes that took place in the instantiations of its variables. For instance, let there be the definition below for predicate *collect*/2 — we shall follow Bowles' [Bow92] style and present both the normal form (left) and conventional versions (right) of a procedure the first time we show it:

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

It 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. The execution triggered by the query

```
?- collect([1,foo,3],Int).
```

is shown in Figure 3.1 in our proposed representation. The curly brackets enclose the set of substitutions that hold at that point in the clause. Each element of a substitution is of the form x/t where x is a Prolog variable and t is either a Prolog term associated

```
collect(A,B):-        {A/[1,foo,3],B/↓1,X/↓2,Xs/↓3,Ys/↓4}
   A = [X|Xs],        {A/[1,foo,3],B/↓1,X/1,Xs/[foo,3],Ys/↓4}
   B = [X|Ys],        {A/[1,foo,3],B/[1|↓4],X/1,Xs/[foo,3],Ys/↓4}
   integer(X),        {A/[1,foo,3],B/[1|↓4],X/1,Xs/[foo,3],Ys/↓4}
   collect(Xs,Ys).    {A/[1,foo,3],B/[1,3],X/1,Xs/[foo,3],Ys/[3]}
collect(A,B):-        {A/[foo,3],B/↓4,X/↓5,Xs/↓6}
   A = [X|Xs],        {A/[foo,3],B/↓4,X/foo,Xs/[3]}
   collect(Xs,B).     {A/[foo,3],B/[3],X/foo,Xs/[3]}
collect(A,B):-        {A/[3],B/↓4,X/↓7,Xs/↓8,Ys/↓9}
   A = [X|Xs],        {A/[3],B/↓4,X/3,Xs/[],Ys/↓9}
   B = [X|Ys],        {A/[3],B/[3|↓9],X/3,Xs/[],Ys/↓9}
   integer(X),        {A/[3],B/[3|↓9],X/3,Xs/[],Ys/↓9}
   collect(Xs,Ys).    {A/[3],B/[3],X/3,Xs/[],Ys/[]}
collect(A,B):-        {A/[],B/↓9}
   A = [],            {A/[],B/↓9}
   B = [].            {A/[],B/[]}
```

Figure 3.1: Clauses of Procedure *collect*/2 Annotated with the Instantiation of Variables

with x, or a reference to a memory cell [AK90] or a term with references to memory cells. A reference to a memory cell, denoted by $\downarrow_n$, is a pointer to a specific memory position: if a variable is associated with a memory cell then it has no value assigned to it. The substitutions are placed after the head goal is matched and after each subgoal execution.

The depiction of the execution above has the second clause of *collect*/2 as its first clause. After the matching of the query with its head goal, we have `A` instantiated to the list `[1,foo,3]` and all the other variables of the clause associated with empty memory cells, *i.e.* they are *free*. Although it is not shown, the `Int` variable in the query is associated with $\downarrow_1$, hence the value associated with `B`. After the execution of its first subgoal, variables `X` and `Xs` change their instantiation, becoming associated with `1` and `[foo,3]` respectively, and all the other variables in the clause remain as they were. After the second subgoal is executed, `B` becomes associated with the term `[1|`$\downarrow_4$`]`: the occurrence of `Ys` is replaced by the memory cell it points to. The execution of "`integer(X)`" does not change the instantiation of any variable, since it performs a simple test. The rest of the execution is triggered by the recursive call at the end, "`collect([foo,3],`$\downarrow_4$`)`", whose execution assigns the term `[3]` to the memory position pointed at by $\downarrow_4$.

The rest of the annotated clauses are obtained in a similar way: the execution of subgoals performs changes on their variables and the recursive calls trigger other executions until the empty list is reached and the base case clause is executed. The annotated clauses are shown in one piece, without disruptions to show the execution of user-defined subgoals or recursive calls, as a trace would do. Failed clauses are discarded: the subgoals in their bodies that succeeded and were successfully annotated are not relevant to our characterisation of the programming techniques of that procedure. Only clauses that succeed are actually collected.

Given a program and a query, the representation for its execution as shown above can be obtained by the enhanced meta-interpreter [SS86, SL88] shown in Figure 3.2 below. In *prolog1*/4 the substitutions of a procedure are kept explicitly in the `Th` variables. The

```
prolog1((G,Gs),Th0,Th,(AnG,AnGs)):-
   prolog1(G,Th0,Th1,AnG),
   prolog1(Gs,Th1,Th,AnGs).
prolog1(\+G,Th,Th,(\+G,Th)):-
   \+ prolog1(G,Th,_).
prolog1(G,Th0,Th,(G,Th)):-
   system(G),
   call1(G,Th0,Th).
prolog1(G,Th0,Th,(G,Th)):-
   clause1(G,Th0,(Hd:-Bd),Th1),
   prolog1(Bd,Th1,Th2,AnnBd),
   update(Th0,Th2,Th),
   retract(execution(AnCls)),
   assert(execution([(Hd:-Th1,AnnBd)|AnCls])).
```

Figure 3.2: Enhanced Meta-Interpreter to Record the Instantiation of Variables

first argument of *prolog1*/4 holds the subgoals to be executed; the second argument holds the instantiation of the variables before the subgoal (or subgoals) is (or are) executed; the third argument holds the instantiation of variables after the subgoal (or subgoals) is (or are) executed; and the fourth argument holds the subgoal(s) with its (theirs) annotation(s). The initial call must have its substitutions explicit; to obtain the mode-annotated clauses shown in Figure 3.1, it should have been

```
?- prolog1(collect(A,B),{A/[1,foo,3],B/_},Ans,_).
```

where `B`'s association with an anonymous variable in the initial substitution is due to its being initially free. `Ans` will provide us with an answer to the query, in the form of a substitution.

The *call1*/3 predicate is an adaptation to the *call*/1 system predicate to cope with built-ins: it basically applies a substitution `Th0` to a subgoal `G`, executes it, and obtains its resulting substitution `Th`. The *clause1*/3 is also an adaptation of a system predicate; it adapts the *clause*/2 system predicate to cope with explicit substitutions: it applies `Th0` to `G` and searches in the knowledge base for a clause whose head goal matches `G/Th0`. The *update*/3 predicate updates substitution `Th0` to reflect the final instantiations of the variables of the newly executed clause recorded in `Th2`. In order to keep the enhancement simple, we opted for a global structure (*execution*/1, being updated in the last clause) to store the annotated clauses; at the beginning of our analysis, this global structure is set to the empty list.

One problem this representation for the execution of a program faces is that, depending on the program and the query supplied, there may be many different copies of the same clause, their only difference being the particular values to which their variables were instantiated. In our *collect*/2 example above, the second clause of the procedure was used twice, each time with different values, hence its double occurrence. If the list input in the query had n integer elements the second clause would have appeared replicated n times, with different instantiations. According to our proposal, the number of times a clause has been used is not a relevant piece of information for the detection of programming techniques.

The source of this problem is the recording of particular values used during the execution. Instead of having this detailed and rather verbose account of an execution, we can choose a level of abstraction in which the particular values are overlooked and copies of clauses can be collapsed together. Particular values associated with variables are abstracted as tokens depicting their instantiation status, *i.e.* if the variable is free, ground or instantiated. This solution is explained in the following section.

Another problem arises when a bad choice of a query causes important clauses in the procedure to be left out. For example, if we had executed our procedure *collect*/2 with respect to query "`?- collect([foo1,foo2,foo3],Int)`" the second clause of the procedure would not have been used at all, since no elements in the list would satisfy the *integer*/1 predicate.

3.4 Recording the Instantiation Mode of Variables

A more abstract and economic way to represent the execution of a Prolog program can be achieved by a refinement of the previous proposal. Instead of annotating the used clauses with the actual values associated with its variables, we can employ their instantiation status, *i.e.* if the variables are free, ground, and so on. The copies of a clause which differ only by the actual values of its variables would become equal and a single copy of it could be used instead, thus solving the problem of having multiple copies of a same clause.

According to this approach, instead of having actual values associated with variables, we would relate to each variable a token describing its instantiation status:

- "**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;
- "**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. The tokens are also named *modes*.

Different sets of tokens have been proposed for specific applications. For instance, [MU87] uses the set $\{\mathbf{g}, \mathbf{ng}\}$ (ground and non-ground, respectively) to analyse the flow of Prolog programs; [DW86] and [BJCD87] use the set $\{\mathbf{e}, \mathbf{c}, \mathbf{f}, \mathbf{d}\}$ (respectively: empty, closed or ground, free and don't know) for mode inferencing; [Mel87] uses $\{II, IU, IM, U\}$ (respectively: totally ground, term with totally uninstantiated components, term which is not II or IU, and totally uninstantiated) also for mode inferencing.

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. Those subsequent copies of a clause with similar tokens associated with its variables can be safely discarded, since they do not add any relevant information to help the identification of those techniques employed in the procedure. The execution of our *collect*/2 example above, for instance, with the same query provided, yields the annotated clauses shown in Figure 3.3. The actual values are replaced by their modes, enhancing the similarities between clauses. Identical clauses need not be kept. In Figure 3.3 the second clause which is used twice during the execution has only one copy since its annotations are exactly the same. The annotations consisting of the modes of variables are called *mode-annotations* and the procedures with mode-annotations are called *mode-annotated procedures*.

A mode-annotated procedure is 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, which stands for all their copies.

A simple way to obtain the mode-annotated representation of a procedure is to analyse the clauses obtained in *prolog1*/4 which are stored in *execution*/1 and replace their annotations containing the actual values of variables with respective mode-annotations, getting rid of repeated clauses. The built-in Prolog predicates *var*/1 and *ground*/1 can be used to assign modes to variables based on their associated contents; the "**i**" mode

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

Figure 3.3: Mode-Annotated Clauses of Procedure *collect*/2

is assigned to a variable when its associated value satisfies neither *var*/1 nor *ground*/1. We explore other alternatives in the following sections.

In the next section we describe a method to get mode-annotated procedures using the conventional interpreter for Prolog programs. This interpreter is also called a *concrete interpreter* as opposed to an *abstract interpreter*: in the former, computations are carried out in the concrete domain of Prolog terms; in the latter, computations are performed over an abstract domain representing *sets* of actual Prolog terms. In Section 3.6 we explain a method to get the mode-annotations employing an abstract interpreter. It turns out that both approaches have advantages and drawbacks, which are listed at the end of their presentations.

3.5 Mode-Annotation via Concrete Interpretation

In the previous section we explained a means to represent the execution of a program using the instantiation modes of its variables. We also suggested a way to adapt the results of the enhanced meta-interpreter of Figure 3.2 replacing the actual values associated with the variables with their modes and getting rid of repeated clauses. In this section we provide another way to get the mode-annotated procedure: it consists of adapting the *prolog1*/4 procedure to prepare each annotation in terms of the modes of its variables, as the execution proceeds.

The mode-annotated clauses of a program execution can be obtained by a very simple adaptation to the *prolog1*/4 meta-interpreter, which translates the substitutions into mode-annotations. The *prolog2*/4 meta-interpreter shown in Figure 3.4 implements these adaptations. Predicate *mode_annotation*/2 converts a substitution `Th` with actual values associated with variables to a mode-annotation `MTh` in which the values are replaced by tokens. The system predicates *var*/1 and *ground*/1 are used to replace the value t associated with a variable x; for each x/t in `Th`, then

- if `var(`t`)`[1] holds then $x/\mathbf{f}$ is in `MTh`;
- if `ground(`t`)` holds then $x/\mathbf{g}$ is in `MTh`;

[1] The *var*/1 predicate holds when the content of a variable is an empty memory cell.

```
prolog2((G,Gs),Th0,Th,(AnG,AnGs)):-
   prolog2(G,Th0,Th1,AnG),
   prolog2(Gs,Th1,Th,AnGs).
prolog2(\+G,Th,Th,(\+G,MTh)):-
   \+ prolog2(G,Th,_),
   mode_annotation(Th,MTh).
prolog2(G,Th0,Th,(G,MTh)):-
   system(G),
   call1(G,Th0,Th),
   mode_annotation(Th,MTh).
prolog2(G,Th0,Th,(G,MTh)):-
   clause1(G,Th0,(Hd:-Bd),Th1),
   prolog2(Bd,Th1,Th2,AnnBd),
   update(Th0,Th2,Th),
   mode_annotation(Th,MTh),
   mode_annotation(Th1,MTh1),
   update_clauses((Hd:-MTh1,AnnBd)).
```

Figure 3.4: Concrete Interpreter to Record the Mode of Variables

- if neither `var`(t) nor `ground`(t) hold then $x/\mathbf{i}$ is in `MTh`.

In order to keep the meta-interpreter simple, we also employ here a global structure to store the execution. The *update_clauses*/1 predicate is employed to perform the insertion of the recently obtained mode-annotated clause into the execution. Since repeated clauses should be discarded, a test must be performed before the clause is actually inserted. If the recently obtained clause is already in the execution then it is ignored; otherwise it is inserted.

3.5.1 Advantages and Drawbacks

The mode-annotation of a procedure using an enhanced (concrete) 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 as 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.

Non-termination might not be an important issue in this context for two reasons. Firstly, and more importantly, the current analysis relies on the participation of an expert 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 detect the techniques of the procedure, one would preferably need 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.

3.6 Mode-Annotation via Abstract Interpretation

In the previous section we described a simple way of mode-annotating a clause by concretely interpreting a program with respect to an initial query. As the execution proceeds, we collect information about the variables of each subgoal after its execution, and use this information to annotate the clauses.

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, the main drawback of this approach, to be exemplified later, is the potential inaccuracy of the information obtained.

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.

3.6.1 Abstract Domain

An abstract domain consists of a set of tokens each of which represents a possibly infinite set of actual values. We 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 are employed for this purpose. However, due to limitations inherent in the abstract interpretation techniques used here, an extra token "**?**", standing for any of the other tokens, has to be included. Having the token "**?**" associated with 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))`", and so on. Token "**i**" stands for an abstraction of Prolog constructs such as "`f(X,a)`", "`[foo|X]`", and so on, (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 3.5. The abstract interpretation functions simulating concrete Prolog computations are also naturally defined; the fundamental one, the binding of variables, is explained below.

We say that a token T subsumes another token T', denoted by $T \sqsupseteq T'$, when the instantiation status depicted by T can also be depicted as T'. Mode "**i**", for instance, subsumes mode "**g**", since all ground variables are also instantiated. The "**?**" mode

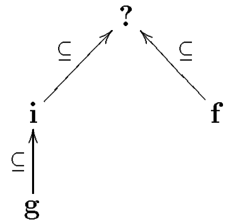


Figure 3.5: Set Inclusion Relationship between Tokens

subsumes all other modes. More formally we have:

Definition 3.6.1 A token T subsumes another token T', $T \sqsupseteq T'$, if one of the cases below holds

1. $T = T'$,
2. $T = \mathbf{i}, T' = \mathbf{g}$,
3. $T = \mathbf{?}$.

The subsumption between tokens faithfully reproduces the relation between the sets of values represented by each token: $T \sqsupseteq T'$ if, and only if, $T \supseteq T'$. We shall assume that the subsuming (or subsumed) token is always minimal, that is, if we want to find a token T which subsumes a given token T' (or, alternatively, a token T' subsumed by a given token T) then we shall consider the possible outcomes in the following ordering:

1. $T = T'$, otherwise
2. $T' = \mathbf{f}$ and $T = \mathbf{?}$, otherwise
3. $T' = \mathbf{g}$ and $T = \mathbf{i}$, otherwise
4. $T' = \mathbf{g}$ and $T = \mathbf{?}$, otherwise
5. $T' = \mathbf{i}$ and $T = \mathbf{?}$.

This assumption defines a least upper bound for each set of values represented by a token.

3.6.2 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 (see Section 1.5) where only explicit unifications are permitted, enable us to view computations as the effect of these explicit unifications over the mode-annotations. The unification of variables is abstracted in terms of changes made to their associated tokens. The possible changes of token T associated with x into another token T', denoted by $T \preceq T'$, is defined by the arrows in the graph of Figure 3.6. The $\preceq$ relation captures the possible changes

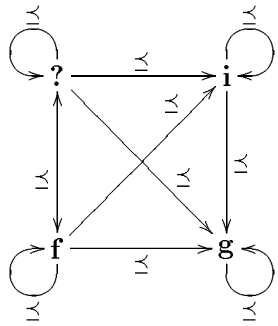


Figure 3.6: Possible Changes to Tokens

of the instantiation status of a variable during the execution of a subgoal: a variable whose associated token is "**f**" (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 3.1. The first column shows the possible values of x and the first line 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 $x/\mathbf{i}$ with $y/\mathbf{i}$ does not alter their tokens.

In our implementation of the abstract interpreter the sharing of variables within subterms is not recorded. If, for example, we are given a substitution $\{\mathtt{X}/\mathbf{f}, \mathtt{Y}/\mathbf{f}\}$ and a subgoal `X = f(Y)` to interpret abstractly, then its outcome is $\{\mathtt{X}/\mathbf{i}, \mathtt{Y}/\mathbf{f}\}$, without the sharing between `X` and `Y`, as there would be in the concrete interpretation of Prolog. If `Y` later becomes ground the mode of `X` is not updated to ground; if, however, `X` becomes ground the mode of `Y` is updated to "**?**". This limitation is responsible for the inaccuracy of the mode-annotations given by the abstract interpreter, as explained below. One should notice, though, that these modes are correct, although less accurate, with respect to those obtained in the concrete interpretation: the modes of the latter are respectively subsumed by those of the former.

3.6.3 Abstract Interpretation with Extension Tables

In order to mode-annotate a procedure via abstract interpretation we had 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 were 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 1.5.
- *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.

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

Table 3.1: Abstract Unification of Variables x and y

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 $\{\mathtt{X}/\mathbf{g}, \mathtt{Y}/\mathbf{f}, \mathtt{Z}/\mathbf{f}\}$, yields substitution $\{\mathtt{X}/\mathbf{g}, \mathtt{Y}/\mathbf{g}, \mathtt{Z}/\mathbf{g}\}$. The abstract interpreter defines the meaning of each Prolog system predicate in terms of changes in the tokens of the variables involved.

During our 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 [Die87] 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.

3.6.4 An Abstract Interpreter to Mode-Annotate a Program

We have adapted a meta-interpreter to perform the abstract interpretation of a Prolog program, and to insert the mode-annotations with the modes of variables after the abstract interpretation of each subgoal. We have incorporated an extension table mechanism as the one suggested in [Die87] and roughly followed those abstract interpreters shown in [Wae88] and in [Bow92].

The meta-interpreter shown in Figure 3.7 implements an abstract interpreter. The

```
abs_int((G,Gs),Th0,Th,(AnG,AnGs)):-
   abs_int(G,Th0,Th1,AnG),
   abs_int(Gs,Th1,Th,AnGs).
abs_int(\+G,Th,Th,(\+G,Th)).
abs_int(G,Th0,Th,(G,Th)):-
   system(G),
   abs_call(G,Th0,Th).
abs_int(G,Th0,Th,(G,Th)):-
   in_xtable(G,Th0,Th).
abs_int(G,Th0,Th,(G,Th)):-
   all_clauses(G,Cls),
   abs_int_procedure(Cls,G,Th0,Th),
   insert_xtable(G,Th0,Th).

abs_int_procedure(Cls,G,Th0,Th):-
   xtable(XTBef),
   abs_int_clauses(Cls,G,Th0,AnCls),
   xtable(XTAft),
   XTBef = XTAft,
   in_xtable(G,Th0,Th),
   insert_clauses(AnCls).
abs_int_procedure(Cls,G,Th0,Th):-
   abs_int_procedure(Cls,G,Th0,Th).

abs_int_clauses([],_,_,[]).
abs_int_clauses([(Hd:-Bd)|Cls],G,Th0,[(Hd:-Th1,AnnBd)|AnCls]):-
   unify(Hd,Bd,G,Th0,Th1),
   abs_int(Bd,Th1,Th2,AnBd),
   abs_int_clauses(Cls,G,Th0,AnCls).
```

Figure 3.7: Abstract Interpreter to Record the Mode of Variables

abs_int/4 predicate abstractly interprets the subgoals of the body of a clause. The *abs_call*/3 predicate describes the meaning of each system predicate in terms of changes performed to the modes of variables stored in `Th0`; for instance, those system predicates that can only perform tests, such as the comparator `>`, the *integer*/1 predicate, and so on, have the following clauses:

```
abs_call((_>_),Th,Th).
abs_call(integer(_),Th,Th).
```

and so on, with unchanged substitutions. The mode-altering system predicates have more elaborate formulations, in which only the modes of those variables of the subgoal are updated and the others remain unaltered.

The fourth clause of *abs_int*/4 checks if a user-defined predicate has an entry in the extension table, via predicate *in_xtable*/3: if it has an entry (or more than one) then *in_xtable*/3 subsumes the success modes of all the entries and returns it as `Th`. The fifth clause collects the definition of subgoal `G`, abstract interprets it, obtaining `Th`, and inserts the entry comprising of `G`, `Th0` and `Th` in the extension table.

The *abs_int_procedure*/4 controls the abstract interpretation of the clauses defining a predicate: if there is a new entry in the extension table, the abstract interpretation of the set of clauses defining `G` is performed again, from the beginning (second clause of *abs_int_procedure*/4, as the exception of testing `XTBef` (extension table *before* the abstract interpretation) against `XTAft` (extension table *after* the abstract interpretation). The abstract substitution `Th` is obtained by checking in the extension table; the mode-annotated clauses obtained by *abs_int_clauses*/4 are inserted in a global data structure by *insert_clauses*/1. The *abs_int_clauses*/4 applies the *abs_int*/4 to each clause of the procedure and obtains its mode-annotated version.

The implementation of the abstract interpreter of Figure 3.7 may loop, if the non-recursive clauses are not processed first, initialising the extension table with the exit modes of their head goal variables. Since the order of clauses is immaterial in abstract interpretation and is not important for our characterisation of programming techniques, this problem can be circumvented by having the clauses reordered prior to the application of the abstract interpreter, the non-recursive clauses coming first. Alternatively, a suspension mechanism can be used [Wae88, Bow92] so that recursive calls have their abstract interpretation suspended until the extension table has an entry.

3.6.5 Advantages and Drawbacks

There are two main advantages when using abstract interpretation to mode-annotate programs:

- *Termination* — the abstract interpretation of programs eventually terminates. This is true even for normally non-terminating programs. The program has its execution *simulated* by having each clause separately interpreted, but the potentially non-terminating flow of control of the program interpreted is not actually established.

- *Completeness* — the supplied query is abstracted in terms of its modes and the analysis is carried out with respect to this abstraction. If the same query is supplied to both concrete and abstract interpreters, the clauses obtained in the latter form a superset of those obtained in the former.

These features overcome the disadvantages of the concrete interpretation referred to previously. The system does not have to 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. In our current implemented version, 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. This feature of our implementation might cause mistakes in some circumstances: given the following program

```
p(A,B):-
   q(A,B),
   A = a.

q(A,B):-
   A = B.
```

if procedure $p/2$ is mode-annotated with respect to query `?-p(A,B)` (`A` and `B` are free) using the abstract interpreter above, then the following version is obtained

```
p(A,B):-     {A/f,B/f}
   q(A,B),   {A/f,B/f}
   A = a.    {A/g,B/f}
```

The final mode of B was incorrectly inferred to be **f**, because the variable dependency between A and B defined in procedure *q*/2 was not recorded. This problem can be circumvented, though, if the dependency of variables is also stored in the explicit substitution Th of our implementation.

To illustrate the points above, we shall obtain a mode-annotated version of procedure *collect*/2 with respect to the query "collect([foo1,foo2],L)" this time via abstract interpretation. This query is abstracted as "collect(X/g,Y/f)" and the mode-annotated procedure shown in Figure 3.8 below is obtained. It is worth noticing

```
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 3.8: Clauses of Procedure *collect*/2 Annotated with the Instantiation of Variables via an Abstract Interpreter

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 3.5). 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 token "**g**" is associated with both X and Ys, then B should also have "**g**" associated with it, but this does not happen because no variable dependency is recorded.

The order of the clauses is not that of conventional Prolog execution. It is the order of the clauses as in the definition of *collect*/2, in Section 3.3. The meta-interpreter of Figure 3.7 abstractly interprets each clause of the procedure, collecting the modes of its variables: in *collect*/2 the non-recursive clause came first, followed by the recursive clauses. If the definition of the procedure were such that the recursive clauses came first, the mode-annotated clauses would still bear a similar ordering, non-recursive clauses first. This is due to the reordering of the clauses before the abstract interpretation is performed, as explained above. The relative ordering among the non-recursive and recursive clauses is, however, maintained.

3.7 Duplication of Mode-Annotations

A mode-annotated subgoal can be made a stand-alone description of a computation if we add to it the mode-annotation *before* its execution. The mode-annotation before the subgoal can be obtained by replicating the mode-annotation after its previous subgoal. Given a mode-annotated clause of the form

$$H \; {:\!-} \; \theta_0 \;\; S_1\,\theta_1, S_2\,\theta_2, \ldots, S_n\,\theta_n$$

where H and S_i are subgoals and the θ_i are mode-annotations, the new form of mode-annotation can be obtained by inserting a copy of θ_i before S_{i+1}:

$$H \; {:\!-} \; \theta_0 \;\; \theta_0\,S_1\,\theta_1, \theta_1\,S_2\,\theta_2, \ldots, \theta_{n-1}\,S_n\,\theta_n$$

A simple process could translate clauses of the former kind into those of the latter. Alternatively, the meta-interpreters for concrete and abstract mode-annotation shown previously can be easily altered to build this new form of mode-annotated clauses. We shall assume from now on that the mode-annotated clauses are of the latter kind.

This replication is also made necessary because during the separation of the contributions of each argument position (the *argument slicing*, explained in Chapter 4) some mode-annotated subgoals may not become part of a contribution and their gap would make the description of the computations in that contribution inaccurate. Another reason for the duplication of mode-annotations is the notational convenience: individual subgoals with the mode-annotations before and after it provide us with a description of those computations performed at that point, and as we shall see on Chapter 5, this description plays an important role in the organisation of our library.

A simpler representation scheme would provide the mode-annotations only after those subgoals that have changed the mode of variables. For instance, our mode-annotated *collect*/2 procedure obtained by the abstract interpreter and shown in Figure 3.8, could be represented as:

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

where the mode-annotations after subgoals "`A = []`" and "`integer(X)`" are omitted. This representation, although simplified, would have to be adapted during the argument slicing to provide a faithful account of the ongoing computations. Furthermore, when subgoals are required to be addressed together with the modes of variables before and after them, a more sophisticated notation would become necessary.

3.8 Subsumption of Mode-Annotated Clauses

During the mode-annotation via the concrete interpreter, if an argument position in the subgoal of a clause appears with different modes then that clause will appear more than once with different mode-annotations. For instance, the mode-annotation of *collect*/2 using a concrete interpreter with respect to query `collect([X,foo,10],L)` provides us with the following[2]:

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

[2] As stated in the previous subsection, the mode-annotated procedures will be considered from now on as having their mode-annotations replicated.

The third clause appeared replicated because the first argument in the head goal of the clauses gets values with different instantiation status after the match of recursive calls; it is assigned the values `[X,foo,10]`, `[foo,10]`, `[10]` and `[]`, in this order: the first and second values are handled by the third clause of *collect*/2, and since they have different modes ("**i**" and "**g**", respectively) two copies of the third clause appeared, with different modes associated with `A`.

This poses a problem to the current approach aiming at our ideal description of an execution representing the use of each clause exactly once. We have devised a solution to this problem using the notion of *subsumption* of modes (Def. 3.6.1) and extending it to cope with mode-annotations and mode-annotated clauses. This solution consists of checking a mode-annotated procedure for copies of the same clause; if there are copies of a clause with different mode-annotations, then we devise a mode-annotated clause whose mode-annotations *subsume*, respectively, the mode-annotations of all those copies.

A mode-annotated clause $\widetilde{C}$ subsumes mode-annotated clauses $\widetilde{C}_1, \ldots, \widetilde{C}_m$ if its subgoals are equal and the mode-annotations of $\widetilde{C}$ subsume respectively those of each $\widetilde{C}_i$:

Definition 3.8.1 $\widetilde{C}$ of the form

$$H \texttt{ :- } \theta_0 \;\; \theta_0 \, S_1 \, \theta_1, \theta_1 \, S_2 \, \theta_2, \ldots, \theta_{n-1} \, S_n \, \theta_n$$

subsumes $\{\widetilde{C}_1, \ldots, \widetilde{C}_m\}$, $\widetilde{C} \sqsupseteq^{\widetilde{C}} \{\widetilde{C}_1, \ldots, \widetilde{C}_m\}$, if, and only if, each $\widetilde{C}_i, 1 \leq i \leq m$ is of the form

$$H \texttt{ :- } \theta_0^i \;\; \theta_0^i \, S_1 \, \theta_1^i, \theta_1^i \, S_2 \, \theta_2^i, \ldots, \theta_{n-1}^i \, S_n \, \theta_n^i$$

and $\theta_j \sqsupseteq^\theta \theta_j^i$, for every $j, 0 \leq j \leq n$.

The subsumption of mode-annotations $\sqsupseteq^\theta$ makes use of the subsumption of tokens, given by Definition 3.6.1, and is defined as:

Definition 3.8.2 $\theta = \{x_1/T_1, \ldots, x_n/T_n\}$ subsumes $\theta' = \{y_1/T_1', \ldots, y_m/T_m'\}$, $\theta \sqsupseteq^\theta \theta'$, if, for every y_j there is an x_i such that $x_i = y_j$ and $T_i \sqsupseteq T_j'$.

It should be clear that the $\sqsupseteq^{\theta}$ relation is reflexive and transitive. The subsumption of mode-annotated clauses is necessary to circumvent the problem of having more than one mode-annotated version of the same clause: this happens in certain conditions during the mode-annotation using a concrete interpreter. Different mode-annotated versions of the same clause would never happen when the mode-annotation is performed via an abstract interpreter.

During the mode-annotation via an abstract interpreter, however, a similar subsumption process takes place, but on a subgoal level: if there is more than one mode associated with a variable of a subgoal then a mode subsuming them all will become associated with that variable instead.

3.9 Execution with Respect to Sets of Queries

The results of the mode-annotation using a concrete interpreter could be improved if we allow a program to be executed with respect to a *set* of queries. According to this proposal the program is executed with respect to each query of the set and the mode-annotated clauses obtained are then stored. Each new mode-annotated clause obtained is checked for copies with different mode-annotations: these are replaced by their subsuming mode-annotated clause (Section 3.8).

This improvement naturally adapts to the mode-annotation via abstract interpretation: the set of (concrete) queries is analysed and a query consisting of modes subsuming all the queries would be used in the abstract interpreter.

The user is, however, still responsible for supplying a convenient set of queries. A set of queries with completely different queries would not be of much use: disparate queries may introduce "**i**" or "**?**" modes and these modes do not tell much about the computations being performed at each subgoal. If, however, a set of queries bearing similar modes but different contents is supplied, we have a more substantial number of examples with respect to which the techniques of procedures will be analysed.

If, for instance, our *collect*/2 procedure is supplied with the following set of queries

$$\{\texttt{collect([],L)}, \texttt{collect([1,2,3],R)}, \texttt{collect([foo1,foo2],S)}\}$$

we would obtain via a concrete interpreter, the same mode-annotated clauses as those of Figure 3.3; the outcome using an abstract interpreter would be exactly as that of Figure 3.8.

3.10 Conclusions

We have proposed a means to complement the syntax of a procedure with information concerning its execution with respect to a query or a set of queries. Our objective is to gather more information about a given procedure so as to be able to detect its programming techniques.

Initially we explained a rather detailed approach in which the actual values assigned to variables during the execution are represented. We then developed this first proposal to a more economic description in which the modes of variables are used instead. A procedure augmented with the modes of its variables is called a *mode-annotated procedure*.

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 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.

The mode-annotation of procedures can be achieved by concrete or abstract interpretation. Both approaches have been implemented and our proposed method to detect techniques (Chapter 4) 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 or set of queries, important clauses may be left out. The potential inaccuracy of the mode-annotations of the abstract interpreter degrades the analysis of techniques gradually as it gets more inaccurate. The potential absence of clauses in the output of the concrete interpretation prevents the appropriate extraction of the programming techniques of that procedure, but this can be automatically detected and a human intervention requested to supply another query.

The mode-annotation obtained after the concrete interpretation 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.

We can appropriately describe the instantiation status of any variable in an execution with the adopted 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 spot 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.

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 chapter was to describe each approach and explain some implementational issues. The advantages and deficiencies of each alternative were listed and compared and samples of their outputs were shown.

3.11 Summary

In this chapter we have:

- proposed a representation for the execution of programs which complements the syntax of a procedure with information about the manner its commands have been used;
- described how to obtain the execution of programs in the proposed representation, either employing the actual values associated with variables or the information on the instantiation mode of variables;
- compared the different approaches to obtaining the execution of programs in our suggested representation, listing their advantages and drawbacks; some problems arising were also considered and solutions to them were proposed.

Chapter 4

Extracting Prolog Programming Techniques

4.1 Introduction

In this chapter we present the method used in our knowledge-management tool for detecting and extracting the programming techniques of Prolog programs[1]. Given a mode-annotated procedure (as explained in the previous chapter), our method partitions it into the contributions of each argument position. These contributions may share variables and this information is also represented in our formal characterisation of programming techniques. The proposed method carries out the analysis and extraction of the techniques of a procedure using its mode-annotated clauses.

The extraction method consists of a two-stage process. In the first stage, the mode-annotated procedure $\widetilde{P}$ is partitioned into a sequence $\langle \widetilde{P}_1, \ldots, \widetilde{P}_n \rangle$ of single-argument procedures, its *argument slices*, describing the contributions of each argument position. Each argument position in the head goal of a clause has an argument slice consisting of those subgoals *relevant* to the argument position in every clause. A notion of *relevance* of a subgoal with respect to an argument slice is formally described in this chapter.

The second 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. The clause annotations state the required and offered

[1] The material of this chapter has been published as a departmental technical paper [Vas94a] and a shorter version of it has appeared as a research paper [Vas94b] and in the proceedings of the XI Brazilian Symposium on Artificial Intelligence (Ceará, Brazil, 17-20 Oct, 1994).

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.

4.2 Working Example

In this section we provide a working example to informally explain the proposed method. Let there be the following formulation of *prefix*/2 predicate [SS86, O'K90],

```
prefix(A,B):-                          prefix([],_).
   A = [].
prefix(A,B):-                          prefix([X|Xs],[X|Ys]):-
   A = [X|Xs],                               prefix(Xs,Ys).
   B = [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 is 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 programming techniques of a procedure are analysed with respect to a particular use of its clauses: in the previous chapter we proposed to represent this usage by merging the syntax of the procedure with its execution with respect to a query or set of queries. Let us suppose that the programming techniques of *prefix*/2 are to be analysed with respect to the set of queries

$$\{\texttt{prefix(A,[])}, \texttt{prefix(A,[1,2,3])}, \texttt{prefix(A,[a,b])}\}$$

That is, we want to detect the programming techniques employed when *prefix*/2 is used to build the prefixes of a list. The mode-annotated clauses depicting this usage of *prefix*/2, obtained via an abstract interpreter, are:

```
                          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/g,Xs/f,Ys/g}
{A/i,B/g,X/g,Xs/f,Ys/g}      prefix(Xs,Ys).  {A/i,B/g,X/g,Xs/i,Ys/g}
```

In the first stage of the extraction method, the mode-annotated 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 4.1, without their mode-annotations. The notion of *relevance* of a subgoal with respect to an argument

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

Figure 4.1: Argument Slices of Procedure *prefix*/2

slice is one of the contributions of this work, and is formally stated in Section 4.3, together with a detailed account of how the argument slicing is performed.

During the second 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 4.2 (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 X in the first argument slice is required and it

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

Figure 4.2: Clause-Annotated Argument Slices of Procedure *prefix*/2

is also offered in the second argument slice. 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. The idea of a technique as being the syntax of the procedure and its usage underlies our 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 (the second argument slice) and another building a list with those values `X` obtained as another list is decomposed (the combination of first and second argument slices).

4.3 Argument-Slicing of Mode-Annotated Procedures

The first stage of our method partitions a mode-annotated procedure 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.

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; or
- 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 the argument slicing.

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.

Given a mode-annotated procedure $\widetilde{P}$ with arity n and clauses $\widetilde{C}_1, \ldots, \widetilde{C}_m$, each of the form

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

where $\vec{S}_i, 0 \leq i \leq r+1$, are possibly empty vectors of non-recursive (*i.e.* they contain no references to predicate p/n) mode-annotated subgoals[2], $\vec{S}_i = \widetilde{S}_0^i, \ldots, \widetilde{S}_n^i$, and each $\widetilde{S}_k^i$ is of the form $\theta \, q(\ldots \, y_1 \, \ldots \, y_m \, \ldots) \, \theta'$ then its i-th mode-annotated argument slice $\widetilde{P}_i, 1 \leq i \leq n$, is obtained by restricting the mode-annotated subgoals in each clause $j, 1 \leq j \leq m$, to those subgoals relevant to position i, denoted by $\widetilde{C}_{[j,i]}$. The i-th mode-annotated argument slice $\widetilde{P}_i, 1 \leq i \leq n$ consists of clauses $\widetilde{C}_{[1,i]}, \ldots, \widetilde{C}_{[m,i]}$.

For each mode-annotated clause $\widetilde{C}_j$ of the form above, its i-th mode-annotated argument slice, $\widetilde{C}_{[j,i]}$, is of the form

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

The head goal is sliced by restricting its variables to x_i occupying position i; recursive

[2] Sometimes mode-annotated subgoals will be shown enclosed in boxes to facilitate their visualisation.

subgoals are sliced simply by restricting their arguments to the variable $x_{[j,i]}$ occupying position i in the j-th recursive call. The slicing of a vector $\vec{S}_{[k,i]}$ is performed by checking each of its mode-annotated subgoals for its relevance to the argument slice: if the mode-annotated subgoal is relevant then it becomes part of the argument slice; otherwise it is suppressed.

The test of the mode-annotated subgoals in $\vec{S}_k$ for their relevance to a clause of an argument slice i is carried out with respect to variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$ of the argument-sliced head and recursive subgoals. These variables are essential to the clause since they can control the flow of execution or be responsible for important computations in the procedure: a subgoal is relevant if it affects these variables by

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

A subgoal relevant to an argument position would either contribute to the computations of the procedure (first case above) or interfere with its flow of control (second case). These cases are formalised below as two decision procedures, $relevant^c(\widetilde{C}, i, \widetilde{S})$ and $relevant^t(\widetilde{C}, i, \widetilde{S})$, respectively.

The i-th argument slice of a vector of mode-annotated subgoals is obtained by checking if each of its mode-annotated subgoals satisfies one of the *relevant* relations above: if it does, then it is copied onto the argument slice; otherwise it is discarded. Given a vector $\vec{S} = \widetilde{S}_0, \ldots, \widetilde{S}_n$ in $\widetilde{C}$, its i-th slice $\vec{S}_i$ of subgoals relevant to argument slice i consists of those subgoals $\widetilde{S}_j$, in the same order they appear in $\vec{S}$, such that $relevant^c(\widetilde{C}, i, \widetilde{S}_j)$ or $relevant^t(\widetilde{C}, i, \widetilde{S}_j)$.

Example: Let there be the following mode-annotated procedure *sum*/2 (it holds if the second argument is the sum of the elements of the list comprising the first argument)

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

Its first mode-annotated argument slice is

```
                  sum(A):-       {A/g}
         {A/g}       A = [].     {A/g}
                  sum(A):-       {A/g,C/f,D/f}
 {A/g,C/f,D/f}       A = [C|D],  {A/g,C/g,D/g}
 {A/g,C/g,D/g}       sum(D).     {A/g,C/g,D/g}
```

its mode-annotations have been altered to remove those variables which do not appear in the clause anymore. The head goals and recursive calls show only the variable which occupied first position in the initial mode-annotated procedure. The subgoal "`B = 0`" in the first clause was removed because it is not relevant to this first argument slice; "`B is E + C`" in the second clause was also removed for similar reasons, even though it has a variable `C` which was obtained by the decomposition of the list `A`: this happened because `C` is merely being used to calculate `B`, that is, `C` is "donating" its value to the computation of `B`. The analysis of the relevance of non-recursive subgoals in the body of a clause takes into account indirect relationships between variables, such as the relationship between `A` and `C` in the second clause above, and is explained below.

The second mode-annotated argument slice of *sum*/2 above is

```
                  sum(B):-       {B/f}
         {B/f}       B = 0,      {B/g}
                  sum(B):-       {B/f,C/f,E/f}
 {B/f,C/g,E/f}       sum(E),     {B/f,C/g,E/g}
 {B/f,C/g,E/g}       B is E + C. {B/g,C/g,E/g}
```

those variables which do not appear in the clauses have been removed from the mode-annotations. The subgoal "`A = []`" in the first clause was removed since it is irrelevant to the computation of `B`. Subgoal "`A = [C|D]`" in the second clause was also removed, although it describes the manner `C` was obtained: as far as the final result computed by this argument slice is concerned, it is enough to have a value in `C` provided at the end of each recursive clause.

4.3.1 Relevance of Non-Recursive Subgoals

Analysing the relevance of a non-recursive 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 procedures $relevant^t$ and $relevant^c$ to check the relevance of a subgoal to an argument slice rely 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 relationships between variables of different subgoals*: they support the recognition of *chains* of subgoals which may be 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.

These items are explained in the following sections.

4.3.2 Relationships between Variables of Subgoals

The relevance of a non-recursive subgoal to an argument slice can be of an indirect kind, in which the variables $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$ of the head goal and recursive subgoals are not directly involved. Our decision procedures must be provided with a means to account for these indirect relationships. We propose a way to automatically infer and represent the relationships between variables of subgoals capturing the effects of those computations performed. Our proposal, described below, extends Bowles' ([Bow92], Chapter 3) characterisation of techniques in terms of unifications, to cope with more procedural aspects of Prolog.

The mode-annotated subgoals provide us with a description of the effects of the computations performed by each subgoal in terms of the modes of its variables. We employ this information to model the relationships between variables: a variable is seen as "donor" of its value for the computation of a subgoal, or as a "receiver" of a value obtained by the computation of a subgoal. For instance, the mode-annotated subgoal

$$\{\ldots \mathtt{B}/\mathbf{f}, \mathtt{C}/\mathbf{g}, \mathtt{E}/\mathbf{g} \ldots\}\ \mathtt{B\ is\ E\ +\ C}\ \{\ldots \mathtt{B}/\mathbf{g}, \mathtt{C}/\mathbf{g}, \mathtt{E}/\mathbf{g} \ldots\}$$

defines relationships between `B` and `C` and between `B` and `E`: `C` and `E`, associated with "**g**" throughout the execution of the subgoal, *provide* their values for the computation performed by the *is*/2 predicate, and `B` *receives* its value by means of it. We represent this donor-receiver relationship by means of the symbol "$\lhd$": $x \lhd y$ can be understood as "variable y is donating its value to compute the value of x". In the example above, we have that $\mathtt{B} \lhd \mathtt{C}$ and $\mathtt{B} \lhd \mathtt{E}$.

A variable is considered to be a donor in a subgoal if its content, represented by its modes in the mode-annotation, remains unchanged. A variable is considered a receiver if its content is changed. The actual content of a variable is abstracted as a token and this has to be considered during this analysis. 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 $\theta\ S\ \theta'$, 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:

- $\mathit{fixed}(x, \theta, \theta')$ holds if $x/T \in \theta$ and $x/T \in \theta'$ and $T \in \{\mathbf{g}, \mathbf{f}\}$.
- $\mathit{change}(x, \theta, \theta')$ holds if $x/T \in \theta$ and $x/T' \in \theta'$ and $T = \mathbf{f}, T' \in \{\mathbf{g}, \mathbf{i}\}$.
- $\mathit{unknown}(x, \theta, \theta')$ holds if $\neg\mathit{fixed}(x, \theta, \theta')$ and $\neg\mathit{change}(x, \theta, \theta')$.

where the "$\neg$" operator represents the negation by failure: $\neg P$ holds if it is not possible to prove P.

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*. Variables whose tokens satisfy the relations *unknown* pose a serious problem, because nothing accurate can be stated about their relationships. A mode-annotated subgoal of the form

$$\{\ldots \mathtt{X}/\mathbf{i}, \mathtt{Y}/\mathbf{?} \ldots\}\ \mathtt{X\ =\ Y}\ \{\ldots \mathtt{X}/\mathbf{i}, \mathtt{Y}/\mathbf{i} \ldots\}$$

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 cope with these cases, our donor-receiver symbol "$\lhd$" is given a broader interpretation: $x \lhd y$ means "y is donating its value to compute the changes that may have happened in x" or alternatively, "x *may* have been changed using y".

Each mode-annotated subgoal defines its own set of $\lhd$-relations between its variables. Given a mode-annotated subgoal $\widetilde{S}$ of the form $\theta\ S\ \theta'$, its set of $\lhd$-related pairs (or simply $\lhd$-pairs), denoted by $\rho_{\widetilde{S}}$, is prepared by analysing each pair of distinct variables x and y in S, and if one of the cases below holds

1. $\mathit{change}(x, \theta, \theta')$ and $\mathit{fixed}(y, \theta, \theta')$,
2. $\mathit{unknown}(x, \theta, \theta')$ and $\mathit{fixed}(y, \theta, \theta')$,
3. $\mathit{change}(x, \theta, \theta')$ and $\mathit{unknown}(y, \theta, \theta')$,

4. $unknown(x, \theta, \theta')$ and $unknown(y, \theta, \theta')$.

then $x \lhd y$ should be inserted into $\rho_{\widetilde{S}}$. If all the variables in $\widetilde{S}$ are fixed, then no $\lhd$-relation is established and its set $\rho_{\widetilde{S}}$ is empty.

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

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

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

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

Relationships between Variables of a Clause

A variable may be indirectly related, via an intermediate subgoal, to another variable. To deal with these situations, 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: the set $\rho_{\vec{S}}$ of $\lhd$-pairs of $\vec{S} = \widetilde{S}_1, \ldots, \widetilde{S}_n$ is

$$\rho_{\vec{S}} = \bigcup_{i=1}^{n} \rho_{\widetilde{S}_i}$$

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: the set $\rho_{\widetilde{C}}$ of $\lhd$-pairs of $\widetilde{C}$ is

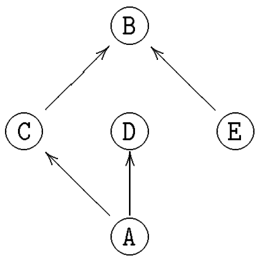
$$\rho_{\widetilde{C}} = \bigcup_{i=0}^{r+1} \rho_{\vec{S}_i}$$

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:



Indirect $\lhd$-Relationships between Variables

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: 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. We have adopted a depth-first approach in our implementation.

Example: Given the set $\rho_{\widetilde{C_2}}$ of the previous example, it is the case that B $\lhd^*_{\rho_{\widetilde{C_2}}}$ A, that is, variable B is indirectly related to variable A (via intermediate variable C).

The definition of $\lhd^*_\rho$ above has to 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: 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 $x_i \lhd^*_\rho y_j$.

4.3.3 Meaning of a Subgoal = Syntax + 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. To assist this decision, the mode-annotations provide more information concerning the way each predicate has been employed in terms of the instantiation status of its variables.

Subgoals employing those "◇" system predicates different from = and =.., that is, =:=, >, >=, and so on, are always tests, whatever their mode-annotations may be. The same is true for the "¶" built-in tests *atom*/1, *var*/1, and so on. In order to find out what computations those subgoals employing other system predicates perform, the mode-annotations have to be used. The = system predicate, for instance, could either assign a value to a variable, that is,

{...X/**f**,Y/**g**...} X = Y {...X/**g**,Y/**g**...}

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

{...X/**g**,Y/**g**...} X = Y {...X/**g**,Y/**g**...}

The mode-annotations provide us with the extra information we need to differentiate the two ways of employing 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 so the analysis carried out employing them inherits this imprecision.

Inaccurate mode-annotated subgoals contain variables satisfying the *unknown* relation. 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 of deciding 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*/2, 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 happens 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 simply to 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 admit 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 procedures proposed here for the relevance of a subgoal have some overlap between the cases of variables changing and remaining fixed, making the policy adopted 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. For instance, in the mode-annotated subgoal

$$\{\ldots,\mathtt{X/?},\ldots\}\ \mathtt{read(X)}\ \{\ldots,\mathtt{X/?},\ldots\}$$

the tokens associated with variable `X` are inaccurate "?" tokens, making it impossible to state which computation is actually taking place. Our approach, however, considers the different possibilities by formalising the potential changes ($\neg$*fixed*, *i.e. fixed* does not hold) or lack of changes ($\neg$*change*, *i.e. change* does not hold). In the example above it does not matter if the instantiation status of `X` is changing or not: if it is changing then that subgoal is relevant because it depicts an important computation; if the instantiation modes of `X` do not change then a test is being performed and the subgoal is also relevant to those argument slices related to `X`.

4.3.4 The $relevance^c$ of Subgoals

Non-recursive subgoals changing 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 to define the flow of control of the program or may be the final values computed by the programming technique.

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 4.3.1 $\widetilde{S}$ in $\widetilde{C}$ is c-relevant to argument slice i, $relevant^c(\widetilde{C}, i, \widetilde{S})$, if it has a variable x possibly being changed, $change(x, \widetilde{S})$, and 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, that is, $\forall y[(y \lhd x \in \rho_{\widetilde{C}}) \rightarrow (\{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\} \lhd^*_{\rho_{\widetilde{C}}} \{y\})]$

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

The second condition describes those subgoals computing intermediate values x employed to possibly change $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$. 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, in their turn, "donate" their value to $x_i, x_{[0,i]}, \ldots, x_{[r,i]}$.

The *change* relationship of a variable with respect to a mode-annotated subgoal employed in the definition above, is an enhancement of the *change* relationship of a variable with respect to a pair of mode-annotations described in Section 4.3.2. This newer form of *change* relation uses the syntax of the subgoal to complement the information from the mode-annotations. 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 4.3.2 A variable x has its content possibly changed in subgoal $\widetilde{S}$, denoted by $change(x, \widetilde{S})$, 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. $\widetilde{S} = \boxed{\theta\ x\ \texttt{is}\ f_i^j(\ldots)\ \theta'}, \neg \mathit{fixed}(x, \theta, \theta')$;

5. $\widetilde{S} = \boxed{\theta\ p(\ldots x \ldots)\ \theta'}, \neg \mathit{system}(p(\ldots x \ldots)), \neg \mathit{fixed}(x, \theta, \theta')$.

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*/1 or *get*/1 (denoted by $\Im$) such that its variable x is not fixed. The fourth case depicts those subgoals making use of *is*/2 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 `>`, `<`, and so on, 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

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

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*/2 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

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

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*/2 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*/2. All the other system predicates left out of the list above (*e.g.*, >, <, *write*/1, and so on) never perform changes in the contents of their variables.

It should be noted 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. 4.3.2 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.

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

	sum(A,B):-	{A/g,B/f}	$\rho_{\tilde{C}_1} = \emptyset$
{A/g,B/f}	A = [],	{A/g,B/f}	
{A/g,B/f}	B = 0,	{A/g,B/g}	
	sum(A,B):-	{A/g,B/f,C/f,D/f,E/f}	
{A/g,B/f,C/f,D/f,E/f}	A = [C|D],	{A/g,B/f,C/g,D/g,E/f}	$\rho_{\tilde{C}_2} = \{\texttt{C} \lhd \texttt{A}, \texttt{D} \lhd \texttt{A},$
{A/g,B/f,C/g,D/g,E/f}	sum(D,E),	{A/g,B/f,C/g,D/g,E/g}	
{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 $\widetilde{S} = \boxed{\{\texttt{A/g},\texttt{B/f}\}\ \texttt{A = []}\ \{\texttt{A/g},\texttt{B/f}\}}$, $\neg relevant^c(\widetilde{C}_1, i, \widetilde{S}), i = 1, 2$: 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 $\widetilde{S} = \boxed{\{\texttt{A/g},\texttt{B/f}\}\ \texttt{B = 0}\ \{\texttt{A/g},\texttt{B/g}\}}$, we have that $\neg relevant^c(\widetilde{C}_1, 1, \widetilde{S})$ (`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. 4.3.1 for the first argument slice, hence the negative result) and $relevant^c(\widetilde{C}_1, 2, \widetilde{S})$ (for the second argument slice `B` satisfies the first case of Def. 4.3.1, and hence the result).

3. $\neg relevant^c(\widetilde{C}_2, i, \boxed{\{\ldots,\texttt{A/g},\texttt{C/f},\texttt{D/f},\ldots\}\ \texttt{A = [C|D]}\ \{\ldots,\texttt{A/g},\texttt{C/g},\texttt{D/g},\ldots\}}), i = 1, 2$: this subgoal describes a data structure decomposition and is not covered by the cases of Def. 4.3.2, hence the subgoal is not c-relevant to any argument slice.

4. for $\widetilde{S} = \boxed{\{\ldots,\texttt{B/f},\texttt{C/g},\texttt{E/g},\ldots\}\ \texttt{B is E + C}\ \{\ldots,\texttt{B/g},\texttt{C/g},\texttt{E/g},\ldots\}}$, $\neg relevant^c(\widetilde{C}_2, 1, \widetilde{S})$ (`B` is the only changing variable, but as the result 2 above, it does not satisfy any of the case of Def. 4.3.1 for the first argument slice, and hence this negative result) and $relevant^c(\widetilde{C}_2, 2, \widetilde{S})$ (as in the previous result).

4.3.5 The *relevance*^{t} of Subgoals

Non-recursive subgoals performing tests 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. Test subgoals are relevant if at least one of its variables satisfies certain properties, given by the definition below:

Definition 4.3.3 $\widetilde{S}$ in $\widetilde{C}$ is t-relevant to argument slice i, $relevant^t(\widetilde{C}, i, \widetilde{S})$, if it has a variable x possibly being tested, $test(x, \widetilde{S})$, and 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, $\forall y[(y \lhd x \in \rho_{\widetilde{C}}) \rightarrow (\{x_i, x_{[0,i]}, \ldots, x_{[r,i]}\} \lhd^*_{\rho_{\widetilde{C}}} \{y\})]$

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

The first condition addresses those cases where one of 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 the subgoal 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 *test* relationship of a variable with respect to a mode-annotated subgoal used in the definition above formalises those cases when we can state, based on the mode-annotations and syntax of the subgoal, that a variable may have had its contents tested:

Definition 4.3.4 A variable x has its content possibly tested in subgoal $\widetilde{S}$, $test(x, \widetilde{S})$, if, and only if, one of the cases below holds:

1. $\widetilde{S} = \boxed{\theta\ x\ \diamond\ \ldots\ \theta'}$, $\diamond \notin \{\texttt{=}, \texttt{=..}\}$;

2. $\widetilde{S} = \boxed{\theta\ \ldots\ \diamond\ x\ \theta'}$, $\diamond \notin \{\texttt{=}, \texttt{=..}\}$;

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

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

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

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, `>`, `<`, and so on) 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 depict 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 ¶, *var*/1, *ground*/1, *integer*/1, and so on. The seventh case depicts subgoals using the system input predicates *read*/1 or *get*/1 with a non-changing variable. The eighth case represents those subgoals making use of *is*/2 such that the variable x on its left-hand side does not change: the *is*/2 operator is being employed to calculate the expression $f_i^j(\ldots)$ and to 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 one being tested. The last case shows the user-defined test subgoals: *user-test* is defined in Section 4.3.6 and it holds if the mode-annotated subgoal is a user-defined predicate which may fail.

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.

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:

for $\widetilde{S} = \boxed{\{\texttt{A/g}, \texttt{B/f}\}\ \texttt{A = []}\ \{\texttt{A/g}, \texttt{B/f}\}}$, $relevant^t(\widetilde{C}_1, 1, \widetilde{S})$ (for $test(\texttt{A}, \widetilde{S})$ and the first condition of the previous definition holds) and $\neg relevant^t(\widetilde{C}_1, 2, \widetilde{S})$ (there is only one x such that $test(x, \widetilde{S})$ but $x \notin \{x_j, x_{[0,j]}, \ldots, x_{[r,j]}\}$ and hence the negative result).

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

for $\widetilde{S} = \boxed{\{\ldots,\texttt{A/g,C/f,D/f},\ldots\}\ \texttt{A = [C|D]}\ \{\ldots,\texttt{A/g,C/g,D/g},\ldots\}}$, $relevant^t(\widetilde{C}_2, 1, \widetilde{S})$ (because $test(\texttt{A}, \widetilde{S})$ and the first condition of Def. 4.3.3 holds) and $\neg relevant^t(\widetilde{C}_2, 2, \widetilde{S})$ (there is only one x such that $test(x, \widetilde{S})$ but $x \notin \{x_j, x_{[0,j]}, \ldots, x_{[r,j]}\}$).

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

4.3.6 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. It should be noted that we are not referring to failures caused by an aborted execution: our assumption is that this analysis is supplied with correct (*i.e.* with 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 are not only 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 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 is

```
                          sum2(A,B):-        {A/g,B/f}
           {A/g,B/f}          A = [],        {A/g,B/f}
           {A/g,B/f}          B = 0.         {A/g,B/g}
                          sum2(A,B):-        {A/g,B/f,C/f,D/f,E/f}
 {A/g,B/f,C/f,D/f,E/f}        A = [C|D],     {A/g,B/f,C/g,D/g,E/f}
 {A/g,B/f,C/g,D/g,E/f}        sum2(D,E),     {A/g,B/f,C/g,D/g,E/g}
 {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 because we are being selective in the examples we present) 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

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

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

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

(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.

The set of user-defined predicates which may fail (*i.e.* the 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 $\widetilde{U}$. We make use of the subsumption of mode-annotations (Def. 3.8.2) to define the subsumption between mode-annotated subgoals: $\widetilde{S} = \boxed{\theta_S\, p(y_1,\ldots,y_n)\,\theta'_S}$ subsumes $\widetilde{U} = \boxed{\theta_U\, p(z_1,\ldots,z_n)\,\theta'_U}$, denoted by $\widetilde{S} \sqsupseteq^S \widetilde{U}$, if $\theta_S \sqsupseteq^\theta \theta_U$ and $\theta'_S \sqsupseteq^\theta \theta'_U$.

A mode-annotated subgoal $\widetilde{S}$ is a user-defined test if it subsumes an element of τ, that is, 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, *user-test*$(\widetilde{S}, \tau)$, if there is an element $\widetilde{U}$ of τ such that $\widetilde{S} \sqsupseteq^S \widetilde{U}$.

4.3.7 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. 4.3.1 and 4.3.3, 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 is also clearly finite, given a finite set τ 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 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 may not have been considered relevant.

The source of this imprecision is the third case of Def. 4.3.3, and inaccurate mode-annotations when unknown variables are considered to be fixed. The other cases overlap with those cases of Def 4.3.1, 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. 4.3.3 causes $\widetilde{S}$ to be considered t-relevant, a mistake might have been introduced.

4.3.8 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 may then be considered as part of the argument slice, compromising the extraction of the techniques of 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 to 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.

4.4 Clause-Annotation of Mode-Annotated Argument Slices

Different argument slices of a procedure may share variables. 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 their contents are potentially 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 appear, 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 intended to be the same in distinct argument slices be explicitly linked. A clause- and mode-annotated clause is of the form

$$H \text{ :- } \theta_0 \gamma_0' \ \gamma_1 \theta_1 \ S_1 \theta_1' \gamma_1', \ldots, \gamma_n \theta_n \ S_n \theta_n' \gamma_n'.$$

where γ_i are *required* annotations, γ_j' are *offer* annotations and θ_j and θ_i' are mode-annotations.

4.4.1 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 any 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 $p(x)$:- $\theta_0, \widetilde{S}_1, \ldots, \widetilde{S}_n$ 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: given a mode-annotated argument slice $\widetilde{P}$, comprised of mode-annotated clauses $\widetilde{C}_1, \ldots, \widetilde{C}_n$, its *clause-annotated version*, $\widehat{\widetilde{P}}$, is comprised of the clause-annotated version of each mode-annotated clause, $\widehat{\widetilde{C}}_1, \ldots, \widehat{\widetilde{C}}_n$.

The clause-annotated version of a mode-annotated clause consists of the clause-annotated version of each subgoal, plus an initial *offer* annotation: given $\widetilde{C}$, 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 $\neg \mathit{change}(x, \theta_0, \theta'_n), 1 \leq i \leq n$; 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

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

Given $\widetilde{S}_i$ its *clause-annotated version* $\widehat{\widetilde{S}}_i$ 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 V_i and W_i are defined as follows:

1. $y_l \in V_i, \neg \mathit{change}(y_l, \theta_p, \theta'_p), y_l \notin V_j, y_l \notin W_k, 0 < j < i, 0 \leq k < i, 1 \leq p \leq n;$

[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.

2. $y_l \in W_i, \neg \mathit{fixed}(y_l, \theta_i, \theta_i'), y_l \notin W_j, 0 \leq j < i, y_l \notin V_k, 0 < k \leq i$.

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

Example: The clause- and mode-annotated argument slices of $sum/2$ are (for the sake of brevity the clause-annotations with empty sets are omitted):

```
                                      sum(A):-        {A/g}              ⟨⟨offer({A})⟩⟩
                         {A/g}          A = [].       {A/g}
                                      sum(A):-        {A/g,C/f,D/f}      ⟨⟨offer({A})⟩⟩
                 {A/g,C/f,D/f}          A = [C|D],    {A/g,C/g,D/g}      ⟨⟨offer({C,D})⟩⟩
                 {A/g,C/g,D/g}          sum(D).       {A/g,C/g,D/g}

                                      sum(B):-        {B/f}
                         {B/f}          B = 0,        {B/g}              ⟨⟨offer({B})⟩⟩
                                      sum(B):-        {B/f,C/f,E/f}
                 {B/f,C/g,E/f}          sum(E),       {B/f,C/g,E/g}      ⟨⟨offer({E})⟩⟩
⟨⟨required({C})⟩⟩  {B/f,C/g,E/g}          B is E + C,   {B/g,C/g,E/g}      ⟨⟨offer({B})⟩⟩
```

The definitions above provide the basis for the algorithm *ClauseAnnotate* shown below:

Algorithm 4.4.1 *ClauseAnnotate*

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

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

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

1 **begin**
2   **for** $i := 1$ **to** n **do**
3   **if** $\widetilde{C}_i = p(x)\,\theta$
4   **then** $\widehat{\widetilde{C}}_i := p(x)\,\theta\,\langle\!\langle \mathit{offer}(\{x\})\rangle\!\rangle$
5   **else if** $\widetilde{C}_i = p(x)$ `:-` $\theta_0\ \widetilde{S}_1, \ldots, \widetilde{S}_m$ **then**
6   **begin**
7   **if** $\neg change(x, \theta_0, \theta'_m)$ **then** $W_0 = \{x\}$ **else** $W_0 = \emptyset$
8   **for** $j := 1$ **to** m **do**
9   **begin**
10           $V_j := \emptyset, W_j := \emptyset$
11   **if** $\widetilde{S}_j = \theta_j\, q(\ldots, y_1, \ldots, y_p, \ldots)\, \theta'_j$ **then**
12   **begin**
13   **for** $k := 1$ **to** p **do**
14   **if** $\neg \mathit{fixed}(y_k, \theta_j, \theta'_j)$
15   **then** $W_j := W_j \cup \{y_k\}$
16   **else if** $\forall l, 1 \leq l \leq m, \neg change(y_k, \theta_l, \theta'_l)$
17   **then** $V_j := V_j \cup \{y_k\}$
18   **for** $l := 1$ **to** $j - 1$ **do**
19   **begin**
20               $V_j := V_j - (V_j \cap V_l), W_j := W_j - (W_j \cap V_l)$
21   **end**
22   **for** $l := 0$ **to** $j - 1$ **do**
23   **begin**
24               $V_j := V_j - (V_j \cap W_l), W_j := W_j - (W_j \cap W_l)$
25   **end**
26             $W_j := W_j - (W_j \cap V_j)$
27             $\widehat{\widetilde{S}}_j := \langle\!\langle \mathit{required}(V_j)\rangle\!\rangle\ \theta_j\, q(\ldots, y_1, \ldots, y_p, \ldots)\, \theta'_j\ \langle\!\langle \mathit{offer}(W_j)\rangle\!\rangle$
28   **end**
29   **end**
30         $\widehat{\widetilde{C}} := p(x)$ `:-`$\theta_0\ \langle\!\langle \mathit{offer}(W_0)\rangle\!\rangle\ \widehat{\widetilde{S}}_1, \ldots, \widehat{\widetilde{S}}_m$
31   **end**
32 **end**

Given a mode-annotated argument slice $\widetilde{P}$, this algorithm obtains a clause-annotated version $\widehat{\widetilde{P}}$ of it. Line 4 of algorithm *ClauseAnnotate* deals with facts; clauses with non-empty bodies are dealt with in lines 5–29. The loop of lines 9–29 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 13–17). 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 18–25) and intersections between W_j and V_j (line 27). One should notice the different lower limit of the loops in 18–21 and 22–25: this is because there is a W_0 set, but not a V_0.

4.5 Formalising Extracted Techniques

The clause- and mode-annotated argument slices obtained previously 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 "$\sim$" 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.

Two clause-annotated argument slices must share variables, by means of *required* and *offer* annotations, in order to be considered a technique or part of a technique. The clause-annotated argument slice $\widehat{P}_i = \widehat{C}_{[1,i]}, \ldots, \widehat{C}_{[n,i]}$ *requires* $\widehat{P}_j = \widehat{C}_{[1,j]}, \ldots, \widehat{C}_{[n,j]}$, denoted by *requires*$(\widehat{P}_i, \widehat{P}_j)$, if there is a clause $\widehat{C}_{[k,i]}$ in $\widehat{P}_i$ of the form

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

requesting a variable offered by a clause $\widehat{C}_{[k,j]}$ in $\widehat{P}_j$, of the form

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

or

$$\widehat{C}_{[k,j]} = H_{[k,j]}\, \theta\, \langle\!\langle \mathit{offer}(W) \rangle\!\rangle$$

and their sets V and W are such that $V \cap W \neq \emptyset$. The symbols $\vec{G}_i$ stand for possibly empty sequences of subgoals. There are 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.

If we represent the clause-annotated argument slices of a procedure as a sequence (Def. E.2.1) of the form $\langle \widehat{P}_1, \ldots, \widehat{P}_n \rangle$ then we can define a programming technique as the shortest subsequence T of clause-annotated argument slices such that if $\widehat{P}_i$ belongs to T and $requires(\widehat{P}_i, \widehat{P}_j)$ then $\widehat{P}_j$ also belongs to T. We employ sequences to represent the argument slices and techniques of a procedure because the relative ordering of the argument slices comprising a technique is important and must be consistent with their original ordering after their extraction: the ordering of the subgoals of a technique is depicted by the relative ordering of its argument slices.

Example: If we denote the first clause- and mode-annotated argument slice of procedure *sum/2* shown in the previous example by $\widehat{\widehat{P}}_1$ and the second one by $\widehat{\widehat{P}}_2$, we have the following techniques:

- $\mathsf{T}_1 = \langle \widehat{\widehat{P}}_1 \rangle$, a technique to decompose a list until the empty list is found;
- $\mathsf{T}_2 = \langle \widehat{\widehat{P}}_1, \widehat{\widehat{P}}_2 \rangle$, a technique to sum elements supplied by argument slice $\widehat{\widehat{P}}_1$, *i.e.*, technique T_1

We describe below an algorithm to partition a sequence of clause-annotated argument slices into a set of programming techniques:

Algorithm 4.5.1 *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 a subsequence of arg. slices in S with which each $\widehat{P}_i$ shares values, then obtain arg. slices sharing values with the components of this subseq.

begin
  for $i := 1$ **to** n **do**
  begin
    $\mathsf{T}_i := \langle \widehat{P}_i \rangle$
    for $j := 1$ **to** n **do**
    begin
      let $\mathsf{T}_i = \langle \widehat{P}_{[i,1]}, \ldots, \widehat{P}_{[i,p_1]} \rangle$
      for $k := 1$ **to** p_i **do**
      begin
        if $requires(\widehat{P}_{[i,k]}, \widehat{P}_j)$ **then**
        begin
          $insertion(\widehat{P}_j, \mathsf{S}, \mathsf{T}_i, \mathsf{T}'_i)$
          $\mathsf{T}_i := \mathsf{T}'_i$
        end
      end
    end
    $\mathcal{T} := \mathcal{T} \cup \mathsf{T}_i$
  end
end

A programming technique is a sub-sequence of the original sequence of all clause-annotated argument slices obtained at the clause annotation stage, such that the relative ordering of the slices is maintained. The *SeparateTechniques* 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 algorithm initially sets the sub-sequence T_i to the clause-annotated argument slice $\widehat{P}_i$ (line 4). It then inserts (Def. E.3.2) all those argument slices $\widehat{P}_j$ from which the current components of T_i require values (relation *requires*) (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. 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 $\mathcal{T}$ is devised by the repeated insertion of argument slices (Def. E.3.2) preserving the same relative ordering as in the initial complete sequence S provided by the argument slicing. 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.

4.6 Program Slicing and Argument Slicing

In Subsection 2.11.1 we mentioned the connection between our method to decompose Prolog procedures into their programming techniques and the work within the procedural paradigm community on program slicing [Wei82, Wei84, RW89, GL91, JR94, RHSR94]. In this section we compare these approaches more carefully, providing more details on program slicing and its similarities with our proposed argument slicing method, as well as its distinctions, advantages and drawbacks.

Program slicing is a decomposition method aimed at procedural code. It employs data flow and control flow information to restrict a given program to subsequences of its commands which are relevant to some criterion. Weiser [Wei82, Wei84] introduced the notion of program slices using the criterion $\langle i, V \rangle$, where i is a statement and V is a set of variables; a program slice consists of all those commands up to line i relevant to the variables V. His method employs the notion of sets of variables whose values are being defined or being referred at a particular statement n, denoted respectively by DEF(n) and REF(n). For instance, a command "`A = A + C`" is such that its set DEF is $\{\mathtt{A}\}$ and its set REF is $\{\mathtt{A}, \mathtt{C}\}$.

In order to obtain the program slice of a given program on a slicing criterion, an algorithm is proposed which checks for relationships between the DEF'ed and REF'ed variables of each statement. The algorithm works by analysing the influence of those DEF'ed variables in the command for the variables V of the criterion; secondary relations and branching commands (if-then-else) are also accounted for.

A programming technique, that is, a sequence of argument slices sharing variables can be seen as a program slice. Our proposed method to decompose a program in its argument slices and programming techniques can be understood as a means to obtain the logic programming equivalent of the program slices. Our criterion is an argument position in the head goal of a procedure: every clause is analysed in its entirety, each subgoal being tested for its relevance to the variable in the argument position being considered. Each clause will bear some form of contribution, in the form of subgoals, to the argument slice and programming technique under consideration.

Both program and argument slicing employ data flow and control flow information to analyse the dependency of variables. The dependency analysis in both methods employs information on the sets of variables whose values are defined or simply referred to in each command or subgoal.

A fundamental distinction arises from the disparate paradigms the methods are aimed at. Procedural languages do not offer multiple usages for their commands as logic programming languages do. The sets of DEF'ed and REF'ed variables are easy to obtain in procedural languages: the unique semantics of their commands provides this information straightforwardly. There is no need to execute a program in a procedural programming language, since the meaning of each command can be obtained from its syntax alone.

However, for logic programs, where the same construct may give rise to different procedural readings, the meaning of a subgoal is not so easy to obtain. The sets of variables whose values are defined or referred to in each subgoal are obtained by means of the *change* and *fixed* definitions. We circumvent the problem of the non-uniqueness of the procedural interpretation of a subgoal by carrying out our analysis with respect to a specific usage of the subgoal, as represented by its mode-annotations.

Our argument slicing and techniques detection method is closely related to program slicing: both proposals have the same purpose, *i.e.* identifying portions of a program sharing some kind of relationship with respect to a criterion. Our approaches do differ though, in which we address different programming paradigms: those techniques employed in each method reflect the need to tackle distinct features of each paradigm.

4.7 Conclusions

A method of extracting the programming techniques of Prolog programs has been presented here. Programming techniques are *dynamic* entities characterised by the syntax of the program and how it has been used. They are extracted with respect to the use of the procedure, as represented in its mode-annotated version. The method employs the mode-annotations 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.

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 $\rho_{\tilde{C}}$ of $\triangleleft$-pairs is prepared and then each non-recursive subgoal is checked for relationships between its variables and the argument slice i being built. The slicing of mode-annotated recursive subgoals is simpler: 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 in argument slice i, depending on the properties its variables possess and the set $\rho_{\tilde{C}}$.

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 during the mode-annotation. Our method employs this information and will be affected if inaccuracies are present.

The extracted programming techniques are to be stored in the knowledge base of our techniques-based environment, the library of programming techniques. Some examples of procedures and their extracted techniques are shown in Appendix A. A manner of organising the extracted programming techniques in the library is depicted in the next chapter.

4.8 Summary

In this chapter we have

- presented a method for partitioning a mode-annotated procedure into the contributions of each argument position; this method is called the *argument slicing* of a procedure, and the contributions are called the *argument slices* of a procedure.
- introduced a means to represent the sharing of variables across distinct argument slices: the *clause-annotations*, place markers for the shared variables; an automatic way to obtain the clause-annotations has also been described.
- proposed a way to formalise the techniques of a mode-annotated procedure in terms of its clause- and mode-annotated argument slices.

Chapter 5

Organising Prolog Programming Techniques

5.1 Introduction

In this chapter we propose a way to organise the programming knowledge captured by our formalisation of programming techniques described in the previous chapter. The suggested organisation aims at providing a convenient form to present the programming techniques to other applications of our techniques-based software development environment, and human users, supporting the navigation in the programming knowledge-base.

In the previous chapter we described a method for extracting the programming techniques of Prolog programs. These programming techniques are to be stored in the *library* of programming techniques, the knowledge-base of our techniques-based software development environment described in Section 1.4. In this chapter we propose a way to organise this repository of programming techniques employing a hierarchical scheme; we also describe the automatic creation and upgrading of this library using those components obtained via the extraction method.

5.2 Organising Prolog Programming Techniques

The proposed organisation has to serve two distinct purposes: it must allow other applications to employ its contents and it must also enable human users to conveniently browse through the library. The following management services are also expected to be available:

1. Efficient insertion and deletion of techniques and argument slices.
2. Automatic definition of new techniques via the redefinition (reimplementation) of existing argument slices.
3. Navigation through the library.
4. Defining more restricted libraries by posing constraints ("filters") on the existing library.

In order not to overburden its human users, and since there is a large amount of information to be displayed, any visualisation facilities must be economical, suppressing details unless they are explicitly required to be shown. It is also highly desirable that, to save space, identical components should not be replicated.

The reuse of existing components in the library to define new techniques can be facilitated if the *abstractions* of each argument slice are also stored. An abstraction is a finite sequence of program transformations in which specific design decisions of an argument slice are gradually concealed (abstracted) until a most generic construct is obtained (Section 5.7). Another advantage of keeping the abstractions of argument slices as part of the library is that seemingly disparate constructs can be found to be related by sharing more abstract forms of their components. Our proposal thus incorporates the abstract forms of argument slices to the library of components.

In the following sections we describe our proposal and explain how it meets the requirements listed above; we also describe how each sub-component of the library is organised, how they are related to each other and how the library is updated. We do not claim that our proposed organisation is the most efficient and adequate manner to store and manage the repository of programming knowledge, but it meets the requirements laid out above.

We propose that the library be comprised of three distinct entities:

1. a list $\mathcal{T}$ containing all the available techniques of the library and the argument slice(s) comprising each one of them (its details are explained in subsection 5.3);

2. a list $\mathcal{S}$ containing all the available clause- and mode-annotated argument slices (details in subsection 5.4);

3. a tree $\mathcal{H}$ containing the argument slices of $\mathcal{S}$ divided into *input* and *output* argument slices, and organised hierarchically according to their abstraction(s) (details in subsection 5.5).

As mentioned above, the abstractions of argument slices can be very useful and are thus stored in our library. The same argument slice may appear in more than one abstraction, and hence we suggest that, in order to save space, a unique identifier be assigned to each argument slice of the library. The actual content of each clause- and mode-annotated argument slices is stored in $\mathcal{S}$, together with its unique identifier. It is necessary that an identifier be associated to an argument slice so as to allow its retrieval.

5.3 The List $\mathcal{T}$ of Techniques

The extraction method supplies us with those techniques employed in a given procedure. A technique T is formalised as a sequence of clause- and mode-annotated argument slices sharing variables, $\mathsf{T} = \langle\, \hat{\hat{P}}_1, \ldots, \hat{\hat{P}}_n\rangle$, and it is stored in $\mathcal{T}$ as the pair

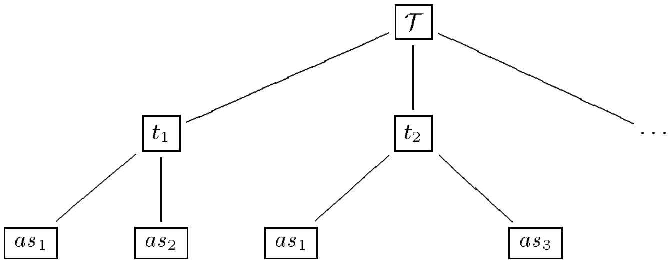
$$t, \langle as_1, \ldots, as_n\rangle$$

where t is a unique identifier and each as_i is the identifier of argument slice $\hat{\hat{P}}_i$, stored in $\mathcal{S}$.

The identifier t provides the user with an economical means to retrieve the technique. The insertion of a new technique is carried out by firstly inserting its argument slices $\hat{\hat{P}}_i$ in the list $\mathcal{S}$ and getting hold of their identifiers as_i (see subsection 5.4). Before a technique is inserted, a check must be made to ensure that it is not already found in $\mathcal{T}$. This check consists of traversing $\mathcal{T}$ and testing if the sequence of identifiers $\langle as_1, \ldots, as_n\rangle$ is not found. If the sequence of identifiers is not already in $\mathcal{T}$ then an identifier t is automatically generated and assigned to that technique. Deletion is straightforward: upon the user's request, a technique t can be efficiently removed from $\mathcal{T}$.

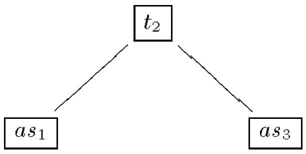
A repertoire of facilities for viewing $\mathcal{T}$ and its elements has been implemented. The user is offered the following set of commands:

1. *show_techniques*: the techniques in $\mathcal{T}$ are shown as a tree of the form



   This format permits the visualisation of the library of techniques as a whole, showing only the identifiers of its components. A node can be further examined upon the user's request.

2. *show_technique*(t): only the technique t is shown, also as a tree: the root node is t and its leaf nodes are its argument slices. Given the diagram above, the command *show_technique*(t_2) yields



   This command provides a more localised view of the library, focusing on a single technique and its argument slices.

3. *show_abs_techniques*: shows all the techniques of $\mathcal{T}$ but their argument slices are shown in their most abstract form, obtained from $\mathcal{H}$. This command provides the users with an overall view of the library from an abstract perspective in which distinct concrete argument slices are replaced by a common abstract form.

4. *show_abs_technique*(t): shows technique t with the most abstract version of each of its argument slices.

The list $\mathcal{T}$ will be supplied as the available techniques in the library. Sublists $\mathcal{T}', \mathcal{T}''$, etc., can be derived from $\mathcal{T}$ by posing constraints on the structure of its components. More restricted collections of techniques can then be defined to meet special needs. For instance, a Prolog techniques editor aimed at novices should only offer simple singly-recursive techniques, manipulating lists and arithmetic. This sub-collection can be obtained by examining the structure of the argument slices of each technique, leaving out those techniques which do not comply with the restrictions.

5.4 The List $\mathcal{S}$ of Argument Slices

Clause- and mode-annotated argument slices are stored in the list $\mathcal{S}$ as the pair

$$as,\ \hat{\hat{P}}_i$$

where *as* is a unique identifier and $\hat{\hat{P}}_i$ is the actual clause- and mode-annotated argument slice. The identifier *as* provides the user with a means to retrieve the argument slice and also saves space by having different ocurrences of the same construct represented by multiple references to its identifier.

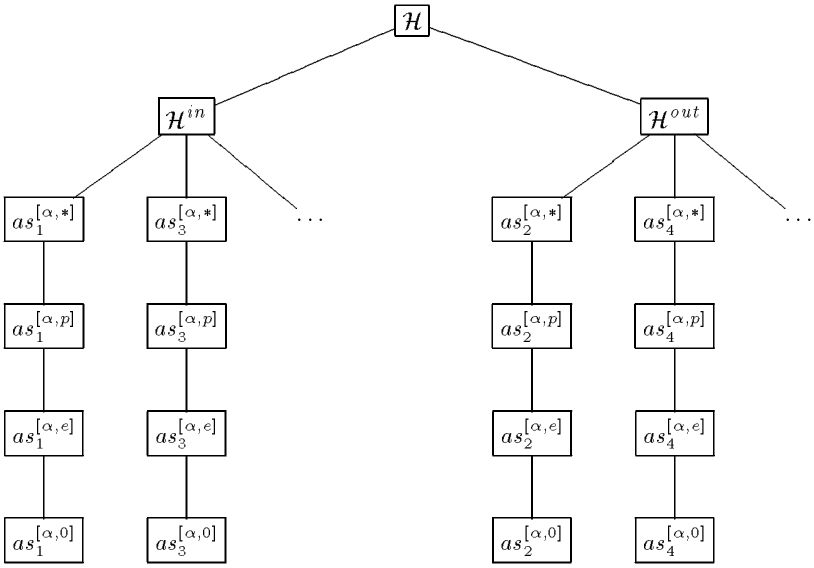
The insertion of a clause- and mode-annotated argument slice in $\mathcal{S}$ is carried out by firstly checking if the component is not already found in $\mathcal{S}$. This consists of testing whether the new component is syntactically identical, modulo variable names, to an existing component of $\mathcal{S}$. If it is not already in the list, it is inserted and an identifier is automatically assigned to it. The deletion envolves a more elaborate procedure: since argument slices are refered to by techniques and by entries in the hierarchy $\mathcal{H}$, their deletion should be followed by the removal of those techniques which employed them, and the appropriate update in $\mathcal{H}$.

The following commands for viewing $\mathcal{S}$ are available:

1. *arg_slice*(*as*): shows the content of the clause- and mode-annotated argument slice whose identifier is *as*.
2. *abstractions*(*as*): shows the abstract forms of argument slice *as*. This command produces a tree with *as* as its leaves and its root node the unique most abstract form of *as*.

5.5 The Hierarchy $\mathcal{H}$ of Argument Slices

The $\mathcal{H}$ component offers an alternative way to view the argument slices of the library: they are shown as nodes of a tree initially divided as *input* and *output* argument slices according to the changes in the instantiation status of the head variable. If the variable does not remain fixed, then the argument slice is an output slice; if the variable does not change, then it is an input argument slice (more details in Section 5.5.2 below). Graphically, $\mathcal{H}$ is of the form



where $\mathcal{H}^{in}$ and $\mathcal{H}^{out}$ are, respectively, the sub-trees of input and output argument slices. Nodes $as_i^{[\alpha,*]}$ contain the most abstract form of argument slice as_i: its particular design decisions have been concealed by the appropriate replacement of its specific subgoals by more abstract *relation* descriptors. In Section 5.7 we explain how the most abstract version of an argument slice is obtained.

Node $as_i^{[\alpha,p]}$ contain the *procedural abstraction* of an argument slice: the subgoals of the argument slice are associated with special descriptors stating the computations taking place at that point. In Section 5.7 we explain how the procedural abstraction of an argument slice can be obtained.

Node $as_i^{[\alpha,e]}$ contains the *explicit form* (e-form) of an argument slice, in which complex computations are split into simpler, equivalent sequences of subgoals, rendering the design decisions clearer; in Section 5.7.6 we describe how the e-form of a mode-annotated argument slice is obtained. When the initial argument slice does not have any complex computations then its e-form is syntactically equivalent to itself and in this case there is no e-form node in the subtree.

The leaf node $as_i^{[\alpha,0]}$ is a clause- and mode-annotated argument slice with the predicate symbol of its head goals and recursive subgoals replaced by the abstract symbol $\mathcal{P}$, described in Section 5.5.1 below and formally depicted in Section B.1.

The four layers introduced above are strategic points in which the design decisions of the argument slice are gradually concealed. They are strategic in the sense that they provide coherent new manners to view an argument slice: the explicit form breaks complex computations into equivalent sequences of simpler subgoals; the procedural abstraction provides a procedural description of the programming practices in the argument slice; and the most abstract form shows an argument slice as simple generic relations between variables. The procedural abstraction and most abstract forms of the argument slices, as we shall see below, may help users finding connections between seemingly disparate constructs.

Example: The first clause- and mode-annotated argument slice of *sum*/2 defines the hierarchy subtree below. The computations of its initial form are not complex and hence its has no distinct e-form

	$\mathcal{P}$(A):-	{A/g}	⟪*offer*({A})⟫
{A/g}	*relation*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,C/f,D/f}	⟪*offer*({A})⟫
{A/g,C/f,D/f}	*relation*({A,C,D}),	{A/g,C/g,D/g}	⟪*offer*({C,D})⟫
{A/g,C/g,D/g}	$\mathcal{P}$(D).	{A/g,C/g,D/g}	

	$\mathcal{P}$(A):-	{A/g}	⟪*offer*({A})⟫
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,C/f,D/f}	⟪*offer*({A})⟫
{A/g,C/f,D/f}	*decomp*(A,{C,D}),	{A/g,C/g,D/g}	⟪*offer*({C,D})⟫
{A/g,C/g,D/g}	$\mathcal{P}$(D).	{A/g,C/g,D/g}	

	$\mathcal{P}$(A):-	{A/g}	⟪*offer*({A})⟫
{A/g}	A = [].	{A/g}	
	$\mathcal{P}$(A):-	{A/g,C/f,D/f}	⟪*offer*({A})⟫
{A/g,C/f,D/f}	A = [C|D],	{A/g,C/g,D/g}	⟪*offer*({C,D})⟫
{A/g,C/g,D/g}	$\mathcal{P}$(D).	{A/g,C/g,D/g}	

The top-most node shows the most abstract form of the argument slice: its particular design decisions are abstractly replaced by "*relation*" subgoals. The middle node shows the procedural abstraction of the argument slice: it shows a generic "*test*" label describing the computation taking place in the only subgoal of the first clause and a "*decomp*" stating that a data structure decomposition is taking place in the first subgoal of the second clause; recursive subgoals are not changed. The bottom node shows the original clause- and mode-annotated argument slice with its predicate symbol replaced by $\mathcal{P}$.

There are a number of intermediate stages between the four levels of abstraction of an argument slice. The different orderings in which the clauses and subgoals are considered for abstraction may yield a very large search-space, depending on the number of clauses and subgoals. These intermediate representations may, however, be useful and we offer a way to obtain them, at the user's request. This is explained in more detail in Section 5.7.

Identical clause- and mode-annotated argument slices will share the same identifier which works as a pointer to the actual component stored in $\mathcal{S}$. The following commands are offered:

1. *show_hierarchy*: shows the whole tree structure of $\mathcal{H}$.
2. *show_input_hierarchy*: shows the sub-tree rooted in $\mathcal{H}^{in}$.
3. *show_output_hierarchy*: shows the sub-tree rooted in $\mathcal{H}^{out}$.

Upon a user request the content of each node can be shown, pretty-printed in another window.

The input and output clause- and mode-annotated argument slices are organised as two disjunct sets of hierarchies. Each hierarchy is built by systematically abstracting the design decisions of an extracted argument slice and obtaining less specific constructs, including a most abstract component. The non-leaf nodes of the hierarchy are abstract versions of those elements below it. The abstraction process is carried out through rewriting rules, as explained in Section 5.7.

5.5.1 Abstraction of Predicate Symbols

The clause- and mode-annotated argument slices obtained in the extraction method maintain the predicate name of the procedure they originate from. This is a minor issue, since predicate symbols are not important in the characterisation of a programming technique, and can safely be ignored when analysing and comparing them. However, since the argument slices and programming techniques will also be examined by human users, it is appropriate to suppress irrelevant details thus providing a cleaner picture of the components.

We propose that the predicate symbol p of each argument slice obtained in the extraction method be consistently replaced by the symbol $\mathcal{P}$. For instance, the first clause- and mode-annotated argument slice of procedure $sum/2$ shown above will be rewritten as

```
                    𝒫(A):-          {A/g}           ⟨⟨offer({A})⟩⟩
          {A/g}        A = [].      {A/g}
                    𝒫(A):-          {A/g,C/f,D/f}   ⟨⟨offer({A})⟩⟩
  {A/g,C/f,D/f}        A = [C|D],   {A/g,C/g,D/g}   ⟨⟨offer({C,D})⟩⟩
  {A/g,C/g,D/g}        𝒫(D).        {A/g,C/g,D/g}
```

In Section B.1 we propose a simple declarative way to relate a clause- and mode-annotated argument slice with a more abstract version in which the head goal predicate symbol and its recursive calls are consistently replaced by $\mathcal{P}$.

5.5.2 Input and Output Argument Slices

Our proposed organisation scheme views clause- and mode-annotated argument slices from a functional perspective, distinguishing between those argument slices in which values are initially supplied and manipulated as the computation proceeds and those in which values are obtained at the end of the computation. For instance, in the following argument slices of procedure $sum/2$

```
                                  sum(A):-          {A/g}           ⟨⟨offer({A})⟩⟩
                        {A/g}       A = [].         {A/g}
                                  sum(A):-          {A/g,C/f,D/f}   ⟨⟨offer({A})⟩⟩
                {A/g,C/f,D/f}       A = [C|D],      {A/g,C/g,D/g}   ⟨⟨offer({C,D})⟩⟩
                {A/g,C/g,D/g}       sum(D).         {A/g,C/g,D/g}
```

```
                                  sum(B):-          {B/f}
                        {B/f}       B = 0,          {B/g}           ⟨⟨offer({B})⟩⟩
                                  sum(B):-          {B/f,C/f,E/f}
                {B/f,C/g,E/f}       sum(E),         {B/f,C/g,E/g}   ⟨⟨offer({E})⟩⟩
⟨⟨required({C})⟩⟩ {B/f,C/g,E/g}       B is E + C,     {B/g,C/g,E/g}   ⟨⟨offer({B})⟩⟩
```

The first construct is an input argument slice: the head variable `A` is initially ground and does not change its instantiation status throughout the computations. The second argument slice is responsible for computing the sum and assigning the final result to the head variable: it is an output argument slice.

More formally, given a clause- and mode-annotated argument slice $\hat{\hat{P}}_i$, such that one of its clauses is of the form

$$\begin{array}{cccc} & p(x)\,\texttt{:-} & \theta\,\gamma \\ \gamma_1\,\theta_1 & S_1\,, & \theta'_1\,\gamma'_1 \\ & \vdots & \\ \gamma_n\,\theta_n & S_n\,. & \theta'_n\,\gamma'_n \end{array}$$

where $\neg \mathit{fixed}(x, \theta, \theta'_n)$, then $\hat{\hat{P}}_i$ is an *output* argument slice, that is, the argument slice describes a sequence of computations that might have changed the instantiation of the head goal variable. If $\hat{\hat{P}}_i$ does not fulfil the requirements above then it is an *input* argument slice.

We can formally define *skeletons* and *additions* as introduced by Kirschenbaum and colleagues [SS86, KLS89, Lak89, SK93] in terms of input and output argument slices. Skeletons are programming techniques comprising solely of input argument slices; additions are input or output argument slices that may share variables with an existing programming technique (a skeleton).

5.6 Updating the Library

After the techniques of a procedure are extracted the user decides if they are to be incorporated to the existing library. If the answer is yes then the following process takes place:

1. *abstraction*: the four different abstract forms of each clause- and mode-annotated argument are obtained.

2. *insertion of argument slices*: the four abstract forms of the argument slices are inserted in $\mathcal{S}$.

3. *insertion of techniques*: the identifiers of the initial argument slices supplied in the extraction stage are retrieved and are employed to define the techniques to be inserted in $\mathcal{T}$.

To illustrate the upgrading process, let us suppose the user decides to extract the techniques of the following version of the *append*/3 procedure:

```
append(A,B,C):-
   A = [],
   B = C.
append(A,B,C):-
   A = [D|E],
   C = [D|F],
   append(E,B,F).
```

with respect to query `?- append([1,2,3],[4,5],A)`, providing the following mode-annotated version of *append*/3:

	`append(A,B,C):-`	{A/g,B/g,C/f}
{A/g,B/g,C/f}	`A = [],`	{A/g,B/g,C/f}
{A/g,B/g,C/f}	`B = C.`	{A/g,B/g,C/g}
	`append(A,B,C):-`	{A/g,B/g,C/f,D/f,E/f,F/f}
{A/g,B/g,C/f,D/f,E/f,F/f}	`A = [D|E],`	{A/g,B/g,C/f,D/g,E/g,F/f}
{A/g,B/g,C/f,D/g,E/g,F/f}	`C = [D|F],`	{A/g,B/g,C/i,D/g,E/g,F/f}
{A/g,B/g,C/i,D/g,E/g,F/f}	`append(E,B,F).`	{A/g,B/g,C/g,D/g,E/g,F/g}

and the following initial clause- and mode-annotated argument slices (its predicate symbols `append` have been replaced by $\mathcal{P}$):

$\widehat{\widehat{P}}{}_1^{[\alpha,0]} =$

	$\mathcal{P}$`(A):-`	{A/g}	⟨⟨ *offer*({A}) ⟩⟩
{A/g}	`A = [].`	{A/g}	
	$\mathcal{P}$`(A):-`	{A/g,D/f,E/f}	⟨⟨ *offer*({A}) ⟩⟩
{A/g,D/f,E/f}	`A = [D|E],`	{A/g,D/g,E/g}	⟨⟨ *offer*({D,E}) ⟩⟩
{A/g,D/g,E/g}	$\mathcal{P}$`(E).`	{A/g,D/g,E/g}	

$\widehat{\widehat{P}}{}_2^{[\alpha,0]} =$

	$\mathcal{P}$`(B).`	{B/g}	⟨⟨ *offer*({B}) ⟩⟩
	$\mathcal{P}$`(B):-`	{B/g}	⟨⟨ *offer*({B}) ⟩⟩
{B/g}	$\mathcal{P}$`(B).`	{B/g}	

$\widehat{\widehat{P}}{}_3^{[\alpha,0]} =$

		$\mathcal{P}$`(C):-`	{B/g,C/f}	
⟨⟨ *required*({B}) ⟩⟩	{B/g,C/f}	`B = C.`	{B/g,C/g}	⟨⟨ *offer*({C}) ⟩⟩
		$\mathcal{P}$`(C):-`	{C/f,D/f,F/f}	
⟨⟨ *required*({D}) ⟩⟩	{C/f,D/g,F/f}	`C = [D|F],`	{C/i,D/g,F/f}	⟨⟨ *offer*({C}) ⟩⟩
	{C/i,D/g,F/f}	$\mathcal{P}$`(F).`	{C/g,D/g,F/g}	⟨⟨ *offer*({F}) ⟩⟩

The techniques of procedure *append*/3 are:

1. $\mathsf{T}_1 = \langle\, \widehat{\widehat{P}}{}_1^{[\alpha,0]} \rangle$, a list-decomposition technique;

2. $\mathsf{T}_2 = \langle\, \widehat{\widehat{P}}{}_2^{[\alpha,0]} \rangle$, a technique passing a parameter down the recursive call;

3. $\mathsf{T}_3 = \langle \widehat{\widehat{P}}{}_1^{[\alpha,0]}, \widehat{\widehat{P}}{}_2^{[\alpha,0]}, \widehat{\widehat{P}}{}_3^{[\alpha,0]} \rangle$, a technique to build a list with the elements supplied by the first technique, followed by the value passed down by the second technique.

If the user decides to upgrade the library with the components of *append*/3, then the three other forms (e-form, procedural abstraction and most abstract) of the clause and mode-annotated argument slices are automatically obtained. The abstract forms, as well as the original argument slices, are inserted in $\mathcal{S}$, being assigned a unique identifier of the form as_n, where n is a natural number. The choice of a short identifier name to the argument slices is due to space and formatting constraints when displaying the hierarchy of components in the library as an n-ary tree.

The abstractions are then inserted in $\mathcal{H}$, with identifiers replacing the actual components. In Figure 5.1 below we show the hierarchy $\mathcal{H}$ (`hierarchy`) with its subtrees $\mathcal{H}^{in}$

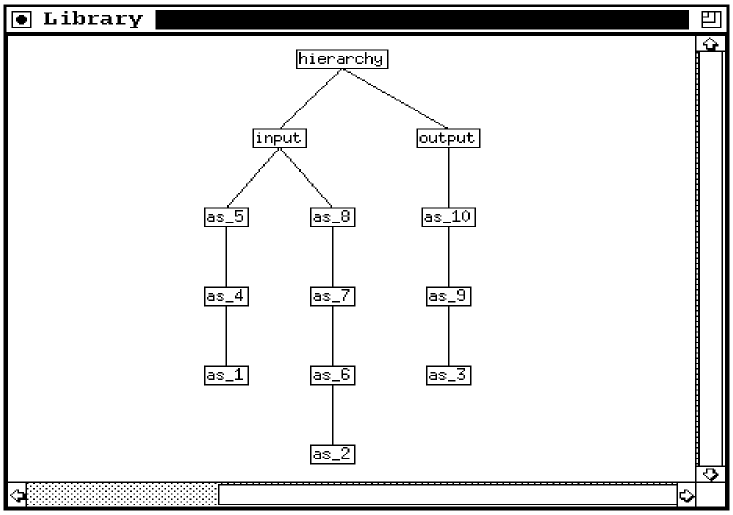


Figure 5.1: Abstract Forms of the Argument Slices of *append*/3

(`input`) and $\mathcal{H}^{out}$ (`output`), after the argument slices are abstracted and inserted in $\mathcal{S}$. The insertion of argument slices assigns as_1 to $\widehat{\widehat{P}}{}_1^{[\alpha,0]}$, as_2 to $\widehat{\widehat{P}}{}_2^{[\alpha,0]}$, and as_3 to $\widehat{\widehat{P}}{}_3^{[\alpha,0]}$. The components of the subtrees above are as_i = $\widehat{\widehat{P}}{}_i^{[\alpha,0]}$, and

as_4 =

	$\mathcal{P}$(A):-	{A/g}	⟨⟨ *offer*({A}) ⟩⟩
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,D/f,E/f}	⟨⟨ *offer*({A}) ⟩⟩
{A/g,D/f,E/f}	*decomp*(A,{D,E}),	{A/g,D/g,E/g}	⟨⟨ *offer*({D,E}) ⟩⟩
{A/g,D/g,E/g}	$\mathcal{P}$(E).	{A/g,D/g,E/g}	

as_5 =

	$\mathcal{P}$(A):-	{A/g}	⟨⟨ *offer*({A}) ⟩⟩
{A/g}	*relation*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,D/f,E/f}	⟨⟨ *offer*({A}) ⟩⟩
{A/g,D/f,E/f}	*relation*({A,D,E}),	{A/g,D/g,E/g}	⟨⟨ *offer*({D,E}) ⟩⟩
{A/g,D/g,E/g}	$\mathcal{P}$(E).	{A/g,D/g,E/g}	

as_6 =

	$\mathcal{P}$(B):-	{B/g}	⟨⟨ *offer*({B}) ⟩⟩
{B/g}	*true*(B).	{B/g}	
	$\mathcal{P}$(B):-	{B/g,G/f}	⟨⟨ *offer*({B}) ⟩⟩
{B/g,G/f}	G = B,	{B/g,G/g}	⟨⟨ *offer*({G}) ⟩⟩
{B/g,G/g}	$\mathcal{P}$(G).	{B/g,G/g}	

as_7 =

		$\mathcal{P}$(B):-	{B/g}	⟨⟨ *offer*({B}) ⟩⟩
	{B/g}	*test*(B).	{B/g}	
		$\mathcal{P}$(B):-	{B/g,G/f}	⟨⟨ *offer*({B}) ⟩⟩
	{B/g,G/f}	*assign*({B},{G}),	{B/g,G/g}	⟨⟨ *offer*({G}) ⟩⟩
	{B/g,G/g}	$\mathcal{P}$(G).	{B/g,G/g}	

as_8 =

		$\mathcal{P}$(B):-	{B/g}	⟨⟨ *offer*({B}) ⟩⟩
	{B/g}	*relation*({B}).	{B/g}	
		$\mathcal{P}$(B):-	{B/g,G/f}	⟨⟨ *offer*({B}) ⟩⟩
	{B/g,G/f}	*relation*({B,G}),	{B/g,G/g}	⟨⟨ *offer*({G}) ⟩⟩
	{B/g,G/g}	$\mathcal{P}$(G).	{B/g,G/g}	

as_9 =

		$\mathcal{P}$(C):-	{B/g,C/f}	
⟨⟨ *required*({B}) ⟩⟩	{B/g,C/f}	*assign*({B},{C}).	{B/g,C/g}	⟨⟨ *offer*({C}) ⟩⟩
		$\mathcal{P}$(C):-	{C/f,D/f,F/f}	
⟨⟨ *required*({D}) ⟩⟩	{C/f,D/g,F/f}	*build*({D,F},C),	{C/i,D/g,F/f}	⟨⟨ *offer*({C}) ⟩⟩
	{C/i,D/g,F/f}	$\mathcal{P}$(F).	{C/g,D/g,F/g}	⟨⟨ *offer*({F}) ⟩⟩

as_10 =

		$\mathcal{P}$(C):-	{B/g,C/f}	
⟨⟨ *required*({B}) ⟩⟩	{B/g,C/f}	*relation*({B,C}).	{B/g,C/g}	⟨⟨ *offer*({C}) ⟩⟩
		$\mathcal{P}$(C):-	{C/f,D/f,F/f}	
⟨⟨ *required*({D}) ⟩⟩	{C/f,D/g,F/f}	*relation*({C,D,F}),	{C/i,D/g,F/f}	⟨⟨ *offer*({C}) ⟩⟩
	{C/i,D/g,F/f}	$\mathcal{P}$(F).	{C/g,D/g,F/g}	⟨⟨ *offer*({F}) ⟩⟩

After $\mathcal{S}$ and $\mathcal{H}$ are updated, then the techniques are inserted, with their argument slices replaced by their identifiers obtained previously. Upon the user's request (command *show_techniques*), the diagram of Figure 5.2 is shown, and the user can examine any

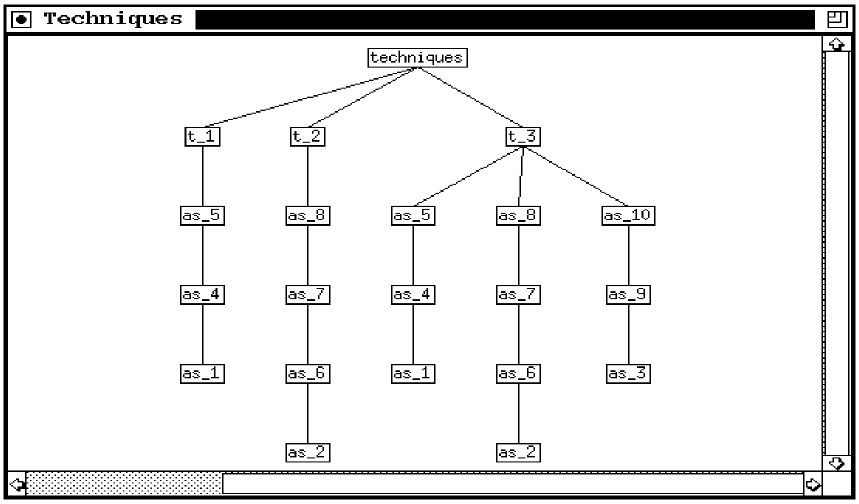


Figure 5.2: Techniques of *append*/3 and their Argument Slices

of the components of the tree.

5.7 Abstracting and Reimplementing Design Decisions

Programming techniques embody many design decisions. Some of these decisions concern the data structures employed, the ordering and the kind of the computations and tests carried out and the commands and structures used to define loops and other control structures. The particular choices of constructs of a program can be given a more abstract characterisation concealing its details while providing an alternative way to view the computations taking place.

We propose a framework within which the design decisions of clause- and mode-annotated argument slices can be gradually abstracted. The abstraction process consists of assigning to each subgoal a special label, a *predicate descriptor*, describing the computations performed at that point. In order to assign a predicate descriptor to a subgoal, its mode-annotations and syntax are taken into account. We provide a set of relations, the *δ-relations*, by means of which any clause- and mode-annotated subgoal (and hence any clause- and mode-annotated argument slice and programming technique) can be abstracted. Relationships between seemingly disparate argument slices and techniques can be automatically discovered when their more abstract versions are examined.

We view the design decisions of a Prolog program as consisting of the syntax together with the usage manner of its subgoals. The manner in which each subgoal has been used is depicted by its mode-annotations. A mode-annotated subgoal can be described in a more abstract manner by a less committed *predicate descriptor* labelling the computations carried out by means of it. The purpose of a descriptor is to conceal particular design decisions, abstractly representing subgoals as operations from a repertoire of basic Prolog programming practices. During the abstraction, each subgoal is assigned a descriptor labelling the computations carried out by means of it. In the reimplementation, the descriptors are replaced by actual Prolog subgoals.

The δ-relations can be employed both to abstract and to customise portions of the programming technique. The framework for automatically abstracting a Prolog programming technique can be used, employing the same δ-relations, for the reimplementation of the abstract constructs. The reimplementation is, however, human-assisted: a human user interactively chooses how to specialise the abstractions, support being given to this task. In Figure 5.3 we can see an illustration of the integrated framework: given a clause- and mode-annotated argument slice $\widehat{\widehat{P}}_i$, its abstract version $\widehat{\widehat{P}}_i^{\alpha}$, whose design decisions are hidden, is used as a template to build a new argument slice $\widehat{\widehat{P}}_i'$.

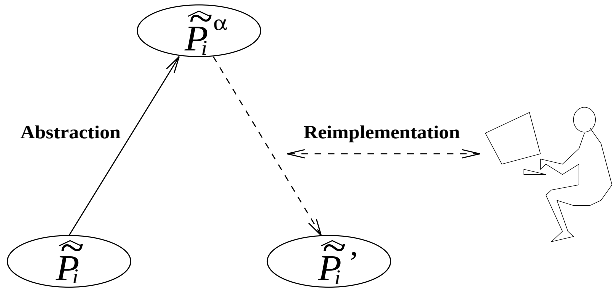


Figure 5.3: Automatic Abstraction and Human-Assisted Reimplementation

The reimplementation of the abstract argument slice $\widehat{\widetilde{P}}_i^{\alpha}$ depends on the participation of a user and support is given at this stage: this is explained in Chapter 6.

5.7.1 Predicate Descriptors

We propose that particular design decisions be assigned generic *descriptors* conveying the meaning of the computation. A descriptor is a predicate meta-variable whose possible values are restricted by its name. Descriptors employ sets of variable symbols as arguments, enabling a range of different arities to be expressed with the same construction. Our repertoire of descriptors and their intended meanings are shown in Table 5.1, where V and W are possibly empty sets of variable symbols, $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_m\}$. These descriptors can be understood as predicate variables with

Descriptor	**Meaning**
$test(V)$	the variables of V are used in a test
$decomp(x, V)$	x is decomposed into the variables of V
$build(V, x)$	the variables of V are used to compose x
$calc(V, x)$	the variables of V are used to calculate x via $is/2$
$assign(V, W)$	the variables in V are used to assign values to those of W
$bind(x, y)$	variables x and y are bound together via `=`
$user\text{-}pred(V)$	the variables V are employed in a user-defined predicate
$relation(V)$	the variables of V are used to define a relation

Table 5.1: Predicate Descriptors and their Meaning

a restricted interpretation. The *test* descriptor represents those subgoals performing tests, such as ¶ and $\diamond$. The *decomp* descriptor stands for a data structure being decomposed via the `=` operator, that is, x `=` $f(V)$. The *build* descriptor represents subgoals with the same syntax as those of *decomp*, the only difference being the instantiation status of variables. A mode-annotated subgoal of the form

{`A/g,B/f,C/f`} `A = [B|C]` {`A/g,B/g,C/g`}

is abstracted as a data structure decomposition (*decomp*), whereas

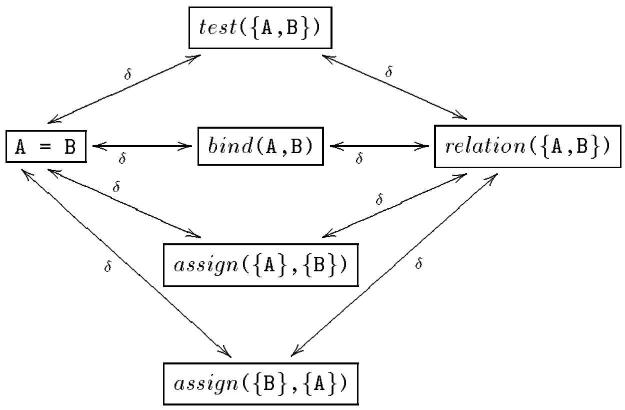
{`A/f,B/g,C/g`} `A = [B|C]` {`A/g,B/g,C/g`}

is abstracted as a data structure being built (*built*). The *calc* descriptor stands for those subgoals employing the *is*/2 system predicate to perform calculations, that is, subgoals of the form x `is` $f(V)$. The *assign* descriptor stands for all other forms of subgoals which change the instantiation status of at least one variable, such as system predicates *read/1* and *get/1*. Subgoals of the form x `=` y where x and y are free before the subgoal (thus remaining free after the subgoal is executed) are assigned the *bind* descriptor. The *user-pred* predicate descriptor abstracts those subgoals performing a call to user-defined procedures; the necessity for it is explained in subsection 5.7.1. These predicate descriptors can be further abstracted by the less committed descriptor *relation*, hiding their procedural meaning and providing the subgoal with a generic and unique declarative meaning.

The suggested repertoire of predicate descriptors provides a detailed account of those practices commonly found in Prolog programs. The argument slices of a program can be procedurally described in terms of its tests, data structure manipulations (via pattern-matching), calculations (via the *is*/2 predicate), assignments, variables binding and other computations by means of auxiliary procedures: these features are widely used and deserve recognition as standard Prolog programming practices. A minimal set of descriptors would contain only *test* and *assign*, since these are the basic actions of any program and all other descriptors can be described in their terms. The *decomp* descriptor, for instance, is a test followed by an assignment; *build* and *calc* are just more elaborate forms of assignments; *bind* is also a special form of assignment.

The *relation* Descriptor

The *relation* predicate descriptor provides a less precise account of the computation carried out by a subgoal, labelling it as a generic relationship being defined between its variables. This neutral descriptor is in accordance with the declarative view of subgoals, promoting the convergence between the many different procedural meanings of a given subgoal described as one of the more specific predicate descriptors. For instance, if we are to abstract the subgoal "`A = B`" solely on the basis of its syntax (or, alternatively, the mode-annotations are inaccurate, with "**?**" associated with `A` and `B` before and after the subgoal) then we would have to consider as if either a *test*, an *assign*, or a *bind* is taking place; all these possibilities are subsumed by the *relation* descriptor. The diagram below depicts the δ-relations between three levels of subgoals: the leftmost component shows the actual "`A = B`" Prolog subgoal, the components in the middle comprise the four procedural readings in terms of our descriptors, and the rightmost component shows a representation of the unique declarative reading of the concrete subgoal, via the *relation* descriptor:



The mode-annotations offer a way to narrow down the set of possible procedural readings of a given subgoal. If the mode-annotations are not available or are inaccurate with tokens "**i**" and "**?**" associated to variables before and after the subgoal, then we can only rely on the syntax and hence we have to consider all possibilities. If only the *relation* predicate descriptor were available the abstraction would be much simpler and there would always be a single possible δ-relation holding.

This, however, would pose severe problems in the reimplementation stage: the users would be left with the infinitely many possible ways to redefine each *relation* and no disciplined manner to help them during this task. The more specific predicate descriptors convey the procedural meanings of subgoals and provide an intermediate level between the generic *relation* and actual Prolog subgoals. They serve as a menu of possible choices for the definition of *relation*: users have the different possibilities distributed among the descriptors bearing more procedural meanings.

Furthermore, by employing more specific predicate descriptors, a procedural account of each subgoal is provided, thus supplying users with updated automatic documentation. Since the abstraction process is fully automated, the different ways to assign a descriptor to a subgoal can be conveniently recorded and pursued separately, at the cost of storage and computational effort.

When a mode-annotated subgoal is replaced by its corresponding mode-annotated descriptor, its particular design details are hidden but its functionality, depicted by its mode-annotations, is maintained. During the reimplementation of an abstract subgoal its mode-annotations work as constraints, ruling out specialisations which are not compatible with the modes of its variables. An abstract subgoal of the form

{... `A/f,B/f` ...} *relation*({`A,B`}) {... `A/f,B/f` ...}

can only be reimplemented[1] as *bind* or *user-pred*: all the remaining descriptors require that the modes of at least one of their variables change.

The *user-pred* Descriptor

Those subgoals invoking user-defined predicates must also be assigned a descriptor conveying the computations taking place. When user-defined predicates are employed, it becomes impossible to state, in general and with certainty, what computations are being carried out because their procedural meaning is hidden in their definition. Accurate mode-annotations reveal whether the contents of variables have changed, and hence whether the user-defined predicate has been used to obtain new values.

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

[1] The *relation* descriptor, as we shall see, can give rise to a sequence of other *relation* descriptors, by having its set of variables split and distributed among the new *relation* descriptors.

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: we assume 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, and which should be assigned the *test* descriptor.

The problem of deciding if a user-defined predicate may fail has been explained in Subsection 4.3.6. User-defined procedures may additionally serve for more than one purpose. For instance, if we have the mode-annotated subgoal

```
{...A/i,B/?...}  p(A,B) {...A/i,B/?...}
```

and the definition of *p/1* as

```
p(X,Y):- Y is X * X.
```

then the mode-annotated subgoal can be assigned either the *test* or *assign* subgoal. If a procedure performs a test using some of its arguments and assigns values to its other arguments then it becomes difficult to choose one descriptor that would correctly describe all the computations performed by that procedure.

We suggest a simple solution to this problem: a special descriptor, *user-pred* should be assigned to those subgoals invoking user-defined predicates. This solution would circumvent the problem of finding out which computations are taking place and simply labels the subgoal as a "call to a user-defined predicate". A *user-pred* is further abstracted as a *relation* and this can be reimplemented as various sorts of predicate descriptors, depending on its mode-annotations.

5.7.2 δ-Relations

We propose a relational operator δ to formalise the abstraction and reimplementation of design decisions of argument slices. A δ-relation is of the form

$$L \xleftrightarrow{\delta} R$$

meaning that L can be abstracted as R, or alternatively, R can be specialised as L. Each δ-relation gives rise to a pair of unidirectional rewriting rules. The above relation, for instance, yields the pair of rewriting rules $L \longrightarrow R$ and $R \longrightarrow L$. A more concise account of the transformations proposed here can be achieved by employing relations, rather than rules. This is a purely stylistic advantage, since they are equivalent.

The δ-relations are defined over clause- and mode-annotated constructs: clauses, sequences of subgoals and single subgoals. The patterns of a δ-relation act as template, and together with other constraints explicitly listed, rule out those constructs which cannot have that abstraction/reimplementation performed on them. The δ-relations between clauses are of the form

$$\boxed{\begin{array}{ccccc} & & H\texttt{:-} & \theta & \gamma \\ \gamma_0 & \theta_0 & S_0 & \theta'_0 & \gamma'_0 \\ & & \vdots & & \\ \gamma_n & \theta_n & S_n & \theta'_n & \gamma'_n \end{array}} \overset{\delta}{\longleftrightarrow} \boxed{\begin{array}{ccccc} & & H^\alpha\texttt{:-} & \theta^\alpha & \gamma^\alpha \\ \gamma_0^\alpha & \theta_0^\alpha & S_0^\alpha & {\theta'}_0^\alpha & {\gamma'}_0^\alpha \\ & & \vdots & & \\ \gamma_m^\alpha & \theta_m^\alpha & S_m^\alpha & {\theta'}_m^\alpha & {\gamma'}_m^\alpha \end{array}}$$

where n and m may be different, that is, there may be δ-relations between clauses with different number of subgoals in their bodies. Any additional requirements to the patterns in either sides of the δ-relation can be listed immediately after the relation; usually these are additional constraints on the relationships between the mode-annotations. We shall enclose the patterns in both sides of the relation within framed boxes to improve their visualisation.

The δ-relations defined over clause- and mode-annotated subgoals are of the form

$$\boxed{\gamma\ \theta\ S\ \theta'\ \gamma'} \overset{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ S^\alpha\ \theta'\ \gamma'}$$

The δ-relations between subgoals are such that they only hold if the clause- and mode-annotations before and after the subgoal in one side are correspondingly equal to those in the other side. This means that subgoals may be abstracted or specialised but their clause- and mode-annotations remain the same.

Those δ-relations between clauses, however, may be such that the clause- and mode-annotations of the subgoals in one side do not correspond to those of the subgoals in the other side. This means that the abstraction or specialisation of a clause may alter the annotations of the subgoals, by inserting or removing variable/token pairs.

5.7.3 Subgoal δ-Relations

When employed to abstract a clause- and mode-annotated subgoal a δ-relation rewrites the subgoal as a predicate descriptor, an informative predicate name conveying the procedural meaning of the subgoal. Every possible combination of predicates and modes has to be addressed.

In some cases, a subgoal can be accurately abstracted as one of the descriptors. These subgoals employ system predicates or operators whose computational meaning is well-defined and known in advance. For instance, those subgoals employing operators `<`, `>`, and so on, are uniquely abstracted as a *test*, despite their mode-annotations. The mode-annotations, in other cases, complement the syntax of the subgoal, ruling out possible meanings. A subgoal of the form "`A = B`" can be either a *test*, an *assign* or a *bind* command: its mode-annotations, if accurate, may provide more information to narrow down the possible abstractions.

The *test* descriptor stands for all subgoals used as tests, that is, those subgoals that may alter the flow of execution of a procedure by causing a failure. For instance, all those system operators $\diamond$ different from `=` and `=..` are necessarily tests, despite their mode-annotations. This can be formalised as the following δ-relation:

$$\boxed{\gamma\ \theta\ x \diamond a\ \theta'\ \gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ test(\{x\})\ \theta'\ \gamma'}$$

where $\diamond \notin \{\texttt{=}, \texttt{=..}\}$. The constant a is an arbitrary construct and is not part of the abstraction. If this relation is used to specify a *test* subgoal, then a must be provided by the user. The *assign* predicate descriptor abstracts those subgoals that may perform changes in the instantiation status of its variables. If the subgoal is of the form $x \diamond a$ and the tokens associated with variable x do not satisfy the *fixed* relationship, we shall consider this subgoal as an assignment, as formalised by the following δ-relation:

$$\boxed{\gamma\ \theta\ x \diamond a\ \theta'\ \gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ assign(x)\ \theta'\ \gamma'}$$

where $\diamond \in \{\texttt{=}, \texttt{=..}\}$ and $\neg fixed(x, \theta, \theta')$. The *decomp* descriptor abstracts those subgoals of the form $x \texttt{=} f(V)$ such that x does not satisfy the *change* relationship, that is,

$$\boxed{\gamma\ \theta\ x \texttt{=} f(y_1, \ldots, y_n)\ \theta'\ \gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ decomp(x, \{y_1, \ldots, y_n\})\ \theta'\ \gamma'}$$

where $\neg change(x, \theta, \theta')$. The same subgoal may be abstracted as the *build* descriptor, if x does not satisfy the *fixed* relationship, that is,

$$\boxed{\gamma\ \theta\ x = f(y_1, \ldots, y_n)\ \theta'\ \gamma'} \overset{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ build(\{y_1, \ldots, y_n\}, x)\ \theta'\ \gamma'}$$

where $\neg fixed(x, \theta, \theta')$.

The predicate descriptors conveying the procedural meaning of the subgoals can be further abstracted as the *relation* descriptor, providing an even less committed account of the design decisions of the program. The *relation* descriptor stands for a generic declarative description of the subgoals: its sets of variables and their mode-annotations preserve the procedural aspect of the subgoal and serve as constraints in future reimplementations. The more specific descriptors all relate to the *relation* descriptor via δ-relations with possible restrictions as to how the variables in each descriptor are grouped together, this depending on the tokens associated with each variable. The *test* descriptor, for instance, relates to *relation* in a straightforward way:

$$\boxed{\gamma\ \theta\ test(W)\ \theta'\ \gamma'} \overset{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ relation(W)\ \theta'\ \gamma'}$$

where $x \in W, \neg change(x, \theta, \theta')$; *test* subgoals should not change the instantiation status of their variables W. As another example, the arguments x and V of *decomp* are merged as W in *relation*:

$$\boxed{\gamma\ \theta\ decomp(x, V)\ \theta'\ \gamma'} \overset{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ relation(W)\ \theta'\ \gamma'}$$

where $W = V \cup \{x\}, \neg change(x, \theta, \theta')$, and $y \in V, \neg fixed(y, \theta, \theta')$; the variables $y \in V$ have their values changed and x remains unchanged. Two consecutive *relation* subgoals are related to a single *relation* subgoal whose set of variables is given by the union of the sets of variables of the consecutive subgoals, that is,

$$\boxed{\begin{array}{l}\langle\!\langle required(V_1)\rangle\!\rangle\ \theta\ relation(W_1)\ \theta_*\langle\!\langle offer(Z_1)\rangle\!\rangle \\ \langle\!\langle required(V_2)\rangle\!\rangle\ \theta_* relation(W_2)\ \theta'\langle\!\langle offer(Z_2)\rangle\!\rangle\end{array}} \overset{\delta}{\longleftrightarrow} \boxed{\langle\!\langle required(V)\rangle\!\rangle\ \theta\ relation(W)\ \theta'\ \langle\!\langle offer(Z)\rangle\!\rangle}$$

where $W = W_1 \cup W_2, V_1 = V \cap W_1, V_2 = V \cap W_2, x \in W, x/T \in \theta, x/T' \in \theta_*, x/T'' \in \theta', T \preceq T' \preceq T''$. The $\preceq$ relation is as described in Section 3.6.2; it poses additional requirements on the mode-annotations of the variables and the way they can be divided into two subsets with their own mode-annotations. This δ-relation merges the *relation* subgoals when used to abstract the subgoals of a clause. Other δ-relations are shown in Appendix B.

5.7.4 E-form δ-Relations

Some Prolog programming practices can be seen as simplifications of more complex sub-sequences of subgoals. The abstraction of these practices would have to consider the whole sub-sequence of subgoals which makes the programming practice explicit. The set of δ relations described here complement the normal form of the programs to be abstracted by further replacing those subgoals carrying out complex computations by equivalent sequences of simpler subgoals. We describe below one of the e-form δ-relations: the rest of them are shown in Section B.2.

When a subgoal performing a data structure decomposition is such that one of the variables in its pattern is not associated with token "**f**", then possibly a test is also being implicitly performed. The δ-relation below replaces the variable y of the data structure pattern which is not associated with "**f**" in θ_{k-1} for a fresh variable z associated with "**f**", and then inserts a subgoal of the form z = y. All the other mode-annotations of the clause have to be altered to accommodate the newly introduced variable z:

$$\boxed{\begin{array}{lll} & H \text{:-} & \theta_0 \;\; \gamma_0 \\ & \vec{S}_1 & \\ \gamma \;\; \theta & x \; = \; f(\vec{V}_1 \; y \; \vec{V}_2) \, , & \theta' \;\; \gamma' \\ & \vec{S}_2 \, . & \end{array}} \quad \stackrel{\delta}{\longleftrightarrow} \quad \boxed{\begin{array}{lll} & H \text{:-} & \theta_0^\alpha \;\; \gamma_0 \\ & \vec{S}_1^\alpha & \\ \gamma^\alpha \;\; \theta^\alpha & x \; = \; f(\vec{V}_1 \; z \; \vec{V}_2) \, , & \theta'^\alpha \;\; \gamma'^\alpha \\ \gamma_* \;\; \theta'^\alpha & y \; = \; z \, , & \theta'^\alpha \;\; \gamma'_* \\ & \vec{S}_2^\alpha \, . & \end{array}}$$

where $y/T \in \theta, T \neq \mathbf{f}$; the other additional constraints on the γ's and θ's are explained in Appendix B.

5.7.5 Sequences, Vectors and Declarative Definitions

In the next section our attention will be drawn to clause- and mode-annotated argument slices and their clauses. In order to refer to these constructs and to parts of them we have employed *sequences*, as in Definition E.2.1. A clause- and mode-annotated argument slice $\hat{\hat{P}}_i$ is denoted by the sequence $\langle \hat{\hat{C}}_1, \ldots, \hat{\hat{C}}_n \rangle$. Each clause- and mode-annotated clause $\hat{\hat{C}}_j$ is denoted by H :- $\theta\gamma$ B where B is a sequence of clause- and mode-annotated subgoals comprising the body of $\hat{\hat{C}}_j$, of the form $\langle \hat{\hat{S}}_1, \ldots, \hat{\hat{S}}_m \rangle$.

To allow for more flexibility and economy in our notation, we employ *vectors* of elements in our sequences: $\vec{C}$ stands for the possibly empty subsequence $\langle \hat{\hat{C}}_0, \ldots, \hat{\hat{C}}_s \rangle$, and $\vec{S}$ stands for the possibly empty subsequence $\langle \hat{\hat{S}}_0, \ldots, \hat{\hat{S}}_s \rangle$. We also employ a vector $\vec{V}$ for a possibly empty sequence $x_0, \ldots, x_n$ of Prolog variables. According to this notation the pattern $\hat{\tilde{P}}_i = \langle \vec{C} \rangle$ describes a clause- and mode-annotated argument slice $\hat{\tilde{P}}_i$ consisting of a possibly empty vector of clause- and mode-annotated clauses. The patterns may become more sophisticated, such as $\hat{\tilde{P}}_i = \langle \vec{C}\, \hat{\tilde{C}} \rangle$ in which a reference to the last clause $\hat{\tilde{C}}$ of $\hat{\tilde{P}}_i$ is made. Further elaborations can be conceived, such as $\hat{\tilde{P}}_i = \langle \vec{C}_1\, \hat{\tilde{C}}\, \vec{C}_2\, \hat{\tilde{C}}'\, \vec{C}_3 \rangle$ describing an argument slice consisting of at least two clause- and mode-annotated clauses $\hat{\tilde{C}}$ and $\hat{\tilde{C}}'$. Similar constructions can be devised to describe clause- and mode-annotated clauses and their subgoals and variables of subgoals.

The vector notation provides us with a clean and economic manner to address specific parts of a mode-annotated program or clause, without having to explicitly refer to their implementational details. For instance, the conjunction $\hat{\tilde{P}}_i = \langle \vec{C}_1 \hat{\tilde{C}} \vec{C}_2 \rangle \ \wedge \ p(\hat{\tilde{C}})$ is true if there is a clause- and mode-annotated clause $\hat{\tilde{C}}$ in $\hat{\tilde{P}}_i$ satisfying predicate p; this construction is such that there are no references whatsoever as to how $\hat{\tilde{P}}_i$ has been represented or implemented nor as to how $\hat{\tilde{P}}_i$ has had its clauses examined until $\hat{\tilde{C}}$, satisfying p, was found.

In the definitions below we have adopted a purely declarative Horn-clause formalism, in which implementational details were deliberately concealed. Sequences and vectors are used instead of specific data structures, thus freeing us from commitments to representations and their explicit manipulation. These declarative definitions are non-deterministic in the sense that some of its parts are not given a systematic procedure but a descriptive solution is proposed instead. However, this notation can safely be understood as that of a Prolog program. More details are found in Subsection E.1.2.

5.7.6 A Framework for Abstracting and Reimplementing Design Decisions

A programming technique is abstracted and reimplemented via the appropriate manipulation of its clause- and mode-annotated argument slices. Programming techniques are sequences of clause- and mode-annotated argument slices: any relationship between programming techniques has to be in terms of their argument slices.

We define here a $\Re$ predicate relating a clause- and mode-annotated argument slice $\hat{\tilde{P}}_i$ to its four abstract forms $\hat{\tilde{P}}_i^{[\alpha,0]}$ ($\mathcal{P}$ replaces the predicate symbol employed in its head goals and recursive subgoals), $\widehat{\widehat{P}}{}_i^{[\alpha,e]}$ (explicit form), $\widehat{\widehat{P}}{}_i^{[\alpha,p]}$ (procedural abstraction), and $\widehat{\widehat{P}}{}_i^{[\alpha,*]}$ (most abstract form). These forms are to be inserted in the library, as explained in Section 5.5. $\Re$ is defined as

$$\begin{array}{rcl} \Re(\widehat{\widehat{P}}_i, \langle \widehat{\widehat{P}}{}_i^{[\alpha,0]}, \widehat{\widehat{P}}{}_i^{[\alpha,e]}, \widehat{\widehat{P}}{}_i^{[\alpha,p]}, \widehat{\widehat{P}}{}_i^{[\alpha,*]}\rangle) & \Longleftarrow & \mathit{pred}(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}{}_i^{[\alpha,0]}, q) \wedge \\ & & \textit{e-form}(\widehat{\widehat{P}}{}_i^{[\alpha,0]}, \widehat{\widehat{P}}{}_i^{[\alpha,e]}) \wedge \\ & & \Re^p(\widehat{\widehat{P}}{}_i^{[\alpha,e]}, \widehat{\widehat{P}}{}_i^{[\alpha,p]}) \wedge \\ & & \Re^*(\widehat{\widehat{P}}{}_i^{[\alpha,p]}, \widehat{\widehat{P}}{}_i^{[\alpha,*]}) \end{array}$$

The *pred*/3 predicate maps an argument slice $\widehat{\widehat{P}}_i$ to its version $\widehat{\widehat{P}}{}_i^{[\alpha,0]}$ in which the predicate symbol q in its head goals and recursive calls were replaced by $\mathcal{P}$; it is formally defined in Section B.1. The *e-form*/2 predicate relates $\widehat{\widehat{P}}{}_i^{[\alpha,0]}$ to its explicit form version $\widehat{\widehat{P}}{}_i^{[\alpha,e]}$. $\Re^p/2$ relates the e-form version of the clause- and mode-annotated argument slice $\widehat{\widehat{P}}{}_i^{[\alpha,e]}$ to its procedural abstraction $\widehat{\widehat{P}}{}_i^{[\alpha,p]}$, in which the subgoals have been replaced by descriptors conveying the computations being performed. Finally $\Re^*/2$ relates the procedural abstraction $\widehat{\widehat{P}}{}_i^{[\alpha,p]}$ to its most abstract form $\widehat{\widehat{P}}{}_i^{[\alpha,*]}$. These auxiliary predicates are described below.

Relation *e-form*

It is common, in the programming practices found in Prolog programs, for a single subgoal to perform complex computations. If these complex subgoals are to be re-engineered, then they should have their intricacies explicitly laid out so that the users can choose which parts they want to reimplement. The *explicit form* (or *e-form*) of an argument slice makes explicit these design decisions, replacing complex computations with an equivalent sequence of simpler subgoals. The abstraction and reimplementation of an argument slice in its e-form addresses design issues which would not have become apparent otherwise.

Given a clause- and mode-annotated argument slice $\widehat{\widehat{P}}_i$, we define its e-form $\widehat{\widehat{P}}{}_i^{[\alpha,e]}$ as being $\widehat{\widehat{P}}_i$ after having the e-form δ-relations (Sections 5.7.4 and B.2) exhaustively applied until there are no more clauses matching them. The *e-form* relation between two clause- and mode-annotated argument slices $\widehat{\widehat{P}}_i$ and $\widehat{\widehat{P}}{}_i^{[\alpha,e]}$ is defined as

$$\textit{e-form}(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_i) \quad \Longleftarrow \quad \forall \widehat{\widehat{C}} (\widehat{\widehat{P}}_i = \langle \vec{C}\ \widehat{\widehat{C}}\ \vec{C'} \rangle \wedge \neg(\widehat{\widehat{C}} \stackrel{\delta}{\longleftrightarrow} \widehat{\widehat{C}}{}^e))$$

$$
\begin{aligned}
e\text{-}form(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_i^{[\alpha,e]}) \quad \Longleftarrow \quad & \widehat{\widehat{P}}_i = \langle \vec{C}\, \widehat{\widehat{C}}\, \vec{C'} \rangle \wedge \widehat{\widehat{P}}'_i = \langle \vec{C}\, \widehat{\widehat{C}}^e\, \vec{C'} \rangle \wedge \\
& (\widehat{\widehat{C}} \stackrel{\delta}{\longleftrightarrow} \widehat{\widehat{C}}^e) \wedge e\text{-}form(\widehat{\widehat{P}}'_i, \widehat{\widehat{P}}_i^{[\alpha,e]})
\end{aligned}
$$

The first clause depicts the case when the argument slice is already in its e-form, and no e-form δ-relation is applicable to any of its clauses $\widehat{\widehat{C}}$. The second line depicts the case when one of the clauses $\widehat{\widehat{C}}$ of $\widehat{\widehat{P}}_i$ is selected and an e-form δ-relation is applied to it, yielding a new clause $\widehat{\widehat{C}}^e$; this new clause is used to assemble a temporary argument slice $\widehat{\widehat{P}}'_i$ which is then used to recursively obtain $\widehat{\widehat{P}}_i^{[\alpha,e]}$.

Example: The following clause- and mode-annotated argument slice of *collect*/2:

```
                                𝒫(B) :-         {B/g}            ⟨⟨offer(B)⟩⟩
                     {B/g}        B = [].       {B/g}
                                𝒫(B) :-         {B/g,X/f,Ys/f}   ⟨⟨offer(B)⟩⟩
⟨⟨required(X)⟩⟩ {B/g,X/g,Ys/f}    B = [X|Ys],   {B/g,X/g,Ys/g}   ⟨⟨offer(Ys)⟩⟩
               {B/g,X/g,Ys/g}    integer(X),   {B/g,X/g,Ys/g}
               {B/g,X/g,Ys/g}    𝒫(Ys).        {B/g,X/g,Ys/g}
                                𝒫(B) :-         {B/g}            ⟨⟨offer(B)⟩⟩
                     {B/g}       𝒫(B).          {B/g}
```

yields the following e-form

```
                                    𝒫(B) :-        {B/g}                  ⟨⟨offer(B)⟩⟩
                         {B/g}        B = [].      {B/g}
                                    𝒫(B) :-        {B/g,X/f,Z/f,Ys/f}     ⟨⟨offer(B)⟩⟩
                {B/g,X/g,Z/f,Ys/f}    B = [Z|Ys],  {B/g,X/g,Z/g,Ys/g}     ⟨⟨offer(Z,Ys)⟩⟩
⟨⟨required(X)⟩⟩  {B/g,X/g,Z/g,Ys/f}    X = Z,       {B/g,X/g,Z/g,Ys/g}
                {B/g,X/g,Z/g,Ys/g}    integer(X),  {B/g,X/g,Z/g,Ys/g}
                {B/g,X/g,Z/g,Ys/g}    𝒫(Ys).       {B/g,X/g,Z/g,Ys/g}
                                    𝒫(B) :-        {B/g,Z/f}              ⟨⟨offer(B)⟩⟩
                      {B/g,Z/f}       Z = B,       {B/g,Z/g}              ⟨⟨offer(Z)⟩⟩
                      {B/g,Z/g}       𝒫(Z).        {B/g,Z/g}
```

The variable `X` in the second subgoal of the second clause is replaced with `Y`, and an extra subgoal "`Z = X`" is inserted by relation B.3. The recursive call of the third clause has its `B` variable replaced with `Z`, with a subgoal "`Z = B`" also being inserted, this time by relation B.1. The complex "`B = [X|Ys]`" subgoal in the second clause, which performs a data structure decomposition and tests parts of the data structure, is replaced in the e-form by two distinct subgoals, one performing solely the data structure decomposition, and another performing a simple test: these two computations can be addressed separately when they are reimplemented. In the third clause, the equality relationship between the variables in the second argument position of the head goals and its recursive call is explicitly represented, thus enabling different possible reimplementations.

The e-form δ-relations describe the insertion of clause- and mode-annotated subgoals and can be seen as specialised forms of the *fold* and *unfold* transformation rules described in [TS84]. When δ is used to rewrite the clause on its left side in the form described on its right side, then a folding operation takes place; the mode-annotations provide the information necessary to carry out the folding operation in the appropriate subgoal. When δ is used to rewrite the clause on its right side in the form described on its left side, then a simple form of unfolding takes place, where the subgoals performing explicit instantiations are replaced by the corresponding result of their instantiation. Since variables are inserted or deleted in the clause, the clause- and mode-annotations must be updated to reflect these changes.

Relation $\Re^p$

The auxiliary predicate $\Re^p$ relates an argument slice $\widehat{\widetilde{P}}_i^{[\alpha,e]}$ in its e-form to its procedural abstraction $\widehat{\widetilde{P}}_i^{[\alpha,p]}$, being defined as

$$
\begin{array}{rcl}
\Re^p(\widehat{\widetilde{P}}_i, \widehat{\widetilde{P}}_i^{[\alpha,p]}) & \Leftarrow & \Re^p_C(\widehat{\widetilde{P}}_i, \widehat{\widetilde{P}}'_i) \wedge \Re^p(\widehat{\widetilde{P}}'_i, \widehat{\widetilde{P}}_i^{[\alpha,p]}) \\
\Re^p(\widehat{\widetilde{P}}_i, \widehat{\widetilde{P}}_i^{[\alpha,p]}) & \Leftarrow & \neg\Re^p_C(\widehat{\widetilde{P}}_i, \widehat{\widetilde{P}}'_i) \wedge \widehat{\widetilde{P}}_i = \widehat{\widetilde{P}}_i^{[\alpha,p]}
\end{array}
$$

The first clause recursively relates $\widehat{\widetilde{P}}_i$ to $\widehat{\widetilde{P}}_i^{[\alpha,p]}$ by means of an intermediate construct $\widehat{\widetilde{P}}'_i$ obtained via the auxiliary predicate $\Re^p_C$. $\Re^p_C$ holds if, and only if, $\widehat{\widetilde{P}}'_i$ differs from $\widehat{\widetilde{P}}_i$ in exactly one clause: this clause has a counterpart in $\widehat{\widetilde{P}}_i$ but has one of its subgoals replaced by a descriptor different from *relation*, that is, one of of the descriptors bearing a procedural meaning. $\Re^p_C$ is defined below. The second clause states that an argument slice $\widehat{\widetilde{P}}_i$ is $\Re^p$-related to another argument slice $\widehat{\widetilde{P}}_i^{[\alpha,p]}$ if it is not possible to find an intermediate argument slice $\widehat{\widetilde{P}}'_i$ that fulfils the $\Re^p_C$ relation; in this case, $\widehat{\widetilde{P}}_i$ and $\widehat{\widetilde{P}}_i^{[\alpha,p]}$ are bound to be the same. Intuitively, the definition above states that the procedural abstraction $\widehat{\widetilde{P}}_i^{[\alpha,p]}$ of argument slice $\widehat{\widetilde{P}}_i$ is found when it is not possible to apply $\Re^p_C$ to the latter.

The definition of $\Re^p_C$ is a mapping between two argument slices such they differ only in one clause:

$$
\begin{array}{rcl}
\Re^p_C(\widehat{\widetilde{P}}_i, \widehat{\widetilde{P}}_i^{\alpha}) & \Leftarrow & \widehat{\widetilde{P}}_i = \langle \vec{C}_1\, \widehat{\widetilde{C}}\, \vec{C}_2\, \widehat{\widetilde{C}}'\, \widehat{\widetilde{C}}\, \vec{C}_3 \rangle \wedge \widehat{\widetilde{P}}_i^{\alpha} = \langle \vec{C}_1\, \widehat{\widetilde{C}}\, \vec{C}_2\, \widehat{\widetilde{C}}'\, \vec{C}_3 \rangle \\
\Re^p_C(\widehat{\widetilde{P}}_i, \widehat{\widetilde{P}}_i^{\alpha}) & \Leftarrow & \widehat{\widetilde{P}}_i = \langle \vec{C}_1\, \widehat{\widetilde{C}}\, \vec{C}_2 \rangle \wedge \widehat{\widetilde{P}}_i^{\alpha} = \langle \vec{C}_1\, \widehat{\widetilde{C}}^{\alpha}\, \vec{C}_2 \rangle \wedge \Re^p_S(\widehat{\widetilde{C}}, \widehat{\widetilde{C}}^{\alpha})
\end{array}
$$

The first case relates two mode-annotated argument slices $\hat{\hat{P}}_i$ and $\hat{\hat{P}}_i^\alpha$ if clause $\hat{\hat{C}}$ in $\hat{\hat{P}}_i^\alpha$ appears replicated after clause $\hat{\hat{C}}'$ in $\hat{\hat{P}}_i$: the procedural abstraction of an argument slice does not have repeated clauses. The second case relates $\hat{\hat{P}}_i$ and $\hat{\hat{P}}_i^\alpha$ if they differ only in clauses $\hat{\hat{C}}$ and $\hat{\hat{C}}^\alpha$ and they satisfy $\Re^p_S$.

$\Re^p_S$ relates two clause- and mode-annotated clauses when one of their subgoals are δ-related:

$$\begin{array}{rcl}\Re^p_S(\hat{\hat{C}}, \hat{\hat{C}}^\alpha) & \Longleftarrow & \hat{\hat{C}} = H \texttt{ :- } \theta\gamma\,\langle \vec{S}_1 \hat{\hat{S}} \vec{S}_2\rangle \wedge \\ & & \hat{\hat{C}}^\alpha = H \texttt{ :- } \theta\gamma\,\langle \vec{S}_1 \hat{\hat{S}}^\alpha \vec{S}_2\rangle \wedge \\ & & \hat{\hat{S}} \stackrel{\delta}{\longleftrightarrow} \hat{\hat{S}}^\alpha \end{array}$$

$\Re^p_S$ holds when $\hat{\hat{S}}$ in $\hat{\hat{C}}$ is δ-related to $\hat{\hat{S}}^\alpha$ in $\hat{\hat{C}}^\alpha$. The δ-relations mentioned in the definition are only those mapping actual clause- and mode-annotated Prolog subgoals to those descriptors different from *relation*, that is, δ-relations B.9 to B.21.

Example: The first clause- and mode-annotated argument slice of *append*/3 performing a list decomposition (Section 5.6) is such that the following holds:

$$\Re^p\left(\boxed{\begin{array}{lll} & \mathcal{P}\texttt{(A):-} & \theta_0^1\ \gamma_0^1 \\ \gamma_1^1\ \theta_1^1 & \quad\texttt{A = [].} & {\theta_1^1}'\ {\gamma_1^1}' \\ & \mathcal{P}\texttt{(A):-} & \theta_0^2\ \gamma_0^2 \\ \gamma_1^2\ \theta_1^2 & \quad\texttt{A = [D|E],} & {\theta_1^2}'\ {\gamma_1^2}' \\ \gamma_2^2\ \theta_2^2 & \quad\mathcal{P}\texttt{(E).} & {\theta_2^2}'\ {\gamma_2^2}' \end{array}}\ ,\ \boxed{\begin{array}{lll} & \mathcal{P}\texttt{(A):-} & \theta_0^1\ \gamma_0^1 \\ \gamma_1^1\ \theta_1^1 & \quad test\texttt{(A).} & {\theta_1^1}'\ {\gamma_1^1}' \\ & \mathcal{P}\texttt{(A):-} & \theta_0^2\ \gamma_0^2 \\ \gamma_1^2\ \theta_1^2 & \quad decomp\texttt{(A,\{D,E\}),} & {\theta_1^2}'\ {\gamma_1^2}' \\ \gamma_2^2\ \theta_2^2 & \quad\mathcal{P}\texttt{(E).} & {\theta_2^2}'\ {\gamma_2^2}' \end{array}}\right)$$

Relation $\Re^*$

Relation $\Re^*$ is defined very similarly to $\Re^p$ above. They differ in the δ-relations employed to map the abstract and specific subgoals of their argument slices. $\Re^*$ is defined as

$$\begin{array}{rcl}\Re^*(\hat{\hat{P}}_i, \hat{\hat{P}}_i^{[\alpha,*]}) & \Longleftarrow & \Re^*_C(\hat{\hat{P}}_i, \hat{\hat{P}}_i') \wedge \Re^*(\hat{\hat{P}}_i', \hat{\hat{P}}_i^{[\alpha,*]}) \\ \Re^*(\hat{\hat{P}}_i, \hat{\hat{P}}_i^{[\alpha,*]}) & \Longleftarrow & \neg\Re^*_C(\hat{\hat{P}}_i, \hat{\hat{P}}_i') \wedge \hat{\hat{P}}_i = \hat{\hat{P}}_i^{[\alpha,*]} \end{array}$$

$\Re^*_C$ is defined as

$$\begin{array}{rcl}\Re^*_C(\hat{\hat{P}}_i, \hat{\hat{P}}_i^\alpha) & \Longleftarrow & \hat{\hat{P}}_i = \langle \vec{C}_1\, \hat{\hat{C}}\, \vec{C}_2\, \hat{\hat{C}}'\, \hat{\hat{C}}\, \vec{C}_3\rangle \wedge \hat{\hat{P}}_i^\alpha = \langle \vec{C}_1\, \hat{\hat{C}}\, \vec{C}_2\, \hat{\hat{C}}'\, \vec{C}_3\rangle \\ \Re^*_C(\hat{\hat{P}}_i, \hat{\hat{P}}_i^\alpha) & \Longleftarrow & \hat{\hat{P}}_i = \langle \vec{C}_1\, \hat{\hat{C}}\, \vec{C}_2\rangle \wedge \hat{\hat{P}}_i^\alpha = \langle \vec{C}_1\, \hat{\hat{C}}^\alpha\, \vec{C}_2\rangle \wedge \Re^*_S(\hat{\hat{C}}, \hat{\hat{C}}^\alpha) \end{array}$$

Finally, the definition of $\Re^*_S$ is

$$\begin{array}{rcl} \Re^*_S(\hat{\hat{C}}, \hat{\hat{C}}^\alpha) & \Leftarrow & \hat{\hat{C}} = H \text{ :- } \theta\gamma \, \langle \vec{S}_1 \vec{S}_2 \vec{S}_3 \rangle \wedge \\ & & \hat{\hat{C}}^\alpha = H \text{ :- } \theta\gamma \, \langle \vec{S}_1 \hat{\hat{S}}^\alpha \vec{S}_3 \rangle \wedge \\ & & \vec{S}_2 \stackrel{\delta}{\longleftrightarrow} \hat{\hat{S}}^\alpha \end{array}$$

$\Re^*_S$ holds when $\vec{S}_2$ in $\hat{\hat{C}}$ is δ-related to $\hat{\hat{S}}^\alpha$ in $\hat{\hat{C}}^\alpha$. The δ-relations mentioned in the definition are only those mapping predicate descriptors into the *relation* descriptor, that is, δ-relations B.22 to B.30. Another distinct feature of this definition is that a (possibly singleton) *vector* of clause- and mode-annotated subgoals $\vec{S}_2$ is abstracted as $\hat{\hat{S}}^\alpha$.

Example: The following holds:

$$\Re^*(\boxed{\begin{array}{lll} & \mathcal{P}(\texttt{A})\texttt{:-} & \theta^1_0 \; \gamma^1_0 \\ \gamma^1_1 \, \theta^1_1 & \quad test(\texttt{A})\texttt{.} & \theta^{1\,\prime}_1 \, \gamma^{1\,\prime}_1 \\ & \mathcal{P}(\texttt{A})\texttt{:-} & \theta^2_0 \; \gamma^2_0 \\ \gamma^2_1 \, \theta^2_1 & \quad decomp(\texttt{A,\{D,E\}})\texttt{,} & \theta^{2\,\prime}_1 \, \gamma^{2\,\prime}_1 \\ \gamma^2_2 \, \theta^2_2 & \quad \mathcal{P}(\texttt{E})\texttt{.} & \theta^{2\,\prime}_2 \, \gamma^{2\,\prime}_2 \end{array}} , \boxed{\begin{array}{lll} & \mathcal{P}(\texttt{A})\texttt{:-} & \theta^1_0 \; \gamma^1_0 \\ \gamma^1_1 \, \theta^1_1 & \quad relation(\texttt{A})\texttt{.} & \theta^{1\,\prime}_1 \, \gamma^{1\,\prime}_1 \\ & \mathcal{P}(\texttt{A})\texttt{:-} & \theta^2_0 \; \gamma^2_0 \\ \gamma^2_1 \, \theta^2_1 & \quad relation(\texttt{\{A,D,E\}})\texttt{,} & \theta^{2\,\prime}_1 \, \gamma^{2\,\prime}_1 \\ \gamma^2_2 \, \theta^2_2 & \quad \mathcal{P}(\texttt{E})\texttt{.} & \theta^{2\,\prime}_2 \, \gamma^{2\,\prime}_2 \end{array}})$$

Homogeneity during the Abstraction and Reimplementation

The definitions above provide a homogeneous framework in which both the abstraction and reimplementation stages can be carried out, depending on the usage of *pred*/3, *e-form*/2, $\Re^p/2$ and $\Re^*/2$.

In *pred*/3, if the first argument $\hat{\hat{P}}_i$, an actual clause- and mode-annotated argument slice extracted by our method of the previous chapter, is provided then it obtains $\hat{\hat{P}}^{[\alpha,0]}_i$, a version of $\hat{\hat{P}}_i$ in which the symbol $\mathcal{P}$ replaces the predicate symbol q in the head goal and recursive subgoals of the latter; the symbol q is also obtained in this context. If, alternatively, $\hat{\hat{P}}^{[\alpha,0]}_i$ and q are provided, *pred*/3 obtains the actual clause- and mode-annotated slice $\hat{\hat{P}}_i$. Predicate *e-form*/2 can be used to obtain the e-form, if the first argument is provided, or to obtain the conventional (non-explicit) form of its second argument.

$\Re^p/2$ and $\Re^*/2$ can also be used in different manners. If their first argument is provided then the second argument (the procedural abstraction or the most abstract version, respectively) is obtained. If the second argument, $\hat{\hat{P}}^{[\alpha,p]}_i$ or $\hat{\hat{P}}^{[\alpha,*]}_i$, is supplied, then $\hat{\hat{P}}_i$, a more specialised version of it, is obtained.

The alternative usage of $\Re^p/2$ and $\Re^*/2$ to specialise a clause- and mode-annotated argument slice does, however, require human intervention because there might be an infinite number of possible specialisations for a given abstract subgoal. For instance, the abstract mode-annotated subgoal

$$\{\ldots\mathtt{A/g}\ldots\}\ \mathit{test}(\{\mathtt{A}\})\ \{\ldots\mathtt{A/g}\ldots\}$$

can be specialised as any of the system predicates ¶ (*integer*/1, *ground*/1, and so on) or as a subgoal of the form "$\mathtt{A} \diamond a$", where $\diamond$ and a must also be defined. These choices depend on the users' needs: they should interactively customise these abstract constructs to suit their purposes, as explained in the following chapter.

The definitions of $\Re^p/2$ and $\Re^*/2$, however, have to be slightly enhanced to allow for the human contribution in the customisation process. This is achieved by inserting a third argument position containing the *history of transformations* R of the abstraction/reimplementation process. The transformations performed by the δ-relations upon the argument slices are stored in R. R is a possibly empty sequence of r-terms. Each r-term represents a record of a transformation relating two clause- and mode-annotated argument slices $\hat{\hat{P}}_i$ and $\hat{\hat{P}}_i^\alpha$, and may have one of the following two formats:

- $\mathit{copy}(\hat{\hat{C}}, c)$ — clause- and mode-annotated clause $\hat{\hat{C}}$ of $\hat{\hat{P}}_i^\alpha$ is copied after the c-th clause of $\hat{\hat{P}}_i$;
- $\mathit{subst}(\vec{S}/\hat{\hat{S}}^\alpha, c, s)$ — the vector of clause- and mode-annotated subgoals $\vec{S}$ in $\hat{\hat{P}}_i$ is δ-related to the s-th clause- and mode-annotated subgoal $\hat{\hat{S}}^\alpha$ in the c-th clause of $\hat{\hat{P}}_i^\alpha$.

The enhanced definition of $\Re^p$ incorporating the history of transformations is

$$\begin{array}{rcl}
\Re^p(\hat{\hat{P}}_i, \hat{\hat{P}}_i^{[\alpha,p]}, \mathsf{R}) & \Longleftarrow & \mathsf{R} = \langle r, \mathsf{R}' \rangle \wedge \Re^p_C(\hat{\hat{P}}_i, \hat{\hat{P}}'_i, r) \wedge \Re^p(\hat{\hat{P}}'_i, \hat{\hat{P}}_i^{[\alpha,p]}, \mathsf{R}') \\
\Re^p(\hat{\hat{P}}_i, \hat{\hat{P}}_i^{[\alpha,p]}, \mathsf{R}) & \Longleftarrow & \mathsf{R} = \langle\ \rangle \wedge \neg\Re^p_C(\hat{\hat{P}}_i, \hat{\hat{P}}'_i, r) \wedge \hat{\hat{P}}_i = \hat{\hat{P}}_i^{[\alpha,p]}
\end{array}$$

The auxiliary predicate $\Re^p_C$ enhanced with the extra argument holding the r-term is defined as

$$\Re^p_C(\hat{\hat{P}}_i, \hat{\hat{P}}_i^\alpha, r) \quad \Longleftarrow \quad \hat{\hat{P}}_i = \langle \vec{C}_1\, \hat{\hat{C}}\, \vec{C}_2\, \hat{\hat{C}}'\, \hat{\hat{C}}\, \vec{C}_3 \rangle \wedge \hat{\hat{P}}_i^\alpha = \langle \vec{C}_1\, \hat{\hat{C}}\, \vec{C}_2\, \hat{\hat{C}}'\, \vec{C}_3 \rangle \wedge$$

$$
\begin{aligned}
& & & pos(\hat{\hat{C}}', \hat{\hat{P}}_i^{\alpha}, c) \wedge r = copy(\hat{\hat{C}}, c) \\
\Re^p_C(\hat{\hat{P}}_i, \hat{\hat{P}}_i^{\alpha}) & \Longleftarrow & & \hat{\hat{P}}_i = \langle \vec{C}_1\, \hat{\hat{C}}\, \vec{C}_2 \rangle \wedge \hat{\hat{P}}_i^{\alpha} = \langle \vec{C}_1\, \hat{\hat{C}}^{\alpha}\, \vec{C}_2 \rangle \wedge pos(\hat{\hat{C}}^{\alpha}, \hat{\hat{P}}_i^{\alpha}, c) \wedge \\
& & & \Re^p_S(\hat{\hat{C}}, \hat{\hat{C}}^{\alpha}, \hat{\hat{S}}/\hat{\hat{S}}^{\alpha}, c, s) \wedge r = subst(\hat{\hat{S}}/\hat{\hat{S}}^{\alpha}, c, s)
\end{aligned}
$$

where $pos/3$ is a predicate that holds if c is the position of the first argument in the sequence comprising the second argument, defined formally in Appendix E (Def. E.3.1). Finally, the definition of the enhanced form of $\Re^p_S$ is

$$
\begin{aligned}
\Re^p_S(\hat{\hat{C}}, \hat{\hat{C}}^{\alpha}, \vec{S}_2/\hat{\hat{S}}_2^{\alpha}) & \Longleftarrow & & \hat{\hat{C}} = H \text{ :- } \theta\gamma\, \langle \vec{S}_1 \hat{\hat{S}} \vec{S}_2 \rangle \wedge \\
& & & \hat{\hat{C}}^{\alpha} = H \text{ :- } \theta\gamma\, \langle \vec{S}_1 \hat{\hat{S}}^{\alpha} \vec{S}_2 \rangle \wedge \\
& & & \hat{\hat{S}} \stackrel{\delta}{\longleftrightarrow} \hat{\hat{S}}^{\alpha} \wedge pos(\hat{\hat{S}}_2^{\alpha}, \hat{\hat{C}}^{\alpha}, s)
\end{aligned}
$$

The sequence R of transformations provides a record on how $\hat{\hat{P}}_i$ can be transformed into $\hat{\hat{P}}_i^{\alpha}$ and vice-versa. If the $\Re^p/2$ and $\Re^*/2$ predicates are being used to abstract automatically an argument slice, then R is obtained as the transformations are applied; if R is provided to $\Re^p/2$ and $\Re^*/2$ then it guides the transformation process.

5.7.7 Bridging the Gap between Abstract Forms

Between the e-form and the procedural abstraction of an argument slice there are a number of intermediate representations in which the subgoals are gradually abstracted, giving rise to the procedural abstraction. The same is true between the procedural abstraction and the most abstract version of the argument slice.

There are different orderings in which the δ-relations can be applied to an argument slice. The procedural abstraction provides a point of convergence for the possible ways to abstract an argument slice in its e-form. The most abstract version also provides a unique point of convergence for all the possible ways to abstract an argument slice in its procedural abstraction version. The diagram in Figure 5.4 below illustrates the branching out and re-converging of the abstraction process. Our system offers a facility to help users exploring the gaps between the abstract forms. By providing a history of transformations, the user can define specific paths between the referential abstractions $\hat{\hat{P}}_i^{[\alpha,e]}$ and $\hat{\hat{P}}_i^{[\alpha,p]}$ and between $\hat{\hat{P}}_i^{[\alpha,p]}$ and $\hat{\hat{P}}_i^{[\alpha,*]}$. These paths can be inserted in the hierarchy $\mathcal{H}$, its nodes being incorporated to the library of components and offering alternative views to the original argument slice.

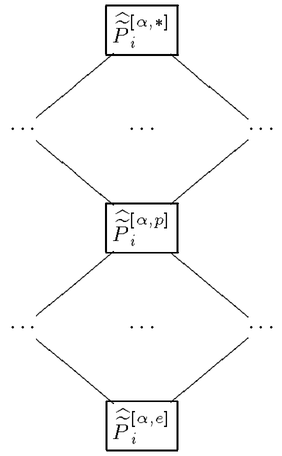


Figure 5.4: Branching out and Re-convergence of Abstraction

As an illustrative example, let there be the first argument slice of *append*/3 shown in Section 5.6: it is such that the abstraction shown in Figure 5.5 is defined: The

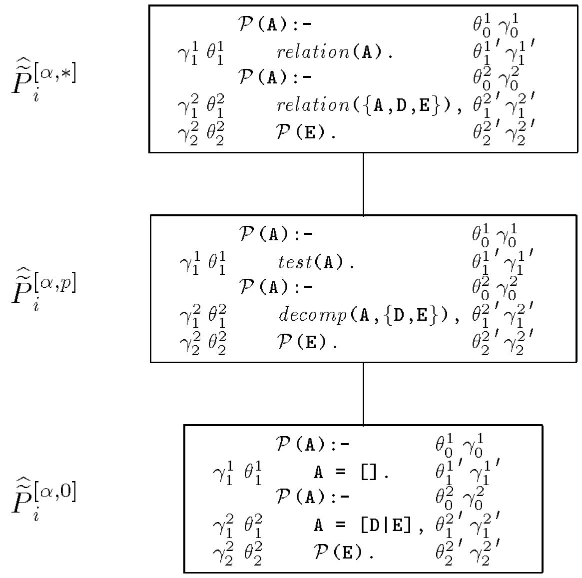
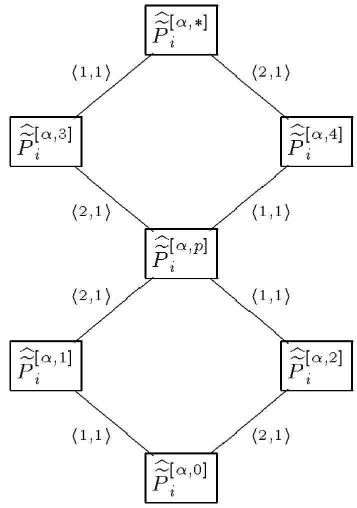


Figure 5.5: Three Strategic Points in the Abstraction of a Component

intermediate abstractions between the strategic points can be explored by the user: since the abstraction is automatic, the user only needs to provide a pair of integers $\langle c, s\rangle$, where c is the clause and s is the subgoal to be abstracted. The abstraction framework defined above automatically obtains a more abstract component in which the subgoal occupying position s in clause c is applied a single δ-relation. If the user provides the pairs $\langle 1,1\rangle$ and $\langle 2,1\rangle$ for exploring the space between $\widehat{\widehat{P}}_i^{[\alpha,e]}$ and $\widehat{\widehat{P}}_i^{[\alpha,p]}$, and the same pair to explore the space between $\widehat{\widehat{P}}_i^{[\alpha,p]}$ and $\widehat{\widehat{P}}_i^{[\alpha,*]}$, the system would obtain the abstraction shown in the diagram

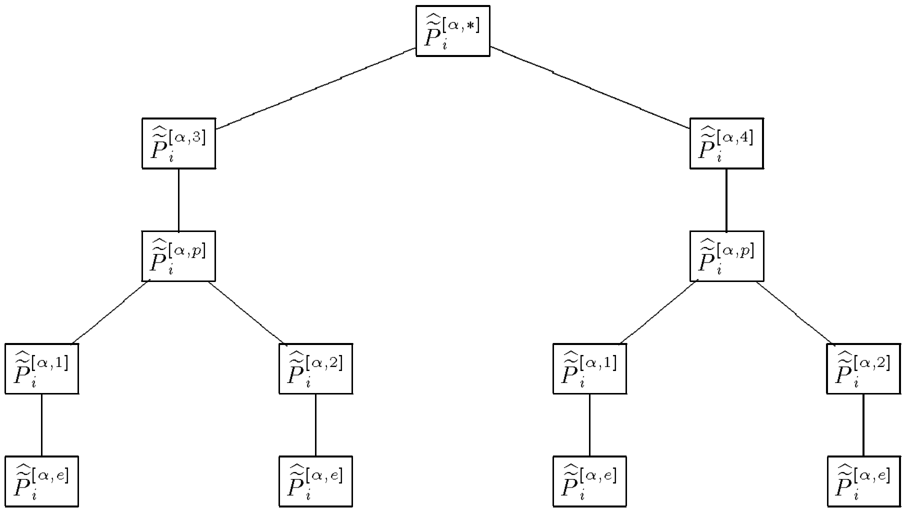


where

$$\widehat{\widehat{P}}_i^{[\alpha,1]} = \boxed{\begin{array}{lll} & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^1\ \gamma_0^1 \\ \gamma_1^1\ \theta_1^1 & \quad test(\{\mathtt{A}\})\mathtt{.} & \theta_1^{1\prime}\ \gamma_1^{1\prime} \\ & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^2\ \gamma_0^2 \\ \gamma_1^2\ \theta_1^2 & \quad \mathtt{A\ =\ [D|E],} & \theta_1^{2\prime}\ \gamma_1^{2\prime} \\ \gamma_2^2\ \theta_2^2 & \quad \mathcal{P}(\mathtt{E})\mathtt{.} & \theta_2^{2\prime}\ \gamma_2^{2\prime} \end{array}} \quad \widehat{\widehat{P}}_i^{[\alpha,2]} = \boxed{\begin{array}{lll} & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^1\ \gamma_0^1 \\ \gamma_1^1\ \theta_1^1 & \quad \mathtt{A\ =\ [].} & \theta_1^{1\prime}\ \gamma_1^{1\prime} \\ & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^2\ \gamma_0^2 \\ \gamma_1^2\ \theta_1^2 & \quad decomp(\mathtt{A},\{\mathtt{D},\mathtt{E}\})\mathtt{,} & \theta_1^{2\prime}\ \gamma_1^{2\prime} \\ \gamma_2^2\ \theta_2^2 & \quad \mathcal{P}(\mathtt{E})\mathtt{.} & \theta_2^{2\prime}\ \gamma_2^{2\prime} \end{array}}$$

$$\widehat{\widehat{P}}_i^{[\alpha,3]} = \boxed{\begin{array}{lll} & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^1\ \gamma_0^1 \\ \gamma_1^1\ \theta_1^1 & \quad relation(\{\mathtt{A}\})\mathtt{.} & \theta_1^{1\prime}\ \gamma_1^{1\prime} \\ & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^2\ \gamma_0^2 \\ \gamma_1^2\ \theta_1^2 & \quad decomp(\mathtt{A},\{\mathtt{D},\mathtt{E}\})\mathtt{,} & \theta_1^{2\prime}\ \gamma_1^{2\prime} \\ \gamma_2^2\ \theta_2^2 & \quad \mathcal{P}(\mathtt{E})\mathtt{.} & \theta_2^{2\prime}\ \gamma_2^{2\prime} \end{array}} \quad \widehat{\widehat{P}}_i^{[\alpha,4]} = \boxed{\begin{array}{lll} & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^1\ \gamma_0^1 \\ \gamma_1^1\ \theta_1^1 & \quad test(\{\mathtt{A}\})\mathtt{.} & \theta_1^{1\prime}\ \gamma_1^{1\prime} \\ & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^2\ \gamma_0^2 \\ \gamma_1^2\ \theta_1^2 & \quad relation(\{\mathtt{A},\mathtt{D},\mathtt{E}\})\mathtt{,} & \theta_1^{2\prime}\ \gamma_1^{2\prime} \\ \gamma_2^2\ \theta_2^2 & \quad \mathcal{P}(\mathtt{E})\mathtt{.} & \theta_2^{2\prime}\ \gamma_2^{2\prime} \end{array}}$$

The new abstractions are inserted in $\mathcal{H}$ as branches of a tree, with similar subsequences leading to the abstract nodes being shared. Our example above yields the following tree:



Not every abstraction can be obtained within our proposed framework. In particular, those abstractions combining *relation* predicate descriptors and actual Prolog subgoals are not addressed. If we were to explore every possibility this would lead to a combinatorial number of abstractions.

5.8 Procedural Abstraction and Most Abstract Argument Slices

If $\Re^p$ is used to automatically abstract a clause- and mode-annotated argument slice in its e-form $\widehat{\widehat{P}}_i^{[\alpha,e]}$ then the procedural abstraction $\widehat{\widehat{P}}_i^{[\alpha,p]}$ of that component is guaranteed to be eventually reached and is such that $\Re^p$ will not abstract it further. $\widehat{\widehat{P}}_i^{[\alpha,p]}$ has no duplicate clauses and each of its clauses is of the form $\mathcal{P}(x)$:- $\theta' \gamma' \vec{S}$, $\vec{S}$ being a non-empty vector of clause- and mode-annotated subgoals $\gamma\,\theta\;\mathcal{P}(y)\;\theta'\,\gamma'$ or $\gamma\,\theta\;q(\vec{V})\;\theta'\,\gamma'$ where q is a predicate descriptor different from *relation*. It is possible that there are no recursive subgoals $\mathcal{P}(y)$, but there is at least one subgoal $q(\vec{V})$ in $\vec{S}$.

Similarly, if $\Re^*$ is used to automatically abstract a procedural abstraction $\widehat{\widehat{P}}_i^{[\alpha,p]}$ of a clause- and mode-annotated argument slice then a most abstract procedure $\widehat{\widehat{P}}_i^{[\alpha,*]}$ is eventually reached and it is such that $\Re^*$ will not abstract it any further. $\widehat{\widehat{P}}_i^{[\alpha,*]}$ has no duplicate clauses and each of its clauses is such that:

1. its body consists only of clause- and mode-annotated recursive calls and *relation* subgoals;
2. there are no two consecutive *relation* subgoals;
3. there is at least one *relation* subgoal;

That is, its clauses are of the form $\mathcal{P}(x)$:- $\theta' \gamma' \vec{S}$, $\vec{S}$ being a non-empty vector of clause- and mode-annotated subgoals $\gamma\,\theta\;\mathcal{P}(y)\;\theta'\,\gamma'$ or $\gamma\,\theta\;relation(\vec{V})\;\theta'\,\gamma'$, with at least one subgoal of the latter kind and no two consecutive occurrence of such *relation* subgoals. If there are no recursive subgoals $\mathcal{P}(y)$ then $\vec{S}$ consists of a single $relation(\vec{V})$ subgoal.

In Sections B.4 and B.5 we sketch proofs that the procedural abstraction and the most abstract argument slices are eventually obtained.

5.9 Comparison with Existing Work

We use ideas similar to those of [Wat88], in which the abstraction and reimplementation approach is proposed as a translation technique to improve the quality of the target program. In that proposal, both the abstraction and the reimplementation are fully automated tasks: the source program is first analysed in order to obtain an implementation-independent abstract description of the computations performed by the program; this abstract representation is then automatically reimplemented in the target language.

We have adapted this proposal to the problem of software reuse: an initial clause- and mode-annotated argument slice $\hat{\hat{P}}_i$ is abstracted as $\hat{\hat{P}}_i^{\alpha}$ which is then reimplemented as a different argument slice $\hat{\hat{P}}'$: $\hat{\hat{P}}_i$ and $\hat{\hat{P}}_i'$ are similar argument slices. Our work diverges from the proposal of [Wat88] in two ways: firstly, the source and the target languages are the same, Prolog; secondly, the reimplementation stage is not automatic. The abstract argument slice $\hat{\hat{P}}_i^{\alpha}$ is a template with a large (possibly infinite) number of different reimplementations, and in order to prune down this search-space we have a human user interacting and choosing how to redesign the abstractions.

We have proposed an approach to reusing Prolog programs to build new similar programs. Our proposal considers a program as the embodiment of design decisions: the syntax of its subgoals, together with their usage, provide an account of the programming practices chosen by its author to achieve the desired behaviour/results. We suggest that these design decisions be abstracted and special predicate descriptors be assigned to the subgoals of the initial program. These predicate descriptors comprise a repertoire of programming practices commonly found in Prolog programs.

Abstraction and reimplementation is a translation technique [Wat88] designed to improve the quality of the target program. In current approaches, both the abstraction and the reimplementation are fully automated tasks. One of the manners that has been proposed [PGLS88] to implement software reuse is by means of translating the old program to a higher-level representation (abstraction), then supporting changes to be carried out at this representation by the user. The final changed higher-level description can then be translated into the new executable program (reimplementation). If the abstract representation is properly designed, it may improve the user's understanding of the program being reused [Let88]. The automatic abstraction of a program into a more concise and generic, less implementation-dependent form may provide the user with a useful form of documentation which, being extracted from the program, faithfully reports the actual workings of the program.

5.10 Conclusions and Discussion

We have proposed a way to organise and manage a collection of programming techniques. Our adopted view of a library of techniques incorporates the concept of *abstractions* of argument slices and considers a library of techniques as consisting of the techniques themselves and their clause- and mode-annotated argument slices extracted by the method described in Chapter 4. The abstractions encourage the reuse of existing techniques and argument slices to define other components, as described in the following chapter. Abstractions also help drawing relationships between seemingly disparate components, and new techniques can be automatically defined.

According to our proposal, the library of techniques is divided into three components, $\mathcal{T}$, the techniques themselves, $\mathcal{S}$, the clause- and mode-annotated argument slices and their more abstract forms, and $\mathcal{H}$, the hierarchy of argument slices abstractions. $\mathcal{S}$ provides a centralised storage for argument slices such that multiple occurrences of its elements are multiple references to an identifier, avoiding unnecessary replication of large portions of data. $\mathcal{T}$ stores the definition of each supplied technique, in terms of the argument slices in $\mathcal{S}$, also providing pointers to more abstract formulations of its components, via $\mathcal{H}$. The hierarchy of abstractions of the argument slices are stored in $\mathcal{H}$.

We have also proposed a means to upgrade the library of programming techniques, given a programming technique as formalised in Chapter 4. Firstly the argument slices are abstracted and the abstract formulations plus the original form are inserted in $\mathcal{S}$. The relations between the different abstract formats are inserted in $\mathcal{H}$, with the content of each argument slice replaced by its identifier obtained in $\mathcal{S}$. Finally, the techniques are inserted in $\mathcal{T}$, using the identifiers of the argument slices for this purpose.

We do not claim that this is the only manner or the most efficient way to organise a library of programming techniques. We have, however, addressed some important issues, avoiding unnecessary repetitions of components, incorporating the notion of abstractions, and providing the users of our system with different means to view the library.

The abstraction and reimplementation of mode-annotated procedures is carried out via a series of program transformations. These program transformations were presented as δ-relations which can be used interchangeably for abstracting or reimplementing the argument slices. We have also defined a homogeneous framework to carry out both the abstraction and reimplementation, depending on the user's choice. The framework supports

- the automatic abstraction of $\hat{\hat{P}}_i$;
- the user-assisted abstraction of $\hat{\hat{P}}_i$;
- the user-assisted reimplementation of $\hat{\hat{P}}_i^{\alpha}$.

The framework can also check if a more specific clause- and mode-annotated argument slice $\hat{\hat{P}}_i$ can be abstracted as a given $\hat{\hat{P}}_i^{\alpha}$ or vice-versa, and also generate the infinitely many concrete versions of an abstract argument slice $\hat{\hat{P}}_i^{\alpha}$.

5.10.1 Similarities between Programming Techniques

Similarity between techniques can be detected by examining their argument slices. The programming technique used in the first argument slice of *sum*/2 and *append*/3, for instance, are exactly the same, if we overlook the names of variables. By abstracting the argument slices comprising programming techniques even more similarities can be found. The predicate descriptors, concealing particular design decisions in the argument slices, abstract different constructs into the same label, thus enhancing the resemblance between programming techniques.

For example, the following mode-annotated procedure *rev_dl*/3, a procedure to reverse a list using difference lists [SS86, O'K90]

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

yields the following argument slices (in their e-form)

$\widehat{\widehat{P}}_1^{[\alpha,e]} =$

```
                  P(A):-        {A/g}           ⟪ offer({A}) ⟫
         {A/g}      A = [].     {A/g}
                  P(A):-        {A/g,D/f,E/f}   ⟪ offer({A}) ⟫
 {A/g,D/f,E/f}      A = [D|E],  {A/g,D/g,E/g}   ⟪ offer({D,E}) ⟫
 {A/g,D/g,E/g}      P(E).       {A/g,D/g,E/g}
```

$\widehat{\widehat{P}}_2^{[\alpha,e]} =$

```
                                    P(B):-        {B/g}           ⟪offer({B})⟫
                          {B/g}       true(B).    {B/g}
                                    P(B):-        {B/g,D/f,F/f}   ⟪offer({B})⟫
⟪required({D})⟫   {B/g,D/g,F/f}       F = [D|B],  {B/g,D/g,F/g}   ⟪offer({F})⟫
                  {B/g,D/g,F/g}       P(F).       {B/g,D/g,F/g}
```

$\widehat{\widehat{P}}_3^{[\alpha,e]} =$

```
                              P(C):-       {B/g,C/f}
⟪required({B})⟫   {B/g,C/f}     B = C.     {B/g,C/g}   ⟪offer({C})⟫
                              P(C):-       {C/f,Z/f}
                  {C/f,Z/f}     P(Z),      {C/g,Z/g}   ⟪offer({Z})⟫
                  {C/f,Z/f}     C = Z.     {C/g,Z/g}   ⟪offer({C})⟫
```

And the following mode-annotated procedure *sum_ap*/3, a procedure to sum the costs associated with the nodes of a graph using a pair of accumulators

```
                            sum_ap(A,B,C):-       {A/g,B/g,C/f}
            {A/g,B/g,C/f}     destiny(A),         {A/g,B/g,C/f}
            {A/g,B/g,C/f}     B = C.              {A/g,B/g,C/g}
                            sum_ap(A,B,C):-       {A/g,B/g,C/f,D/f,E/f,F/f}
{A/g,B/g,C/f,D/f,E/f,F/f}     A to E costs D,     {A/g,B/g,C/f,D/g,E/g,F/f}
{A/g,B/g,C/f,D/g,E/g,F/f}     F is B + D,         {A/g,B/g,C/f,D/g,E/g,F/g}
{A/g,B/g,C/f,D/g,E/g,F/g}     sum_ap(E,F,C).      {A/g,B/g,C/g,D/g,E/g,F/g}
```

yields the following argument slices (in their e-form)

$\widehat{\widehat{R}}_1^{[\alpha,e]} =$

```
                  P(A):-              {A/g}           ⟪offer({A})⟫
         {A/g}      destiny(A).       {A/g}
                  P(A):-              {A/g,D/f,E/f}   ⟪offer({A})⟫
 {A/g,D/f,E/f}      A to E costs D,   {A/g,D/g,E/g}   ⟪offer({D,E})⟫
 {A/g,D/g,E/g}      P(E).             {A/g,D/g,E/g}
```

$\widehat{\widehat{R}}_2^{[\alpha,e]} =$

```
                                    P(B):-         {B/g}           ⟪offer({B})⟫
                          {B/g}       true(B).     {B/g}
                                    P(B):-         {B/g,D/f,F/f}   ⟪offer({B})⟫
⟪required({D})⟫   {B/g,D/g,F/f}       F is B + D,  {B/g,D/g,F/g}   ⟪offer({F})⟫
                  {B/g,D/g,F/g}       P(F).        {B/g,D/g,F/g}
```

$\widehat{\widehat{R}}_3^{[\alpha,e]} =$

```
                              P(C):-       {B/g,C/f}
⟪required({B})⟫   {B/g,C/f}     B = C.     {B/g,C/g}   ⟪offer({C})⟫
                              P(C):-       {C/f,Z/f}
                  {C/f,Z/f}     P(Z),      {C/g,Z/g}   ⟪offer({Z})⟫
                  {C/f,Z/f}     C = Z.     {C/g,Z/g}   ⟪offer({C})⟫
```

The argument slice $\widehat{\widehat{P}}_1^{[\alpha,e]}$ and $\widehat{\widehat{R}}_1^{[\alpha,e]}$ can be abstracted as

$\widehat{\widehat{P}}_1^{[\alpha,*]} = \widehat{\widehat{R}}_1^{[\alpha,*]} =$

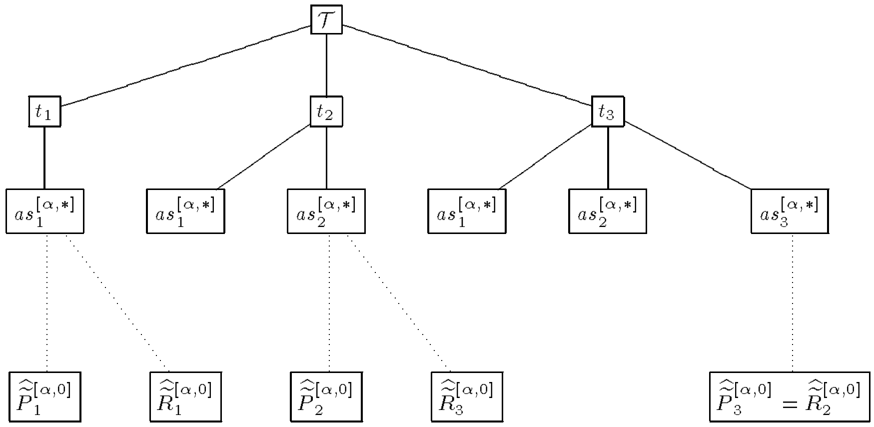
```
                  P(A):-                  {A/g}           ⟪offer({A})⟫
         {A/g}      relation(A).          {A/g}
                  P(A):-                  {A/g,D/f,E/f}   ⟪offer({A})⟫
 {A/g,D/f,E/f}      relation({A,D,E},     {A/g,D/g,E/g}   ⟪offer({D,E})⟫
 {A/g,D/g,E/g}      P(E).                 {A/g,D/g,E/g}
```

and $\widehat{\widehat{P}}_2$ and $\widehat{\widehat{R}}_3$ can be abstracted as

$\widehat{\widehat{P}}_2^{[\alpha,*]} = \widehat{\widehat{R}}_3^{[\alpha,*]} =$

		$\mathcal{P}$(B):-	{B/g}	⟪*offer*({B})⟫
	{B/g}	*relation*(B).	{B/g}	
		$\mathcal{P}$(B):-	{B/g,D/f,F/f}	⟪*offer*({B})⟫
⟪*required*({D})⟫	{B/g,D/g,F/f}	*relation*({F,B,D}),	{B/g,D/g,F/g}	⟪*offer*({F})⟫
	{B/g,D/g,F/g}	$\mathcal{P}$(F).	{B/g,D/g,F/g}	

Although *rev_dl*/3 and *sum_ap*/3 are syntactically different their programming techniques bear similar features, once the particular design decisions are abstracted. They provide our library with 5 clause- and mode-annotated argument slices: once these are abstracted their most abstract forms show further similarities. From this abstract viewpoint our list of techniques $\mathcal{T}$ has only three techniques t_1, t_2 and t_3, with different specialisations, as shown by the diagram below:



5.10.2 The Hierarchy $\mathcal{H}$ as a Lattice

Each subtree rooted in $\mathcal{H}^{in}$ and $\mathcal{H}^{out}$ describes a way to abstract a given clause- and mode-annotated argument slice until its most abstract form is reached. The δ-relations responsible for abstracting an argument slice guarantee, as sketched in Section B.5, that the most abstract form will eventually be reached. The different orderings in which abstractions can be pursued give rise to many different subtrees in the hierarchy $\mathcal{H}$. A more economic way to organise these abstractions would be by means of a lattice in which similar nodes would not be replicated, but shared by different paths. The lattice scheme, however, may become fairly complex: these abstractions are also meant to be navigated by human users. It is visually more appealing to have replicated nodes and a simpler organisation scheme simplifying the navigation on the hierarchy of abstractions than to have an economic lattice representation without replications.

Besides, the actual content of the clause- and mode-annotated argument slices are not stored in the hierarchy $\mathcal{H}$, but only its unique identifier; the content of the argument slices are kept in the list $\mathcal{S}$, explained above.

5.10.3 Inaccurate Mode-Annotations of Subgoals

The mode-annotations play an essential role during the abstraction and reimplementation of argument slices. Each token stands for a set of possible values, and the set represented by a token may be contained within the set of values represented by some other token. During the abstraction of a clause- and mode-annotated subgoal, we try to infer which computations are being performed: in this task we are aided by the syntax of the subgoal and its mode-annotations. If tokens that are supersets are found, then all possible alternatives of more specific tokens have to be pursued separately, since they all represent equally plausible behaviours. The following mode-annotated subgoal, for instance,

`{A/?,B/?} A = B {A/?,B/?}`

stands for any of the following cases with more specific mode-annotations:

1. `{A/f,B/f} A = B {A/f,B/f}`,
2. `{A/f,B/g} A = B {A/g,B/g}`, `{A/g,B/f} A = B {A/g,B/g}`, `{A/i,B/f} A = B {A/i,B/i}`, `{A/f,B/i} A = B {A/i,B/i}`, `{A/i,B/g} A = B {A/g,B/g}`, `{A/g,B/i} A = B {A/g,B/g}`, `{A/i,B/i} A = B {A/g,B/g}`,
3. `{A/g,B/g} A = B {A/g,B/g}`,
4. `{A/i,B/i} A = B {A/i,B/i}`,

These subgoals represent different computations: the first subgoal binds `A` and `B`; the subgoals grouped in the second item change the instantiation status of `A`, `B` or both variables, an assignment; the third item depicts a test; the last item depicts either a test or an assignment.

Instead of listing δ-relations for each system predicate and possible tokens, we have characterised the abstractions/reimplementations in a more generic and economic fashion, in terms of *changes* in the instantiation status of the variables of a subgoal, employing the definitions *change*, *fixed* and *unknown* of Chapter 4.

5.10.4 E-Form of Procedures

The explicit form of the argument slices is not essential to our proposal. If the e-form is not available then δ-relations must be provided to cater for those cases which are avoided by the e-form, for instance, a δ-relation of this form

$$\boxed{\gamma\,\theta\,\Im(x)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\;test(\{x\})\;\theta'\,\gamma'}$$

where $\neg change(x, \theta, \theta')$ would have to be provided. Another consequence of not having the complex computations split into simpler ones would be that the programming practices would be "clustered" making them harder to be reimplemented.

5.10.5 Abstraction and Reverse Software Engineering

Reverse software engineering [PGLS88, CCI90, Van94] concerns the analysis of a system in order to identify its components and their interrelationships and to devise alternative higher-level representations for the system. Our approach of partitioning a procedure into the contributions of each argument position (Chapter 4) and then abstracting them into more abstract forms can be seen as a means to automate the reverse software engineering in Prolog programs. We follow the taxonomy laid out in [CCI90] and consider reverse software engineering as "the process of analysing a subject system to identify its components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction" ([CCI90], page 15).

The argument slices are the components of a procedure with the clause-annotations explicitly relating them. The abstraction process provides us with a conceptual description of each argument slice. This description highlights different aspects of parts or the entire argument slice: the procedural features are portrayed when the actual subgoals are replaced for predicate descriptors; a declarative account of the argument slice is obtained when its subgoals are all replaced by neutral *relation* descriptors.

One is able to conveniently explain the functionality of a procedure by means of its clause- and mode-annotated argument slices and their abstract versions. For instance, the following *path*/2 program

```
path(A,B):-                         path(A,[A]):-
   destination(A),                     destination(A).
   B = [A].
path(A,B):-                         path(A,[A|D]):-
   edge(A,C),                          edge(A,C),
   B = [A|D],                          path(C,D).
   path(C,D).
```

which holds if its second argument is a list with the nodes of a graph represented by means of facts *edge*/2. If it is analysed with respect to its usage mode when building a list in the second argument position, we have the following procedural abstractions of its clause-annotated argument slices[2]:

$\mathcal{P}$(A):- ⟪*offer*({A})⟫ *user_pred*({A}). $\mathcal{P}$(A):- ⟪*offer*({A})⟫ *user_pred*({A,C}), $\mathcal{P}$(C).	$\mathcal{P}$(B):- ⟪*required*({A})⟫ *build*({A},B). $\mathcal{P}$(B):- ⟪*required*({A})⟫ *build*({A,D},B). $\mathcal{P}$(D).

This particular usage of *path*/2 can be explained in terms of its abstracted argument slices: the first argument is responsible for providing values (referred to as A) by means of a user-defined predicate; the second argument slice builds a data structure with those values provided by the first argument slice. The argument slices were abstracted up to a point where each subgoal is replaced by a predicate descriptor with its procedural meaning; if the most abstract form had been used then the relationships between the argument slices would be rephrased in terms of relations between the variables

Our approach uniformly supports the design recovery and the reengineering of the constructs. The abstractions of the argument slices supply us with a simple form of documentation automatically obtained, that is, an updated procedural or declarative account of a procedure. The δ-relations also support the reengineering of programming techniques through the alteration of abstractions into new versions, as explained in the following chapter. The intermediate abstractions of the argument slices are stored in the library because during the reengineering stage the user may want to alter particular design decisions, leaving the rest unchanged.

[2] For the sake of conciseness only those clause-annotations with variables referred to in the other argument slice are shown.

5.10.6 Representing the Abstractions of Components

The four strategic views $\widehat{\widehat{P}}_i^{[\alpha,0]}$, $\widehat{\widehat{P}}_i^{[\alpha,e]}$, $\widehat{\widehat{P}}_i^{[\alpha,p]}$ and $\widehat{\widehat{P}}_i^{[\alpha,*]}$ of a clause- and mode-annotated argument slice are an alternative solution to showing all the possible ways to abstract a component. An exponential number of distinct abstract forms would be generated in the latter case, due to the different orderings of application of the δ-rules to the components' subgoals. It is possible to extend the set of abstractions at the user's request, by the manual exploration of the gap between the strategic points $\widehat{\widehat{P}}_i^{[\alpha,p]}$ and $\widehat{\widehat{P}}_i^{[\alpha,*]}$. This approach, however, still leaves out some abstract forms, namely, those in which portions of the components have already been assigned *relation* descriptors but other portions still contain actual Prolog subgoals to be abstracted.

An alternative solution to the problem of providing an accurate representation for the abstraction space is to show only $\widehat{\widehat{P}}_i^{[\alpha,0]}$, $\widehat{\widehat{P}}_i^{[\alpha,e]}$ and $\widehat{\widehat{P}}_i^{[\alpha,*]}$ and allow the manual exploration of the gap between the e-form $\widehat{\widehat{P}}_i^{[\alpha,e]}$ and the most abstract form $\widehat{\widehat{P}}_i^{[\alpha,*]}$.

5.11 Summary

In this chapter we have:

- proposed a way to store and manage programming techniques extracted by the method described in the previous chapter; and
- described the abstraction process through which the argument slices can have their design decisions concealed.

Chapter 6

Designing Prolog Programming Techniques

6.1 Introduction

The management of a knowledge base has to support its incremental upgrading with new components. In the previous chapters we have described how Prolog programming knowledge, in the form of clause- and mode-annotated argument slices, can be automatically extracted, formalised and inserted into the library of programming techniques, our programming knowledge base.

In this chapter we describe a complementary service supplied by our knowledge-management tool to support the manual definition of programming techniques by a human expert. By means of this service, expert programmers can design new programming techniques from scratch, or by reusing existing components.

A programming technique is designed by the disciplined definition of its argument slices. During the definition of new argument slices the user may choose from a set of abstract clause templates and gradually specialise them using a repertoire of commands. Alternatively, the argument slices extracted from actual procedures and stored in the library can be used to define new components.

A repertoire of commands to manipulate argument slices is made available. These commands can either be applied to an argument slice template or to an abstract version (*i.e.* with at least one predicate descriptor in one of its clauses) of an existing argument slice in the library. The δ-relations, introduced in the previous chapter, comprise our design commands, with an interactive element to narrow down the possibly infinite options during the specialisation of abstract constructs.

The formalism proposed in Chapter 4 views a programming technique as consisting of a sequence of clause- and mode-annotated argument slices. The design of a programming technique, conversely, requires the definition of each of its constituent argument slices and the appropriate binding of their required and offered variables.

Our tool supports the design of argument slices, the building blocks of programming techniques. They can be devised starting from a very abstract template and have it gradually defined, or by reusing existing abstract components (that is, argument slices with predicate descriptors) and refining them.

Alternatively, programming techniques can be defined using existing extracted argument slices chosen from the library and linked in some different manner to other argument slices.

6.2 Designing Argument Slices

The definition of the argument slices of a programming technique can be carried out by choosing an existing component from the tree $\mathcal{H}$ of argument slices in the library, or by devising a new argument slice from "scratch". The preparation of a new argument slice can be performed by means of reengineering existing components (Section 6.2.6) or by assembling a sequence of abstract clause templates and specialising each of them (Section 6.2.1). The expert may also choose an element of $\mathcal{H}$ with predicate descriptors and specialise it differently, thus yielding a new argument slice. The available commands are the δ-relations described in the previous chapter, used here to customise abstract constructs.

6.2.1 Argument Slices via Abstract Clause Templates

During the design of argument slices via abstract clause templates, the user is guided through a series of dialogues aimed at defining generic constructs. The initial template for an argument slice is a sequence of clauses $\langle C_1, \ldots, C_n \rangle$ whose actual number n must be supplied by the user. After the number of clauses of the clause- and mode-annotated argument slice has been supplied, the user can proceed with the definition of each clause C_i.

For each C_i in the initial specification, the user is offered the following template, standing for the *most abstract clause*:

$$\boxed{\begin{array}{l}\mathcal{P}\texttt{(A):-}\\ \quad \mathit{relation}\texttt{(}V_0\texttt{),}\\ \quad \mathcal{P}\texttt{(A1),}\\ \quad \mathit{relation}\texttt{(}V_1\texttt{),}\\ \qquad \vdots\\ \quad \mathcal{P}\texttt{(An),}\\ \quad \mathit{relation}\texttt{(}V_n\texttt{).}\end{array}}$$

The clause- and mode-annotations are omitted at this level of abstraction, but they are shown in later, more specialised, versions. The lack of annotations at this stage provides a cleaner initial template, leaving those details to be supplied later on in the definition.

The user is next prompted about the number of recursive calls each clause is to have. If, for instance, the user decides to devise an argument slice with three clauses, the first one without any recursive calls (base-case clause), the second clause with only one recursive call and the third one with two recursive calls, the tool would present the user with the abstract argument slice resulting from these choices, that is,

$$\boxed{\begin{array}{l}\mathcal{P}\texttt{(A):-}\\ \quad \mathit{relation}\texttt{(}V_0\texttt{).}\\ \mathcal{P}\texttt{(A):-}\\ \quad \mathit{relation}\texttt{(}V_0\texttt{),}\\ \quad \mathcal{P}\texttt{(A1),}\\ \quad \mathit{relation}\texttt{(}V_1\texttt{).}\\ \mathcal{P}\texttt{(A):-}\\ \quad \mathit{relation}\texttt{(}V_0\texttt{),}\\ \quad \mathcal{P}\texttt{(A1),}\\ \quad \mathit{relation}\texttt{(}V_1\texttt{),}\\ \quad \mathcal{P}\texttt{(A2),}\\ \quad \mathit{relation}\texttt{(}V_2\texttt{).}\end{array}}$$

This construct is a generalisation of all those argument slices performing single and double recursive calls, found in meta-interpreters [SL88] or procedures carrying out traversals of nested lists as in the *flatten/2* predicate [SS86, O'K90].

At this stage the sets of variables V_i of each *relation* subgoal must be supplied by the user. In some circumstances, V_i may be specified as empty; depending on the context this will be allowed or not. For instance, in the first clause of our current example, V_0 is not allowed to be empty, since it must depict a relationship for the `A` head goal variable. If, in our example, the user decides to define the sets of variables as:

- Clause 1: $V_0 = \{$`A`$\}$;
- Clause 2: $V_0 = \{$`A,A1`$\}, V_1 = \emptyset$;
- Clause 3: $V_0 = \{$`A,A1,A2`$\}, V_1 = V_2 = \emptyset$.

then the following is obtained:

```
𝒫(A):-
    relation({A}).
𝒫(A):-
    relation({A,A1}),
    𝒫(A1).
𝒫(A):-
    relation({A,A1,A2}),
    𝒫(A1),
    𝒫(A2).
```

Those *relation* subgoals with empty sets have been removed. Another constraint in the definition of the content of the sets of *relation* subgoals is that the variables showing in the head goal and in recursive calls must appear at least once.

An important design decision must be made at this point: the user is to tell the system which purpose is this argument slice going to serve, whether as an *input* or an *output* argument slice. The choice made at this point will provide information for subsequent steps where the mode-annotations are to be supplied. In our current example, the user has decided that the construct under preparation is an input argument slice. Upon this choice, a mode-annotated version is automatically produced:

```
                    𝒫(A):-                    {A/i}
             {A/i}      relation({A}).        {A/i}
                    𝒫(A):-                    {A/i,A1/?}
       {A/i,A1/?}       relation({A,A1}),     {A/i,A1/?}
       {A/i,A1/i}       𝒫(A1).                {A/i,A1/i}
                    𝒫(A):-                    {A/i,A1/?,A2/?}
  {A/i,A1/?,A2/?}       relation({A,A1,A2}),  {A/i,A1/?,A2/?}
  {A/i,A1/i,A2/?}       𝒫(A1),                {A/i,A1/i,A2/?}
  {A/i,A1/i,A2/i}       𝒫(A2).                {A/i,A1/i,A2/i}
```

The system infers the mode of the variables in each clause, using the most specific token to describe it. Since this is an input argument slice, the variables in the recursive subgoals must be at least instantiated before their calls. Instantiated variables cannot become free anymore, hence their mode is preserved from that point onwards. Alternatively, if the user had decided that the intended use of the argument slice is output, the following version would be produced:

	$\mathcal{P}$(A):-	{A/f}
{A/f}	*relation*({A}).	{A/?}
	$\mathcal{P}$(A):-	{A/f,A1/?}
{A/f,A1/?}	*relation*({A,A1}),	{A/?,A1/?}
{A/?,A1/?}	$\mathcal{P}$(A1).	{A/?,A1/?}
	$\mathcal{P}$(A):-	{A/f,A1/?,A2/?}
{A/f,A1/?,A2/?}	*relation*({A,A1,A2}),	{A/?,A1/?,A2/?}
{A/?,A1/?,A2/?}	$\mathcal{P}$(A1),	{A/?,A1/?,A2/?}
{A/?,A1/?,A2/?}	$\mathcal{P}$(A2).	{A/?,A1/?,A2/?}

The user may, however, provide more accurate mode-annotations: these should always contain more specific tokens, otherwise they are not accepted. If, in the input argument slice shown previously the user decides that `A` is ground and the rest of the variables are initially free and that the *relation* subgoals instantiate their variables to "**g**", then we have the following mode-annotated clause being defined:

	$\mathcal{P}$(A):-	{A/g}
{A/g}	*relation*({A}).	{A/g}
	$\mathcal{P}$(A):-	{A/g,A1/f}
{A/g,A1/f}	*relation*({A,A1}),	{A/g,A1/g}
{A/g,A1/g}	$\mathcal{P}$(A1).	{A/g,A1/g}
	$\mathcal{P}$(A):-	{A/g,A1/f,A2/f}
{A/g,A1/f,A2/f}	*relation*({A,A1,A2}),	{A/g,A1/g,A2/g}
{A/g,A1/g,A2/g}	$\mathcal{P}$(A1),	{A/g,A1/g,A2/g}
{A/g,A1/g,A2/g}	$\mathcal{P}$(A2).	{A/g,A1/g,A2/g}

The user need not provide the mode-annotations before and after each subgoal: it is possible to infer the modes of variables in some contexts and the system exploits this. For example, if the modes of `A1` and `A2` in the third clause are supplied as being initially free, then their modes before the *relation* subgoal is also free; if the modes of both `A1` and `A2` are ground after the *relation* subgoal then they remain ground throughout the clause. Whenever the mode of a variable is not known, a "**?**" token can be associated with it.

The clause-annotations can be automatically obtained once the mode-annotations are completed, via the same process described in Section 4.4. Our current example has the following non-empty clause-annotations

	$\mathcal{P}$(A):-	{A/g}	$\langle\langle$*offer*({A})$\rangle\rangle$
{A/g}	*relation*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f}	$\langle\langle$*offer*({A})$\rangle\rangle$
{A/g,A1/f}	*relation*({A,A1}),	{A/g,A1/g}	$\langle\langle$*offer*({A1})$\rangle\rangle$
{A/g,A1/g}	$\mathcal{P}$(A1).	{A/g,A1/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f,A2/f}	$\langle\langle$*offer*({A})$\rangle\rangle$
{A/g,A1/f,A2/f}	*relation*({A,A1,A2}),	{A/g,A1/g,A2/g}	$\langle\langle$*offer*({A1,A2})$\rangle\rangle$
{A/g,A1/g,A2/g}	$\mathcal{P}$(A1),	{A/g,A1/g,A2/g}	
{A/g,A1/g,A2/g}	$\mathcal{P}$(A2).	{A/g,A1/g,A2/g}	

At this point the user has a generic representation for an argument slice, with a prescriptive account of the changes in the instantiation of each variable. Any required variables should also appear in a clause-annotation, this being automatically inferred after the mode-annotations are obtained.

6.2.2 Further Specialisation with δ-Relations

The commands to define argument slices from abstract clause templates described above have no correspondence with the set of δ-relations explained in Chapter 5. They are preliminary transformations of a very general description of argument slices into more specialised versions in which it is expected that

1. the number of clauses of the argument slice has been established;
2. each clause has its number of recursive calls defined;
3. each clause has the sets of variables of every *relation* subgoal defined;
4. the purpose of the argument slice, *i.e.* input or output argument slice, has been made explicit;
5. the mode-annotations have been either inferred or provided (or, more likely, both);
6. the clause-annotations have been inferred.

The system only allows the continuation of the specialisation of the argument slice if, and only if, the requirements above are fulfilled. If that is the case then the user can proceed to the application of $\Re^p$ and $\Re^*$ defined in Section 5.7.6, to further specialise the generic constructs of the clause- and mode-annotated argument slice into Prolog subgoals.

However, during the application of δ-relations to specialise constructs, *i.e.* the δ-relations are used to replace the construct on the right with the component on its left-hand side, there might be infinite possibilities and the only way to narrow this down to a unique construct is by interaction with the user. The mode-annotations may help ruling out some of these possibilities: some δ-relations explicitly lists the requirements the mode-annotations must fulfil in order to have them applied. The participation of the expert is crucial at this point though, since in more specialised constructs there is no way to rule out possibilities, as shown below.

Clauses can be copied any time after the user defines how many they should be. In the current example, the user decided to duplicate the first clause, thus obtaining

	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g}	*relation*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g}	*relation*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g,A1/f}	*relation*({A,A1}),	{A/g,A1/g}	$\langle\!\langle$*offer*({A1})$\rangle\!\rangle$
{A/g,A1/g}	$\mathcal{P}$(A1).	{A/g,A1/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f,A2/f}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g,A1/f,A2/f}	*relation*({A,A1,A2}),	{A/g,A1/g,A2/g}	$\langle\!\langle$*offer*({A1,A2})$\rangle\!\rangle$
{A/g,A1/g,A2/g}	$\mathcal{P}$(A1),	{A/g,A1/g,A2/g}	
{A/g,A1/g,A2/g}	$\mathcal{P}$(A2).	{A/g,A1/g,A2/g}	

The *relation* subgoals may also be split into two or more consecutive subgoals, using δ-relation B.30. In our example, the user decided to apply this relation to the second clause, and also provided the new sets of variables for each *relation*, thus getting

	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g}	*relation*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g}	*relation*({A}),	{A/g}	
{A/g}	*relation*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g,A1/f}	*relation*({A,A1}),	{A/g,A1/g}	$\langle\!\langle$*offer*({A1})$\rangle\!\rangle$
{A/g,A1/g}	$\mathcal{P}$(A1).	{A/g,A1/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f,A2/f}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g,A1/f,A2/f}	*relation*({A,A1,A2}),	{A/g,A1/g,A2/g}	$\langle\!\langle$*offer*({A1,A2})$\rangle\!\rangle$
{A/g,A1/g,A2/g}	$\mathcal{P}$(A1),	{A/g,A1/g,A2/g}	
{A/g,A1/g,A2/g}	$\mathcal{P}$(A2).	{A/g,A1/g,A2/g}	

In our current example, the user decided to specialise the *relation* of the first clause, and being offered the possible applicable δ-relations, chose to use this one

$$\boxed{\gamma\ \theta\ test(V)\ \theta'\ \gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ relation(V)\ \theta'\ \gamma'}$$

The user decided to use the same δ-relation for the first and second *relation* subgoals of the second clause, thus obtaining

	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g}	*test*({A}),	{A/g}	
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g,A1/f}	*relation*({A,A1}),	{A/g,A1/g}	$\langle\!\langle$*offer*({A1})$\rangle\!\rangle$
{A/g,A1/g}	$\mathcal{P}$(A1).	{A/g,A1/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f,A2/f}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g,A1/f,A2/f}	*relation*({A,A1,A2}),	{A/g,A1/g,A2/g}	$\langle\!\langle$*offer*({A1,A2})$\rangle\!\rangle$
{A/g,A1/g,A2/g}	$\mathcal{P}$(A1),	{A/g,A1/g,A2/g}	
{A/g,A1/g,A2/g}	$\mathcal{P}$(A2).	{A/g,A1/g,A2/g}	

For the *relation* subgoal of the third clause, the user decided to apply the following δ-relation

$$\boxed{\gamma\ \theta\ user\text{-}pred(W)\ \theta'\ \gamma'} \xleftrightarrow{\delta} \boxed{\gamma\ \theta\ relation(W)\ \theta'\ \gamma'}$$

and in the fourth clause *relation* is instantiated via

$$\boxed{\gamma\ \theta\ decomp(x,V)\ \theta'\ \gamma'} \xleftrightarrow{\delta} \boxed{\gamma\ \theta\ relation(W)\ \theta'\ \gamma'}$$

thus obtaining

	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g}	*test*({A}),	{A/g}	
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g,A1/f}	*user-pred*({A,A1}),	{A/g,A1/g}	$\langle\!\langle$*offer*({A1})$\rangle\!\rangle$
{A/g,A1/g}	$\mathcal{P}$(A1).	{A/g,A1/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f,A2/f}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
{A/g,A1/f,A2/f}	*decomp*(A,{A1,A2}),	{A/g,A1/g,A2/g}	$\langle\!\langle$*offer*({A1,A2})$\rangle\!\rangle$
{A/g,A1/g,A2/g}	$\mathcal{P}$(A1),	{A/g,A1/g,A2/g}	
{A/g,A1/g,A2/g}	$\mathcal{P}$(A2).	{A/g,A1/g,A2/g}	

The predicate descriptors in the argument slice can be further specialised, yielding actual Prolog constructs. The concrete instances of the predicate descriptors, however, may be infinite. For example, the clause and mode-annotated Prolog subgoals on the left-hand side of the diagram below

$$\begin{tikzcd}
\boxed{\gamma\ \theta\ \texttt{X = []}\ \theta'\ \gamma'} \arrow[dr, leftrightarrow, "\delta"] & \\
\boxed{\gamma\ \theta\ \texttt{X = nil}\ \theta'\ \gamma'} \arrow[r, leftrightarrow, "\delta"] & \boxed{\gamma\ \theta\ test(\{\texttt{X}\})\ \theta'\ \gamma'} \\
\boxed{\gamma\ \theta\ \texttt{integer(X)}\ \theta'\ \gamma'} \arrow[ur, leftrightarrow, "\delta"] &
\end{tikzcd}$$

are all possible instances of the abstract *test* subgoal on the right side. This stage also requires the participation of a human user to choose one among the many appropriate instances of the abstract constructs. In our example, the user decided to apply to the first clause the δ-relation

$$\boxed{\gamma\ \theta\, x\ \texttt{ = true}\, \theta'\ \gamma'} \xleftrightarrow{\delta} \boxed{\gamma\ \theta\ \mathit{test}(\{x\})\ \theta'\ \gamma'}$$

To the two *test* descriptors of the second clause, the user decided to apply respectively

$$\boxed{\gamma\ \theta\, \texttt{system}(x)\ \theta'\ \gamma'} \xleftrightarrow{\delta} \boxed{\gamma\ \theta\ \mathit{test}(\{x\})\ \theta'\ \gamma'}$$

and

$$\boxed{\gamma\ \theta\, \texttt{call}(x)\ \theta'\ \gamma'} \xleftrightarrow{\delta} \boxed{\gamma\ \theta\ \mathit{test}(\{x\})\ \theta'\ \gamma'}$$

The third clause has its *user-pred* descriptor specified by

$$\boxed{\gamma\ \theta\, \texttt{clause}(x\texttt{,}y)\ \theta'\ \gamma'} \xleftrightarrow{\delta} \boxed{\gamma\ \theta\ \mathit{user\text{-}pred}(\{x\}\texttt{,}\{y\})\ \theta'\ \gamma'}$$

And the *decomp* of the fourth clause is specified by

$$\boxed{\gamma\ \theta\, x\ \texttt{ = }\ f(V)\ \theta'\ \gamma'} \xleftrightarrow{\delta} \boxed{\gamma\ \theta\ \mathit{decomp}(x\texttt{,}V)\ \theta'\ \gamma'}$$

and the user provided the details of $f(V)$ as "`(A1,A2)`". The resulting argument slice is

	`𝒫(A):-`	`{A/g}`	⟪*offer*(`{A}`)⟫
`{A/g}`	`A = true.`	`{A/g}`	
	`𝒫(A):-`	`{A/g}`	⟪*offer*(`{A}`)⟫
`{A/g}`	`system(A),`	`{A/g}`	
`{A/g}`	`call(A).`	`{A/g}`	
	`𝒫(A):-`	`{A/g,A1/f}`	⟪*offer*(`{A}`)⟫
`{A/g,A1/f}`	`clause(A,A1),`	`{A/g,A1/g}`	⟪*offer*(`{A1}`)⟫
`{A/g,A1/g}`	`𝒫(A1).`	`{A/g,A1/g}`	
	`𝒫(A):-`	`{A/g,A1/f,A2/f}`	⟪*offer*(`{A}`)⟫
`{A/g,A1/f,A2/f}`	`A = (A1,A2),`	`{A/g,A1/g,A2/g}`	⟪*offer*(`{A1,A2}`)⟫
`{A/g,A1/g,A2/g}`	`𝒫(A1),`	`{A/g,A1/g,A2/g}`	
`{A/g,A1/g,A2/g}`	`𝒫(A2).`	`{A/g,A1/g,A2/g}`	

This corresponds to the vanilla meta-interpreter underlying a large number of expert systems and found in Prolog textbooks such as [O'K90] and [SS86].

6.2.3 Reuse of Partially Specialised Constructs

The preparation of a clause- and mode-annotated argument slice from the initial sequence of abstract clause templates may require much effort and time from the expert programmer using our tool. However, the partial specialisations obtained as intermediate constructs in the preparation of the final argument slice can be reused to build different specialisations.

The history of the preparation of an argument slice is recorded by our tool as a tree in which the nodes are the actual constructs and the edges are the commands used to transform the upper node into its descendant node. This scheme is similar to the hierarchy $\mathcal{H}$ of argument slices described in Section 5.5: however, the constructions obtained during the design activity are of an even more abstract nature, and can be seen as comprising an extra layer of less committed templates. When the user is able to carry out further specialisations employing δ-relations, that is, those requirements listed at the beginning of Section 6.2.2 are fulfilled, then a most abstract argument slice of $\mathcal{H}$ is defined.

The history of the design of argument slices is also stored in our library, and the user is able to browse through it in a similar way as in the hierarchy $\mathcal{H}$. The user is also allowed to reuse any partially specified construct and try different alternatives in its specialisation. This ability saves the effort of having to define similar constructions from the very beginning every time, enabling the user to make experiments, and relate argument slices. For instance, the abstract template obtained after the definitions of the number of clauses and their recursive calls in our example above, can be used to devise an argument slice to decompose a list of lists, employed, for instance, in the procedure to "flatten" a nested list,

	$\mathcal{P}$(A):-	{A/g}	⟨⟨*offer*({A})⟩⟩
{A/g}	A = [].	{A/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f,A2/f}	⟨⟨*offer*({A})⟩⟩
{A/g,A1/f,A2/f}	A = [A1|A2],	{A/g,A1/g,A2/g}	⟨⟨*offer*({A1,A2})⟩⟩
{A/g,A1/g,A2/g}	atom(A1),	{A/g,A1/g,A2/g}	
{A/g,A1/g,A2/g}	$\mathcal{P}$(A2).	{A/g,A1/g,A2/g}	
	$\mathcal{P}$(A):-	{A/g,A1/f,A2/f}	⟨⟨*offer*({A})⟩⟩
{A/g,A1/f,A2/f}	A = [A1|A2],	{A/g,A1/g,A2/g}	⟨⟨*offer*({A1,A2})⟩⟩
{A/g,A1/g,A2/g}	$\mathcal{P}$(A1),	{A/g,A1/g,A2/g}	
{A/g,A1/g,A2/g}	$\mathcal{P}$(A2).	{A/g,A1/g,A2/g}	

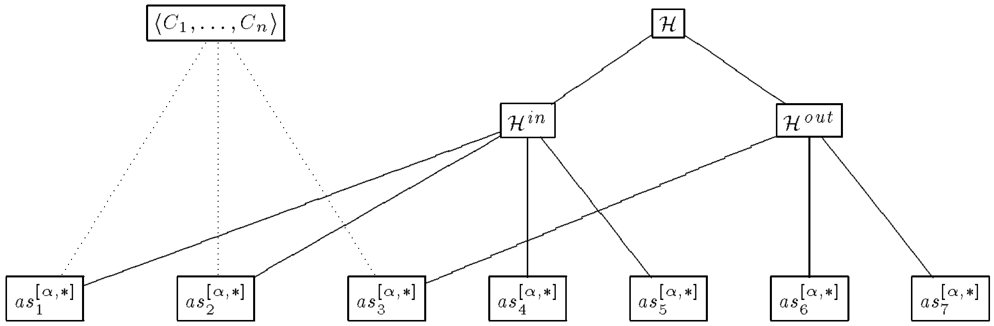
The user may browse through the tree until the appropriate node, depending on his needs, is found. New design decisions can be pursued from that node, possibly generating new constructs. At any moment the user can stop the current development activity and continue it later or use a previous node and try an alternative command.

Our tool supports the manipulation of these partially specified constructions, providing a means to store them employing a tree scheme which conveys their relationships. The user is encouraged to try different alternatives and compare their results by checking their relationships.

6.2.4 Relationship between the Hierarchies of Designed and Extracted Argument Slices

The hierarchy $\mathcal{H}$ of extracted argument slices and the hierarchy of designed argument slices relate to each other in a straightforward fashion: the bottom-most nodes of the latter are the top-most nodes of the subtrees $\mathcal{H}^{in}$ and $\mathcal{H}^{out}$ of $\mathcal{H}$. The hierarchy of designed argument slices is such that its most specific nodes are clause- and mode-annotated argument slices in the format of the most abtract argument slices of $\mathcal{H}$, that is the $as_i^{[\alpha,*]}$.

This relationship can be graphically depicted by the diagram below. The dotted lines stand for the sequence of commands defining clause- and mode-annotated argument slices from the initial very general template $\langle C_1, \ldots, C_n \rangle$.



Once a clause- and mode-annotated argument slice is defined, it can be appropriately inserted in $\mathcal{H}$ if so the user wishes. The commands offered during the initial stage of the design can be seen as higher-level δ-relations mapping the argument slices $as_i^{[\alpha,*]}$ into the initial sequence of abstract clauses template. The distinction between designed and extracted argument slices, however, is not essential but provides a natural manner to organise a large search-space.

The commands used to design the abstract argument slices $as_i^{[\alpha,*]}$, that is, the transformations chosen by the user to map the initial generic sequence of abstract clause templates into a clause- and mode-annotated argument slice, and represented in the diagram above by the dotted lines are:

1. *Number of Clauses* — the user must initially supply the number of abstract clause templates the argument slice will have. Clauses can be replicated later on using more specialised versions, though.

2. *Number of Recursive Calls* — each abstract clause template must have its number of recursive calls specified by the user. The user is requested to supply these values and only after each clause has its number of recursive calls specified is that the design process is able to continue.

3. *Contents of Sets of Variables* — the sets of variables appearing in each *relation* subgoals must be provided by the user. It is required that the variables appearing in the head goal and recursive calls appear in at least one of these sets; if that is not the case then an error message is displayed and the user is asked for another definition.

4. *Purpose of the Argument Slice* — argument slices can be input or output, depending on their intended usage in actual programs (see Section 5.5.2); it is essential that this information be supplied so as to allow the definition of mode-annotations to be suported and checked for mistakes.

5. *Contents of Mode-Annotations* — after the purpose of the argument slice is informed, initial mode-annotations are inserted into the template and the user may alter them to more accurate versions in which more precise tokens are used. The system is able to detect and reject incorrect mode-annotations, in which impossible changes in the mode of variables take place, *e.g.* a ground variable becoming free.

There is a strict ordering in the application of these commands: the design must follow the order of the commands given above.

6.2.5 Redesigning Argument Slices via Predicates $\Re^p$ and $\Re^*$

The clause- and mode-annotated argument slices which meet the requirements listed at the beginning of Section 6.2.2 can be redesigned via predicates $\Re^p$ and $\Re^*$ introduced in Section 5.7.6. As pointed out upon its definition, $\Re^p$ and $\Re^*$ relates argument slices which differ in their levels of abstraction, that is, $\Re^*(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_i^\alpha, \mathsf{R})$ holds if $\widehat{\widehat{P}}_i$ is an instance of $\widehat{\widehat{P}}_i^\alpha$ whose design history is recorded as sequence of r-terms in R, and, similarly, $\Re^p(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_i^\alpha, \mathsf{R})$ holds if $\widehat{\widehat{P}}_i$ is an instance of $\widehat{\widehat{P}}_i^\alpha$ and R is the sequence of r-terms depicting the mapping between these components. Depending on the content of the arguments submitted to $\Re^p$ and $\Re^*$, they may perform the automatic or user-assisted abstraction or the user-assisted redesign (reimplementation). In the redesign process, it is not important to distinguish between the δ-relations as they were grouped in $\Re^p$ and $\Re^*$: we can, in fact, devise a new predicate similar to $\Re^p$ and $\Re^*$ but employing any of the δ-relations. It is fundamental, though, that we employ the history of transformations R as a means to guide the specialisation of abstract predicate descriptors.

The redesign of an argument slice $\widehat{\widehat{P}}^\alpha$ is carried out by having the user supplying the r-terms of the design history. The participation of a user is fundamental during this task, since certain constructions have an infinite number of specialisations and only the user can rule out the choices. At any point during the specialisation of an argument slice, the user supplies the clause and the subgoal number, and the applicable δ-relations can be shown; the user chooses which one to apply.

For example, given the following abstract clause- and mode-annotated output argument slice $\widehat{\widehat{P}}_i^\alpha$, designed via the commands described above

	$\mathcal{P}$(A):-	{A/f}	
{A/f}	*relation*({A}).	{A/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$
	$\mathcal{P}$(A):-	{A/f,A1/f,A2/?}	
{A/f,A1/f,A2/?}	$\mathcal{P}$(A1),	{A/f,A1/g,A2/?}	$\langle\!\langle$*offer*({A1})$\rangle\!\rangle$
$\langle\!\langle$*required*({A2})$\rangle\!\rangle$ {A/f,A1/g,A2/g}	*relation*({A,A1,A2}).	{A/g,A1/g,A2/g}	$\langle\!\langle$*offer*({A})$\rangle\!\rangle$

then the user can use $\Re^p$ and $\Re^*$ to specialise it, provided that the sequence R representing the history of the design decisions is supplied. If the user supplies R as

$$\langle subs(\boxed{\gamma\ \theta\ relation(\texttt{\{A\}})\ \theta'\ \gamma'}, \boxed{\gamma\ \theta\ assign(\emptyset,\texttt{A})\ \theta'\ \gamma'}, 1, 1),$$

$$subs(\boxed{\gamma\ \theta\ assign(\emptyset,\texttt{A})\ \theta'\ \gamma'}, \boxed{\gamma\ \theta\ \texttt{A = []}\ \theta'\ \gamma'}, 1, 1),$$

$$subs(\boxed{\gamma\,\theta\,relation(\{\texttt{A,A1,A2}\})\,\theta'\,\gamma'}, \boxed{\gamma\,\theta\,build(\{\texttt{A1,A2}\}\texttt{,A})\,\theta'\,\gamma'}, 2, 2),$$

$$subs(\boxed{\gamma\,\theta\,build(\{\texttt{A1,A2}\}\texttt{,A})\,\theta'\,\gamma'}, \boxed{\gamma\,\theta\,\texttt{A = [A2|A1]}\,\theta'\,\gamma'}, 2, 2)\rangle$$

The following argument slice to build a list is obtained

		$\mathcal{P}$`(A):-`	`{A/f}`	
	`{A/f}`	`A = [].`	`{A/g}`	$\langle\!\langle$*offer*`({A})`$\rangle\!\rangle$
		$\mathcal{P}$`(A):-`	`{A/f,A1/f,A2/?}`	
	`{A/f,A1/f,A2/?}`	$\mathcal{P}$`(A1),`	`{A/f,A1/g,A2/?}`	$\langle\!\langle$*offer*`({A1})`$\rangle\!\rangle$
$\langle\!\langle$*required*`({A2})`$\rangle\!\rangle$	`{A/f,A1/g,A2/g}`	`A = [A2|A1].`	`{A/g,A1/g,A2/g}`	$\langle\!\langle$*offer*`({A})`$\rangle\!\rangle$

If, alternatively, the user supplies R as

$$\langle subs(\boxed{\gamma\,\theta\,relation(\{\texttt{A}\})\,\theta'\,\gamma'}, \boxed{\gamma\,\theta\,assign(\emptyset\texttt{,A})\,\theta'\,\gamma'}, 1, 1),$$

$$subs(\boxed{\gamma\,\theta\,assign(\emptyset\texttt{,A})\,\theta'\,\gamma'}, \boxed{\gamma\,\theta\,\texttt{A = 0}\,\theta'\,\gamma'}, 1, 1),$$

$$subs(\boxed{\gamma\,\theta\,relation(\{\texttt{A,A1,A2}\})\,\theta'\,\gamma'}, \boxed{\gamma\,\theta\,calc(\{\texttt{A1,A2}\}\texttt{,A})\,\theta'\,\gamma'}, 2, 2),$$

$$subs(\boxed{\gamma\,\theta\,calc(\{\texttt{A1,A2}\}\texttt{,A})\,\theta'\,\gamma'}, \boxed{\gamma\,\theta\,\texttt{A is A2 + A1}\,\theta'\,\gamma'}, 2, 2)\rangle$$

The following argument slice to sum a sequence of elements supplied in a loop is obtained

		$\mathcal{P}$`(A):-`	`{A/f}`	
	`{A/f}`	`A = 0.`	`{A/g}`	$\langle\!\langle$*offer*`({A})`$\rangle\!\rangle$
		$\mathcal{P}$`(A):-`	`{A/f,A1/f,A2/?}`	
	`{A/f,A1/f,A2/?}`	$\mathcal{P}$`(A1),`	`{A/f,A1/g,A2/?}`	$\langle\!\langle$*offer*`({A1})`$\rangle\!\rangle$
$\langle\!\langle$*required*`({A2})`$\rangle\!\rangle$	`{A/f,A1/g,A2/g}`	`A is A2 + A1.`	`{A/g,A1/g,A2/g}`	$\langle\!\langle$*offer*`({A})`$\rangle\!\rangle$

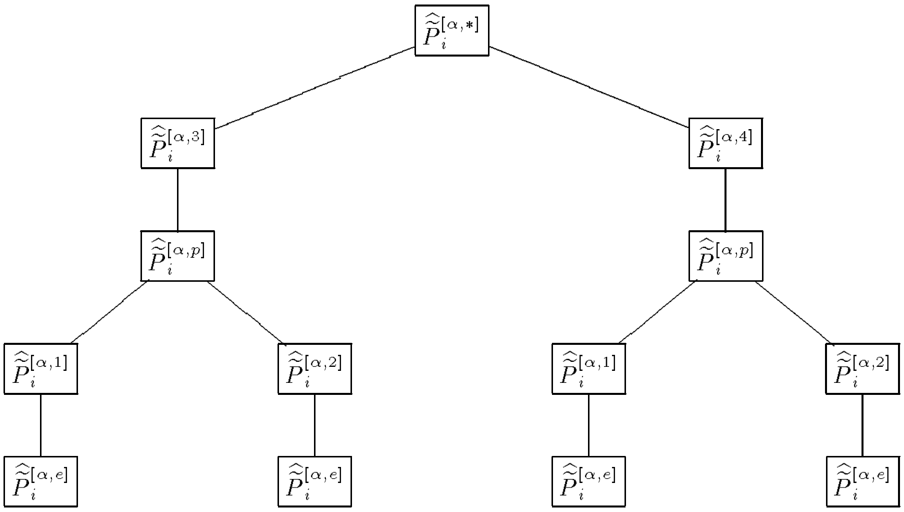
The sequence R of transformations is obtained interactively, the user being offered the possible instantiations, that is, all the δ-relations applicable at that point. As illustrated above, the same abstract constructions may yield different specialisations, depending on the choice of transformations of R. The initial argument slice, the sequence R and the intermediate representations are all stored as a tree, similar to $\mathcal{H}$, which the user can browse through and inspect the nodes and edges. The process of specialisation can use any argument slice with a predicate descriptor as an abstract construction and have its predicate descriptors defined.

A clause- and mode-annotated argument slice with at least one predicate descriptor can be seen as a higher-level specification: its descriptors stand for generic shorthands awaiting further specialisation. Support is given during the human-assisted reimplementation, wrong design choices being ruled out. For instance, clauses are only allowed to be copied if they have at least one predicate descriptor, since a clause without descriptors cannot be differentiated from its copies and it does not make sense to have identical clauses in a procedure.

6.2.6 Reusing Extracted Argument Slices

The clause- and mode-annotated argument slices of $\mathcal{H}$, extracted from actual procedures, can also be redesigned in the same fashion as those argument slices designed via abstract clause templates. Any node of $\mathcal{H}$ with at least a predicate descriptor can be used as an abstract construct and have its predicate descriptors reimplemented in a different way.

The new design history (R and intermediate forms) becomes part of $\mathcal{H}$ and its intermediate abstract forms may, on their turn, be used to define new constructs. Predicates $\Re^p$ and $\Re^*$ are also employed for this purpose, the r-terms being interactively provided by the user. For instance, the following subtree



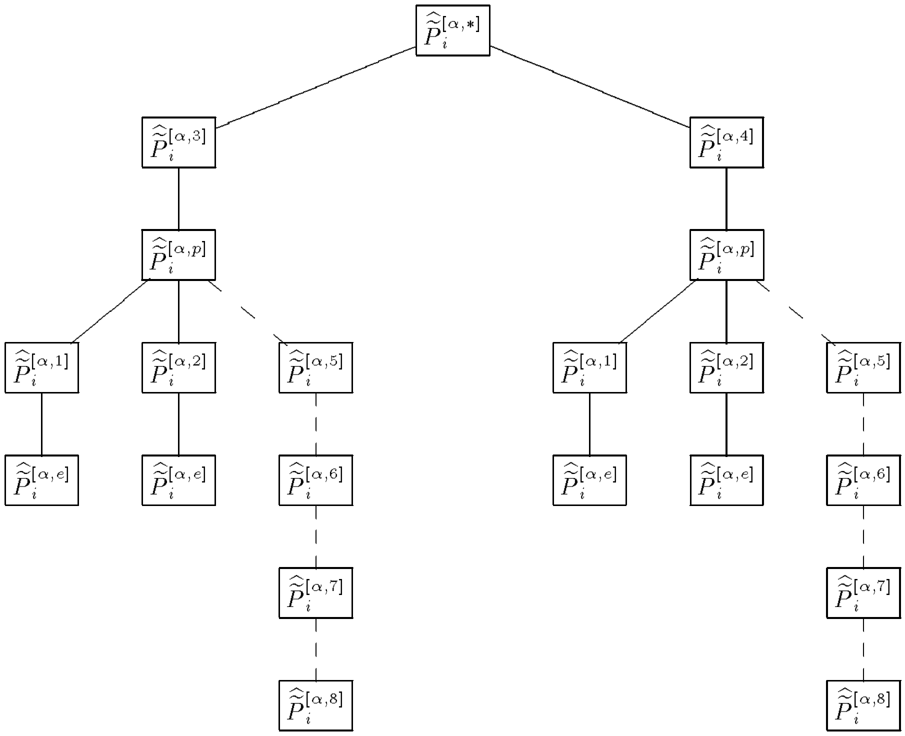
shown in Section 5.7.7 with the strategic abstractions of an extracted clause- and mode-annotated argument slice can be browsed through, and if the user decides to redesign the procedural abstraction $\widehat{\widehat{P}}_i^{[\alpha,p]}$, that is,

$$\widehat{\widehat{P}}_i^{[\alpha,p]} = \boxed{\begin{array}{lll} & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^1\ \gamma_0^1 \\ \gamma_1^1\ \theta_1^1 & \quad test(\{\mathtt{A}\})\mathtt{.} & \theta_1^{1\prime}\ \gamma_1^{1\prime} \\ & \mathcal{P}(\mathtt{A})\mathtt{:-} & \theta_0^2\ \gamma_0^2 \\ \gamma_1^2\ \theta_1^2 & \quad decomp(\mathtt{A},\{\mathtt{D},\mathtt{E}\})\mathtt{,} & \theta_1^{2\prime}\ \gamma_1^{2\prime} \\ \gamma_2^2\ \theta_2^2 & \quad \mathcal{P}(\mathtt{E})\mathtt{.} & \theta_2^{2\prime}\ \gamma_2^{2\prime} \end{array}}$$

and redesigned the predicate descriptors by interactively supplying the following sequence R of transformations

- $r_5 = subs(\boxed{\gamma_1^1\,\theta_1^1\ test(\{\mathtt{A}\});\theta_1^{1'}\,\gamma_1^{1'}} / \boxed{\gamma_1^1\,\theta_1^1\ \mathtt{A\ =\ true}\ \theta_1^{1'}\,\gamma_1^{1'}}, 1, 1);$
- $r_6 = copy(\boxed{\mathcal{P}\mathtt{(A)\!:\!-}\ \theta_0^2\,\gamma_0^2\ \gamma_1^2\,\theta_1^2\ decomp(\mathtt{A},\{\mathtt{D},\mathtt{E}\})\ \theta_1^{2'}\,\gamma_1^{2'},\ \gamma_2^2\,\theta_2^2\ \mathcal{P}\mathtt{(E)}\ \theta_2^{2'}\,\gamma_2^{2'}.}, 2);$
- $r_7 = subs(\boxed{\gamma_1^2\,\theta_1^2\ decomp(\mathtt{A},\{\mathtt{D},\mathtt{E}\})\ \theta_1^{2'}\,\gamma_1^{2'}} / \boxed{\gamma_1^2\,\theta_1^2\ \mathtt{A\ =\ D\ or\ E}\ \theta_1^{2'}\,\gamma_1^{2'}}, 2, 1);$
- $r_8 = subs(\boxed{\gamma_1^2\,\theta_1^2\ decomp(\mathtt{A},\{\mathtt{D},\mathtt{E}\})\ \theta_1^{2'}\,\gamma_1^{2'}} / \boxed{\gamma_1^2\,\theta_1^2\ \mathtt{A\ =\ E\ or\ D}\ \theta_1^{2'}\,\gamma_1^{2'}}, 2, 1);$

The subtree would be updated with the new reengineering of $\widehat{\widehat{P}}_i^{[\alpha,4p}$ (shown with dashed lines), thus yielding



where the bottom-most argument slice is of the form

$$\widehat{\widehat{P}}_i^{[\alpha,8]} = \boxed{\begin{array}{lll} & \mathcal{P}\mathtt{(A)\!:\!-} & \theta_0^1\ \gamma_0^1 \\ \gamma_1^1\ \theta_1^1 & \quad\mathtt{A\ =\ true.} & \theta_1^{1'}\ \gamma_1^{1'} \\ & \mathcal{P}\mathtt{(A)\!:\!-} & \theta_0^2\ \gamma_0^2 \\ \gamma_1^2\ \theta_1^2 & \quad\mathtt{A\ =\ D\ or\ E,} & \theta_1^{2'}\ \gamma_1^{2'} \\ \gamma_2^2\ \theta_2^2 & \quad\mathcal{P}\mathtt{(E).} & \theta_2^{2'}\ \gamma_2^{2'} \\ & \mathcal{P}\mathtt{(A)\!:\!-} & \theta_0^2\ \gamma_0^2 \\ \gamma_1^2\ \theta_1^2 & \quad\mathtt{A\ =\ E\ or\ D,} & \theta_1^{2'}\ \gamma_1^{2'} \\ \gamma_2^2\ \theta_2^2 & \quad\mathcal{P}\mathtt{(E).} & \theta_2^{2'}\ \gamma_2^{2'} \end{array}}$$

that is, an argument slice testing if a disjunction has a "`true`" in it, in which case the disjunction can be said to hold.

6.3 Combining Argument Slices to Define Techniques

A programming technique is defined when a sequence of clause- and mode-annotated argument slices is properly assembled and is such that all those required variable symbols are bound to offered variables of other argument slices. Our tool supports the definition of techniques by enabling the user to choose argument slices and bind their required and offered variable symbols.

The argument slices chosen to define a technique must all share basic syntactic features: they must have the same number of clauses and corresponding clauses with the same number of recursive calls, as formalised in Section 6.3.1. The adopted view of programming technique is such that a programming technique is either manipulating a value into another form or obtaining a value (or values) as the computation proceeds; this is used as another restriction: any programming technique must have at least one input argument slice. The input argument slices establish the flow of control and provide value(s) for the computations performed by the output argument slices in order to provide a final result.

Expert programmers define their programming techniques by assembling a non-empty sequence of argument slices. The requirements of syntactic compatibility between the argument slices and of at least one input argument slice are reinforced during the preparation of techniques, erroneous choices being rejected. Upon the user's choice of a sequence of argument slices, our tool shows the sequence of argument slices appropriately combined, as explained in Section 6.3.2 below. By showing the argument slices combined together, a clearer picture of the programming technique arises. Given a pair of compatible clause- and mode-annotated argument slices, their appropriate combination can be obtained by describing how each clause is to be combined.

For instance, if the user choses the argument slices $\hat{\hat{R}}_1$ from Section 5.10.1, that is,

	$\mathcal{P}$(A):-	{A/g}	$\langle\langle offer(\{A\})\rangle\rangle$
{A/g}	destiny(A).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,D/f,E/f}	$\langle\langle offer(\{A\})\rangle\rangle$
{A/g,D/f,E/f}	A to E costs D,	{A/g,D/g,E/g}	$\langle\langle offer(\{D,E\})\rangle\rangle$
{A/g,D/g,E/g}	$\mathcal{P}$(E).	{A/g,D/g,E/g}	

and the first argument slice devised in Section 6.2.5, that is,

<table>
<tr><td></td><td></td><td>$\mathcal{P}$(A):-</td><td>{A/f}</td><td></td></tr>
<tr><td></td><td>{A/f}</td><td>A = [].</td><td>{A/g}</td><td>⟨⟨offer({A})⟩⟩</td></tr>
<tr><td></td><td></td><td>$\mathcal{P}$(A):-</td><td>{A/f,A1/f,A2/?}</td><td></td></tr>
<tr><td></td><td>{A/f,A1/f,A2/?}</td><td>$\mathcal{P}$(A1),</td><td>{A/f,A1/g,A2/?}</td><td>⟨⟨offer({A1})⟩⟩</td></tr>
<tr><td>⟨⟨required({A2})⟩⟩</td><td>{A/f,A1/g,A2/g}</td><td>A = [A2|A1].</td><td>{A/g,A1/g,A2/g}</td><td>⟨⟨offer({A})⟩⟩</td></tr>
</table>

then the following combination would be displayed

<table>
<tr><td></td><td></td><td>$\mathcal{P}$(A,B):-</td><td>$\theta^{\langle l,r\rangle}_{[1,0]}$</td><td>⟨⟨offer({A})⟩⟩</td></tr>
<tr><td></td><td>$\theta^{\langle l,r\rangle}_{[1,1]}$</td><td>destiny(A),</td><td>$\theta^{\langle l,r\rangle'}_{[1,1]}$</td><td></td></tr>
<tr><td></td><td>$\theta^{\langle l,r\rangle}_{[1,2]}$</td><td>B = [].</td><td>$\theta^{\langle l,r\rangle'}_{[1,2]}$</td><td>⟨⟨offer({B})⟩⟩</td></tr>
<tr><td></td><td></td><td>$\mathcal{P}$(A,F):-</td><td>$\theta^{\langle l,r\rangle}_{[2,0]}$</td><td>⟨⟨offer({A})⟩⟩</td></tr>
<tr><td></td><td>$\theta^{\langle l,r\rangle}_{[2,1]}$</td><td>A to E costs D,</td><td>$\theta^{\langle l,r\rangle'}_{[2,1]}$</td><td>⟨⟨offer({D,E})⟩⟩</td></tr>
<tr><td></td><td>$\theta^{\langle l,r\rangle}_{[2,2]}$</td><td>$\mathcal{P}$(E,G),</td><td>$\theta^{\langle l,r\rangle'}_{[2,2]}$</td><td>⟨⟨offer({G})⟩⟩</td></tr>
<tr><td>⟨⟨required({H})⟩⟩</td><td>$\theta^{\langle l,r\rangle}_{[2,3]}$</td><td>F = [H|G].</td><td>$\theta^{\langle l,r\rangle'}_{[2,3]}$</td><td>⟨⟨offer({F})⟩⟩</td></tr>
</table>

The variables have to be renamed to avoid name clashes and accidental bindings between them. The clause-annotations are maintained in the original position relative to the subgoals of the argument slices they come from and the mode-annotations $\theta^{\langle l,r\rangle}_{[i,j]}$ and $\theta^{\langle l,r\rangle'}_{[i,j]}$ are the appropriate combination of the mode-annotations of the original argument slices. In Section 6.3.2 a formalisation of the combination process is given.

After the choice of argument slices is shown appropriately combined, the user may proceed to link the required and offered variable symbols. If the variable symbol being offered comes after the variable symbol being requested, then depending on the nature of the subgoal which requires the variable, the link may be considered incorrect and rejected. If a variable in the right side of an *is/2* subgoal is required, then it should only be linked to variables offered before it. In our current example, there is only one required variable to be linked; if the user decides to link it to offered variable symbol `D` then a technique to build a list with the associated costs of a graph is defined:

<table>
<tr><td></td><td>$\mathcal{P}$(A,B):-</td><td>$\theta^{\langle l,r\rangle}_{[1,0]}$</td><td>⟨⟨offer({A})⟩⟩</td></tr>
<tr><td>$\theta^{\langle l,r\rangle}_{[1,1]}$</td><td>destiny(A),</td><td>$\theta^{\langle l,r\rangle'}_{[1,1]}$</td><td></td></tr>
<tr><td>$\theta^{\langle l,r\rangle}_{[1,2]}$</td><td>B = [].</td><td>$\theta^{\langle l,r\rangle'}_{[1,2]}$</td><td>⟨⟨offer({B})⟩⟩</td></tr>
<tr><td></td><td>$\mathcal{P}$(A,F):-</td><td>$\theta^{\langle l,r\rangle}_{[2,0]}$</td><td>⟨⟨offer({A})⟩⟩</td></tr>
<tr><td>$\theta^{\langle l,r\rangle}_{[2,1]}$</td><td>A to E costs D,</td><td>$\theta^{\langle l,r\rangle'}_{[2,1]}$</td><td>⟨⟨offer({D,E})⟩⟩</td></tr>
<tr><td>$\theta^{\langle l,r\rangle}_{[2,2]}$</td><td>$\mathcal{P}$(E,G),</td><td>$\theta^{\langle l,r\rangle'}_{[2,2]}$</td><td>⟨⟨offer({G})⟩⟩</td></tr>
<tr><td>$\theta^{\langle l,r\rangle}_{[2,3]}$</td><td>F = [D|G].</td><td>$\theta^{\langle l,r\rangle'}_{[2,3]}$</td><td>⟨⟨offer({F})⟩⟩</td></tr>
</table>

If variable `A` is used instead, a technique to build a list with the names of the visited nodes (excluding the final destination node) is devised:

	$\mathcal{P}$(A,B):-	$\theta^{\langle l,r\rangle}_{[1,0]}$	$\langle\langle \mathit{offer}(\{\texttt{A}\})\rangle\rangle$
$\theta^{\langle l,r\rangle}_{[1,1]}$	destiny(A),	$\theta^{\langle l,r\rangle'}_{[1,1]}$	
$\theta^{\langle l,r\rangle}_{[1,2]}$	B = [].	$\theta^{\langle l,r\rangle'}_{[1,2]}$	$\langle\langle \mathit{offer}(\{\texttt{B}\})\rangle\rangle$
	$\mathcal{P}$(A,F):-	$\theta^{\langle l,r\rangle}_{[2,0]}$	$\langle\langle \mathit{offer}(\{\texttt{A}\})\rangle\rangle$
$\theta^{\langle l,r\rangle}_{[2,1]}$	A to E costs D,	$\theta^{\langle l,r\rangle'}_{[2,1]}$	$\langle\langle \mathit{offer}(\{\texttt{D},\texttt{E}\})\rangle\rangle$
$\theta^{\langle l,r\rangle}_{[2,2]}$	$\mathcal{P}$(E,G),	$\theta^{\langle l,r\rangle'}_{[2,2]}$	$\langle\langle \mathit{offer}(\{\texttt{G}\})\rangle\rangle$
$\theta^{\langle l,r\rangle}_{[2,3]}$	F = [A|G].	$\theta^{\langle l,r\rangle'}_{[2,3]}$	$\langle\langle \mathit{offer}(\{\texttt{F}\})\rangle\rangle$

Another restriction in the linking of variables is that variables are only allowed to be linked if they are from different argument slices. In our current example, `H` could not be linked to `G` or `F`.

The preparation of techniques consists of having the user specifying his choice of argument slices: they can be supplied one at a time, and the user is presented with their appropriate combination; other argument slices can be further added and combined with the existing construct. Incompatible argument slices are not accepted for combination: a message is issued and another argument slice can be provided. Argument slices with more than one possible way to combine its clauses require user intervention: the user must inform the system how the combination is to be performed. If there is at least one variable symbol being required then the user can link it to other variable symbols offered in the construct.

The links between argument slices have the following restrictions:

1. a variable in an *offer* annotation can only be linked to a variable in a *required* annotation and vice-versa;
2. only variables from different argument slices can be linked, that is, a variable in a *required* annotation can only be linked to a variable in an *offer* annotation from a different argument slice and vice-versa;
3. variables must be linked within each clause;
4. required variables appearing in the right-hand side of *is/2* subgoals or in a *user-pred* subgoal can only be linked with variables complying with the previous restrictions and such that they appear *before* (with respect to Prolog's left-to-right subgoal execution) the subgoal requiring the variable(s).

6.3.1 Compatibility of Argument Slices

Given two clause- and mode-annotated argument slices $\widehat{\widehat{P}}_i = \langle \widehat{\widehat{C}}{}^i_1, \ldots, \widehat{\widehat{C}}{}^i_n \rangle$ and $\widehat{\widehat{P}}_j = \langle \widehat{\widehat{C}}{}^j_1, \ldots, \widehat{\widehat{C}}{}^j_n \rangle$ they are said to be *compatible*, denoted by $comp^{\widehat{\widehat{P}}}(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_j)$ if, an only if, for any clause $\widehat{\widehat{C}}{}^i_k$ in $\widehat{\widehat{P}}_i$ there is a compatible clause $\widehat{\widehat{C}}{}^j_m$ in $\widehat{\widehat{P}}_j$, that is,

$$
\begin{array}{rcl}
comp^{\widehat{\widehat{P}}}(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_j) & \Longleftarrow & \widehat{\widehat{P}}_i = \langle \vec{C}^i_1 \, \widehat{\widehat{C}}{}^i_k \, \vec{C}^i_2 \rangle \wedge \widehat{\widehat{P}}_j = \langle \vec{C}^j_1 \, \widehat{\widehat{C}}{}^j_m \, \vec{C}^j_2 \rangle \wedge comp^{\widehat{\widehat{C}}}(\widehat{\widehat{C}}{}^i_k, \widehat{\widehat{C}}{}^j_m) \wedge \\
 & & comp^{\widehat{\widehat{P}}}(\langle \vec{C}^i_1 \, \vec{C}^i_2 \rangle, \langle \vec{C}^j_1 \, \vec{C}^j_2 \rangle) \\
comp^{\widehat{\widehat{P}}}(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_j) & \Longleftarrow & \widehat{\widehat{P}}_i = \langle \, \rangle \wedge \widehat{\widehat{P}}_j = \langle \, \rangle
\end{array}
$$

The above definition represents a declarative formulation to test the syntactic compatibility of two argument slices. It relies on the compatibility of clause- and mode-annotated clauses as defined below. Its general idea is that of finding a clause in each argument slice such that they are compatible, and testing what is left of each argument slice once the clauses are removed from them (first case above); the base case is provided by the second case: two argument slices are compatible if they are empty.

The number of clauses of compatible argument slices is required to be the same, by the definition above. The definition also states that for each clause in $\widehat{\widehat{P}}_i$ there must be a compatible clause in $\widehat{\widehat{P}}_j$. Hence, it is the case that $comp^{\widehat{\widehat{P}}}$ is symmetric, that is, if $comp^{\widehat{\widehat{P}}}(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_j)$ is true then $comp^{\widehat{\widehat{P}}}(\widehat{\widehat{P}}_j, \widehat{\widehat{P}}_i)$ is also true and vice-versa.

The compatibility of clauses, $comp^{\widehat{\widehat{C}}}$, is defined using generic templates. Recursive clauses are compatible if they have the same number of recursive clauses, that is,

$$
comp^{\widehat{\widehat{C}}}\left(
\boxed{\begin{array}{lll}
 & p(x^i)\texttt{:-} & \theta^i_0 \;\; \gamma^i_0 \\
 & \vec{S}^i_0 & \\
\theta^i_1 \;\; \gamma^i_1 & p(x^i_1)\texttt{,} & \theta^{i\,\prime}_1 \;\; \gamma^{i\,\prime}_1 \\
 & \vec{S}^i_1 & \\
 & \vdots & \\
 & \vec{S}^i_{n-1} & \\
\theta^i_n \;\; \gamma^i_n & p(x^i_n)\texttt{,} & \theta^{i\,\prime}_n \;\; \gamma^{i\,\prime}_n \\
 & \vec{S}^i_n &
\end{array}}
\;,\;
\boxed{\begin{array}{lll}
 & p(x^j)\texttt{:-} & \theta^j_0 \;\; \gamma^j_0 \\
 & \vec{S}^j_0 & \\
\theta^j_1 \;\; \gamma^j_1 & p(x^j_1)\texttt{,} & \theta^{j\,\prime}_1 \;\; \gamma^{j\,\prime}_1 \\
 & \vec{S}^j_1 & \\
 & \vdots & \\
 & \vec{S}^j_{n-1} & \\
\theta^j_n \;\; \gamma^j_n & p(x^j_n)\texttt{,} & \theta^{j\,\prime}_n \;\; \gamma^{j\,\prime}_n \\
 & \vec{S}^j_n &
\end{array}}
\right)
$$

Any two non-recursive clauses are compatible, that is,

$$
comp^{\widehat{\widehat{C}}}\left(
\boxed{\begin{array}{c}
p(x^i)\texttt{:-}\; \theta^i_0 \;\; \gamma^i_0 \\
\vec{S}^i
\end{array}}
\;,\;
\boxed{\begin{array}{c}
p(x^j)\texttt{:-}\; \theta^j_0 \;\; \gamma^j_0 \\
\vec{S}^j
\end{array}}
\right)
$$

The relation $comp^{\widehat{\widehat{C}}}$ is also symmetric.

6.3.2 Combination of Argument Slices

The combination of two compatible clauses is performed by pairing off the recursive calls and producing for each pair a combined recursive call with the arguments of the original calls. When combining clauses a distinction should be made between the left- and right-hand side clauses, because the non-recursive subgoals are slotted in the combined version according to the order of the original clauses, the left-hand side subgoals (in their original relative ordering) coming before the right-hand side ones (also in their original ordering). The combination between two clause- and mode-annotated compatible clauses $\hat{\hat{C}}^i$, $i = l, r$, of form

$$\boxed{\begin{array}{lll} & p(x^i)\ \texttt{:-} & \theta_0^i\ \ \gamma_0^i \\ & \vec{S}_0^i & \\ \theta_1^i\ \ \gamma_1^i & p(x_1^i)\ , & {\theta_1^i}'\ \ {\gamma_1^i}' \\ & \vec{S}_1^i & \\ & \vdots & \\ & \vec{S}_{n-1}^i & \\ \theta_n^i\ \ \gamma_n^i & p(x_n^i)\ , & {\theta_n^i}'\ \ {\gamma_n^i}' \\ & \vec{S}_n^i & \end{array}}$$

is $\hat{\hat{C}}^{\langle l,r\rangle} = \hat{\hat{C}}^l \cdot \hat{\hat{C}}^r$, of the form

$$\boxed{\begin{array}{lll} & p(x^l, x^r)\ \texttt{:-} & \theta_0^{\langle l,r\rangle}\ \ \gamma_0^{\langle l,r\rangle} \\ & \vec{S}_0^{\langle l,r\rangle} & \\ \theta_1^{\langle l,r\rangle}\ \ \gamma_1^{\langle l,r\rangle} & p(x_1^l, x_1^r)\ , & {\theta_1^{\langle l,r\rangle}}'\ \ {\gamma_1^{\langle l,r\rangle}}' \\ & \vec{S}_1^{\langle l,r\rangle} & \\ & \vdots & \\ & \vec{S}_{n-1}^{\langle l,r\rangle} & \\ \theta_n^{\langle l,r\rangle}\ \ \gamma_n^{\langle l,r\rangle} & p(x_n^l, x_n^r)\ , & {\theta_n^{\langle l,r\rangle}}'\ \ {\gamma_n^{\langle l,r\rangle}}' \\ & \vec{S}_n^{\langle l,r\rangle} & \end{array}}$$

or if $\hat{\hat{C}}^i$, $i = l, r$, are of the form

$$\boxed{\begin{array}{l} p(x^i)\ \texttt{:-}\ \theta_0^i\ \ \gamma_0^i \\ \quad \vec{S}^i \end{array}}$$

then $\hat{\hat{C}}^{\langle l,r\rangle} = \hat{\hat{C}}^l \cdot \hat{\hat{C}}^r$, is of the form

$$\boxed{\begin{array}{l} p(x^l, x^r)\ \texttt{:-}\ \theta_0^{\langle l,r\rangle}\ \ \gamma_0^{\langle l,r\rangle} \\ \quad \vec{S}^{\langle l,r\rangle} \end{array}}$$

where $\theta_0^{\langle l,r\rangle} = \theta_0^l \cup \theta_0^r$ and $\gamma_0^{\langle l,r\rangle} = \langle\!\langle \mathit{offer}(W^{\langle l,r\rangle}) \rangle\!\rangle$, $W^{\langle l,r\rangle} = W^l \cup W^r$, $\gamma_0^i = \langle\!\langle \mathit{offer}(W^i) \rangle\!\rangle$, $i = l, r$, that is, the mode-annotations of the head goal and recursive subgoals are merged together and the offered variables of each clause are merged in a single set of an *offer* clause-annotation in their combined clause.

The clause- and mode-annotations of the new combined clause are defined in terms of the existing annotations of the left- and right-hand side clauses. For each pair of vectors $\vec{S}^i_j, i = l, r, 0 \leq j \leq n$ of the form

$$\vec{S}^i_j = \boxed{\begin{array}{ccc} \gamma^i_{[j,0]}\ \theta^i_{[j,0]} & S^i_{[j,0]}, & {\theta^i_{[j,0]}}'\ {\gamma^i_{[j,0]}}' \\ & \vdots & \\ \gamma^i_{[j,m_j]}\ \theta^i_{[j,m_j]} & S^i_{[j,m_j]}, & {\theta^i_{[j,m_j]}}'\ {\gamma^i_{[j,m_j]}}' \end{array}}$$

where m_j is the number of subgoals of vector $\vec{S}^i_j$, we have that

$$\vec{S}^{\langle l,r\rangle}_j = \boxed{\begin{array}{ccc} \gamma^l_{[j,0]}\ \theta^{l\cup r}_{[j,0]} & S^i_{[j,0]} & {\theta^{l\cup r}_{[j,0]}}'\ {\gamma^l_{[j,0]}}' \\ & \vdots & \\ \gamma^l_{[j,m_j]}\ \theta^{l\cup r}_{[j,m_j]} & S^i_{[j,m_j]} & {\theta^{l\cup r}_{[j,m_j]}}'\ {\gamma^l_{[j,m_j]}}' \\ \gamma^r_{[j,0]}\ \theta^{r\cup l}_{[j,0]} & S^r_{[j,0]} & {\theta^{r\cup l}_{[j,0]}}'\ {\gamma^r_{[j,0]}}' \\ & \vdots & \\ \gamma^r_{[j,m_j]}\ \theta^{r\cup l}_{[j,m_j]} & S^r_{[j,0]} & {\theta^{r\cup l}_{[j,m_j]}}'\ {\gamma^r_{[j,m_j]}}' \end{array}}$$

where

- $\theta^{l\cup r}_{[j,k]} = \theta^l_{[j,k]} \cup \theta^r_{[j,0]}$; ${\theta^{l\cup r}_{[j,k]}}' = {\theta^l_{[j,k]}}' \cup \theta^r_{[j,0]}$;
- $\theta^{r\cup l}_{[j,k]} = \theta^r_{[j,k]} \cup {\theta^l_{[j,m_j]}}'$; ${\theta^{r\cup l}_{[j,k]}}' = {\theta^r_{[j,k]}}' \cup {\theta^l_{[j,m_j]}}'$;

That is, the first mode-annotation of $\vec{S}^r$ is added to all the mode-annotations of $\vec{S}^l$ and the last mode-annotation of $\vec{S}^l$ is added to all the mode-annotations of $\vec{S}^r$. Whenever one of the vectors of clause- and mode-annotated subgoals is empty, then their combination is the non-empty vector. If both are empty, their combination is also empty.

Given two compatible argument slices $\hat{\hat{P}}_l = \langle \hat{\hat{C}}^l_1, \ldots, \hat{\hat{C}}^l_n \rangle$ and $\hat{\hat{P}}_r = \langle \hat{\hat{C}}^r_1, \ldots, \hat{\hat{C}}^r_n \rangle$, their combination $\hat{\hat{P}}_{\langle l,r\rangle} = \hat{\hat{P}}_l \cdot \hat{\hat{P}}_r$ is $\langle \hat{\hat{C}}^{\langle l,r\rangle}_1, \ldots, \hat{\hat{C}}^{\langle l,r\rangle}_n \rangle$ where $\hat{\hat{C}}^{\langle l,r\rangle}_i = \hat{\hat{C}}^l_i \cdot \hat{\hat{C}}^r_j$, $comp^{\hat{\hat{C}}}(\hat{\hat{C}}^l_i, \hat{\hat{C}}^r_j)$. An alternative constructive definition is

$$\hat{\hat{P}}_l \cdot \hat{\hat{P}}_r = \begin{cases} \langle\ \rangle & \textit{iff} \quad \hat{\hat{P}}_l = \langle\ \rangle \ \textit{and} \ \hat{\hat{P}}_r = \langle\ \rangle \\ \langle \hat{\hat{C}}^{\langle l,r\rangle}\ \vec{C}^{\langle l,r\rangle} \rangle & \textit{iff} \quad \hat{\hat{P}}_l = \langle \hat{\hat{C}}^l\ \vec{C}^l \rangle \ \textit{and} \ \hat{\hat{P}}_r = \langle \vec{C}^r_1\ \hat{\hat{C}}^r\ \vec{C}^r_2 \rangle \ \textit{and} \\ & \qquad \hat{\hat{C}}^{\langle l,r\rangle} = \hat{\hat{C}}^l \cdot \hat{\hat{C}}^r \ \textit{and} \ \vec{C}^{\langle l,r\rangle} = \langle \vec{C}^l \rangle \cdot \langle \vec{C}^l_1\ \vec{C}^l_2 \rangle \end{cases}$$

This definition views argument slices as sequences of clauses in order to formalise their combination. It defines their combination as the combination of the first clause of the left-hand side argument slice with a compatible clause of the right-hand side argument slice, and the rest of the combined argument slice is recursively defined. The base case depicts the combination of two empty argument slices as the empty sequence.

It should be noticed that a clause in an argument slice may have more than one compatible clause in another argument slice. For instance, if $\widehat{\widehat{P}}_i = \langle \widehat{\widehat{C}}{}^i_1, \widehat{\widehat{C}}{}^i_2, \widehat{\widehat{C}}{}^i_3 \rangle$ and $\widehat{\widehat{P}}_j = \langle \widehat{\widehat{C}}{}^j_1, \widehat{\widehat{C}}{}^j_2, \widehat{\widehat{C}}{}^j_3 \rangle$ and their clauses are such that $comp^{\widehat{\widehat{C}}}(\widehat{\widehat{C}}{}^i_1, \widehat{\widehat{C}}{}^j_1)$, $comp^{\widehat{\widehat{C}}}(\widehat{\widehat{C}}{}^i_2, \widehat{\widehat{C}}{}^j_2)$, $comp^{\widehat{\widehat{C}}}(\widehat{\widehat{C}}{}^i_2, \widehat{\widehat{C}}{}^j_3)$, then there is no single way to correctly combine them, unless there is a user intervention to tell the system which clause should be combined with which other clause.

6.3.3 Classes of Programming Techniques

The argument slices chosen to define a programming technique can also be abstract constructs with predicate descriptors instead of actual Prolog predicates. Classes of programming techniques can be defined by binding abstract argument slices: the abstract constructions stand for all the particular instances of them.

If, in the previous example, the user had chosen to use the following more abstract form of the output argument slice

		$\mathcal{P}$`(A):-`	`{A/f}`	
	`{A/f}`	*assign*`(∅,{A}).`	`{A/g}`	⟪*offer*`({A})`⟫
		$\mathcal{P}$`(A):-`	`{A/f,A1/f,A2/?}`	
	`{A/f,A1/f,A2/?}`	$\mathcal{P}$`(A1),`	`{A/f,A1/g,A2/?}`	⟪*offer*`({A1})`⟫
⟪*required*`({A2})`⟫	`{A/f,A1/g,A2/g}`	*relation*`({A2,A1},A).`	`{A/g,A1/g,A2/g}`	⟪*offer*`({A})`⟫

ending up with the following construct

	$\mathcal{P}$`(A,B):-`	$\theta^{\langle l,r\rangle}_{[1,0]}$	⟪*offer*`({A})`⟫
$\theta^{\langle l,r\rangle}_{[1,1]}$	`destiny(A),`	$\theta^{\langle l,r\rangle'}_{[1,1]}$	
$\theta^{\langle l,r\rangle}_{[1,2]}$	*assign*`(∅,{B}).`	$\theta^{\langle l,r\rangle'}_{[1,2]}$	⟪*offer*`({B})`⟫
	$\mathcal{P}$`(A,F):-`	$\theta^{\langle l,r\rangle}_{[2,0]}$	⟪*offer*`({A})`⟫
$\theta^{\langle l,r\rangle}_{[2,1]}$	`A to E costs D,`	$\theta^{\langle l,r\rangle'}_{[2,1]}$	⟪*offer*`({D,E})`⟫
$\theta^{\langle l,r\rangle}_{[2,2]}$	$\mathcal{P}$`(E,G),`	$\theta^{\langle l,r\rangle'}_{[2,2]}$	⟪*offer*`({G})`⟫
$\theta^{\langle l,r\rangle}_{[2,3]}$	*relation*`({D,G},F).`	$\theta^{\langle l,r\rangle'}_{[2,3]}$	⟪*offer*`({F})`⟫

This abstract construct can be specialised in different manners; if, for instance, the abstract predicate descriptor "*assign*`(∅,{B})`" is specified as "`B = []`" and the subgoal "*relation*`({D,G},F)`" as "`F = [D|G]`" then the construction obtained previously would be defined; if, alternatively, the user defines "*assign*`(∅,{B})`" as "`B = 0`" and "*relation*({D,G},F)" as "`F is D + G`" a different technique would be devised

	`𝒫(A,B):-`	$\theta^{\langle l,r\rangle}_{[1,0]}$	⟨⟨*offer*({A})⟩⟩
$\theta^{\langle l,r\rangle}_{[1,1]}$	`destiny(A),`	$\theta^{\langle l,r\rangle'}_{[1,1]}$	
$\theta^{\langle l,r\rangle}_{[1,2]}$	`B = 0.`	$\theta^{\langle l,r\rangle'}_{[1,2]}$	⟨⟨*offer*({B})⟩⟩
	`𝒫(A,F):-`	$\theta^{\langle l,r\rangle}_{[2,0]}$	⟨⟨*offer*({A})⟩⟩
$\theta^{\langle l,r\rangle}_{[2,1]}$	`A to E costs D,`	$\theta^{\langle l,r\rangle'}_{[2,1]}$	⟨⟨*offer*({D,E})⟩⟩
$\theta^{\langle l,r\rangle}_{[2,2]}$	`𝒫(E,G),`	$\theta^{\langle l,r\rangle'}_{[2,2]}$	⟨⟨*offer*({G})⟩⟩
$\theta^{\langle l,r\rangle}_{[2,3]}$	`F is D + G.`	$\theta^{\langle l,r\rangle'}_{[2,3]}$	⟨⟨*offer*({F})⟩⟩

The definition of programming techniques using more abstract argument slices allows whole classes of more specific constructs to be characterised. Any specialisations of the generic constructs automatically define more specific instances in that class of programming techniques. A generic formulation for a programming technique automatically supplies us with special instances of that programming technique through the specialisation of its argument slices.

The example below illustrates the similarities between two acclaimed programming techniques, the accumulator pair and the different list techniques [SS86, Ros89, O'K90] when a more abstract formulation is devised. If the following abstract argument slices

		`𝒫(A):-`	{A/g}	⟨⟨*offer*({A})⟩⟩
	{A/g}	*relation*(A).	{A/g}	
		`𝒫(A):-`	{A/g,B/f,C/f}	⟨⟨*offer*({A})⟩⟩
⟨⟨*required*({B})⟩⟩	{A/g,B/g,C/f}	*relation*({C,A,B}),	{A/g,B/g,C/g}	⟨⟨*offer*({C})⟩⟩
	{A/g,B/g,C/g}	`𝒫(C).`	{A/g,B/g,C/g}	

and

		`𝒫(A):-`	{A/f,B/g}	
⟨⟨*required*({B})⟩⟩	{A/f,B/g}	`B = A.`	{A/g,B/g}	⟨⟨*offer*({B})⟩⟩
		`𝒫(A):-`	{A/f,B/f}	
	{A/f,B/f}	`𝒫(B),`	{A/g,B/g}	⟨⟨*offer*({B})⟩⟩
	{A/f,B/f}	`A = B.`	{A/g,B/g}	⟨⟨*offer*({A})⟩⟩

Then the following combination can be obtained

		`𝒫(A,B):-`	{A/g,B/f,C/g}	⟨⟨*offer*({A})⟩⟩
	{A/g,B/f,C/g}	*relation*(A),	{A/g,B/f,C/g}	
⟨⟨*required*({C})⟩⟩	{A/g,B/f,C/g}	`B = C.`	{A/g,B/g,C/g}	⟨⟨*offer*({B})⟩⟩
		`𝒫(A,D):-`	{A/g,B/f,C/f,D/f,E/f}	⟨⟨*offer*({A})⟩⟩
⟨⟨*required*({B})⟩⟩	{A/g,B/g,C/f,D/f,E/f}	*relation*({C,A,B}),	{A/g,B/g,C/g,D/f,E/f}	⟨⟨*offer*({C})⟩⟩
	{A/g,B/g,C/g,D/f,E/f}	`𝒫(C,E),`	{A/g,B/g,C/g,D/f,E/g}	
	{A/g,B/g,C/g,D/f,E/g}	`D = E.`	{A/g,B/g,C/g,D/g,E/g}	

If variables `C` and `A` are linked in the first clause, we have that

		`𝒫(A,B):-`	{A/g,B/f}	⟨⟨*offer*({A})⟩⟩
	{A/g,B/f}	*relation*(A),	{A/g,B/f}	
	{A/g,B/f}	`B = A.`	{A/g,B/g}	⟨⟨*offer*({B})⟩⟩
		`𝒫(A,D):-`	{A/g,B/f,C/f,D/f,E/f}	⟨⟨*offer*({A})⟩⟩
⟨⟨*required*({B})⟩⟩	{A/g,B/g,C/f,D/f,E/f}	*relation*({C,A,B}),	{A/g,B/g,C/g,D/f,E/f}	⟨⟨*offer*({C})⟩⟩
	{A/g,B/g,C/g,D/f,E/f}	`𝒫(C,E),`	{A/g,B/g,C/g,D/f,E/g}	
	{A/g,B/g,C/g,D/f,E/g}	`D = E.`	{A/g,B/g,C/g,D/g,E/g}	

This construct abstracts both accumulator pair and difference list techniques. Another argument slice must be supplied in order to provide the required variable `B` in the second clause: the partially defined construction above can be used to prepare different programming techniques, once the required values have all been supplied.

6.4 Conclusions and Discussion

In this chapter we have explained the facilities offered by the knowledge management tool to support the design of programming techniques. A constrained medium (comprising of an abstract clause template and commands to customise it) is employed, by means of which expert programmers can express their programming knowledge (in the form of programming techniques) and make it available to others. A methodology to use the design facilities is also introduced: the expert should first devise a sequence of clause- and mode-annotated argument slices, and link their required and offered variable symbols.

The proposed approach uses simple single-argument program fragments supplemented with information concerning the modes and sharing of variables, the clause- and mode-annotated argument slices, initially in a very abstract form, and offers a small repertoire of commands to gradually customise them. The same formalism is employed both during the extraction and design of argument slices and programming techniques.

The formalism used to represent argument slices encourages a methodical design in which more specific instances are gradually prepared. The single-argument slices are, in their turn, used to devise more sophisticated constructions, the programming techniques. Once an argument slice is prepared, it can be employed to define many distinct argument slices. The notation used to formalise argument slices enables the expert programmer to concentrate on a single argument position and the different ways this argument position can be employed in a programming technique. The complementary information provided by the mode-annotations allows for more support in the preparation of correct argument slices. The manner of formalising techniques encourages expert programmers to focus on the relationships between different argument positions.

Extracted argument slices in more abstract format, *viz.* with predicate descriptors, can be used as initial templates: the system supports the interactive definition of their predicate descriptors by the user. Abstract argument slices previously designed can also be reused and specialised differently. This feature saves the user's effort in having to define an argument slice from the initial sequence of abstract clause templates and also allows the reengineering of parts of a programming technique.

The formalisation employed naturally defines a hierarchy of argument slices in which the topmost node is the initial sequence of abstract clause templates offered to expert programmers; their offspring are more specialised forms of them and the leaf nodes are clause- and mode-annotated argument slices with *relation* subgoals only.

In Appendix C we show additional examples of programming techniques being devised via the preparation of its argument slices. We also provide more details as to how the design tool actually works, with screen dumps and a step-by-step account of the design process.

6.5 Summary

In this chapter we have described a medium incorporated by our tool to support the design of programming techniques. This medium allows:

- the definition of programming techniques by first devising its argument slices and linking their required and offered variable symbols;
- the definition of argument slices via the gradual specification of an initial sequence of abstract clause templates;
- the reuse of existing argument slices to define new constructs;
- the use of abstract argument slices (*i.e.* argument slices with predicate descriptors) to define classes of programming techniques.

Chapter 7

Conclusions and Further Work

7.1 Introduction

The motivation of the present work is the prospect of offering automated or semi-automated support to the management of programming knowledge in knowledge-based software tools. We have employed the notion of programming technique to represent the practices found in Prolog programs and we have implemented a knowledge management tool embedding this view. Such a tool supervises a library of programming techniques, enabling its users to upgrade and manipulate the programming knowledge contained therein.

7.2 Summary

The previous chapters can be summarised as follows:

Chapter I: The context of our work is presented: that is, to address the problem of knowledge acquisition and organisation in knowledge-based development tools for logic programs. We also informally described the adopted notion of Prolog programming technique and explained the knowledge-management tool.

Chapter II: Existing proposals to represent Prolog programming knowledge are surveyed and our adopted view of programming techniques is described. We compared the adequacy of our adopted view against the surveyed work to the task at hand.

Chapter III: We developed a means to represent the execution of Prolog programs to enable us to analyse and extract their programming techniques. The execution is to be represented in a simple form in which we show the manner each variable of the clauses of a procedure is used. Different alternatives for this endeavour were considered.

Chapter IV: We described an automated method to extract the programming techniques of Prolog procedures. The termination and correctness of the method are discussed and examples are given.

Chapter V: We proposed a means to store and manage the programming techniques extracted via the method described in the previous chapter. We introduced a set of relations through which the argument slices comprising the programming techniques can have their design decisions concealed.

Chapter VI: We described a facility offered by our tool to support the design of programming techniques. Programming techniques are designed by having its sequence of argument slices prepared separately then combined together. The preparation of argument slices can be from an initial very general template or by reusing existing components in the knowledge base.

7.3 Contributions

We claim that our work offers the following main contributions:

7.3.1 A Formalisation of Prolog Programming Techniques

We have provided a formal characterisation of Prolog programming techniques. They are formalised as sequences of clause- and mode-annotated argument slices sharing variables in their clause-annotations. The mode-annotations provide a record of the usage mode of the variables in each clause. The clause-annotations make explicit the bindings of variables between distinct argument slices.

Previous work has concentrated on the syntactic aspects of programming techniques. They represent programming techniques informally via syntactic schemata or by producing examples of programs showing instances of a technique. We have extended these approaches by introducing some of the dynamic features of a Prolog program, that is, how its execution changes the instantiation status of its variables.

Dynamic aspects of Prolog programs are used in [Bow92] to help characterise programming techniques. In that work programming techniques are formalised using a language based on the effects of unifications which occur during the program execution. Our formalisation of techniques encompasses this approach, also capturing other dynamic features such as tests, arithmetic calculations and input commands.

7.3.2 A Method for Extracting Prolog Programming Techniques

We have devised a semi-automatic method for extracting programming techniques from working Prolog procedures. In the surveyed work this had been an issue previously untackled. The proposed method works by partitioning a given mode-annotated procedure into its argument slices. The argument slices contain the contributions of each argument position in the procedure.

7.3.3 A Framework for Abstracting and Reimplementing Design Decisions

We have conceived a framework for abstracting the design decisions of Prolog programming techniques. Our framework makes use of the mode-annotations before and after a subgoal in order to assign a predicate descriptor to it; a predicate descriptor is a label given to subgoals representing basic Prolog programming practices. The operators devised to perform the abstraction of subgoals are such that they can also be used in the reverse manner, that is, as operators to specialise predicate descriptors into actual Prolog subgoals. Our framework supports the fully automatic abstraction of argument slices and user-assisted reimplementation.

7.3.4 An Organisation Scheme for Prolog Programming Techniques

We have provided an organisation scheme in which those programming techniques extracted via our proposed method can be stored. This scheme has been conceived with issues of economy of storage and ease of use by humans who will be responsible for managing the knowledge base of programming techniques. Our proposed scheme employs the abstraction framework to find similarities between seemingly disparate argument slices and programming techniques and makes use of this to save the replication of components.

7.3.5 A Methodology for Designing Prolog Programming Techniques

We have proposed a methodology to help expert programmers with the designing of programming techniques. This methodology encourages programmers to think of the argument slices comprising a programming technique, guiding them through their definition and integration. Our methodology supports programmers when thinking about one argument position at a time: it provides its users with a generic template for a clause with a single argument which can be specialised to more specific instances via a repertoire of simple commands and the operators to specialise predicate descriptors. The ordering of the commands to customise the initial template promotes a programming discipline in which argument positions have their contributions to the programming technique gradually specified. Programming slices can then have their variables in clause-annotations appropriately bound together.

Our proposed methodology also supports the reuse of existing argument slices to define new programming techniques, and the definition of programming techniques via the reimplementation of existing abstract argument slices and their combination.

7.3.6 Reverse Software Engineering

One of the objectives of reverse software engineering is to help humans understand pieces of software, possibly with a view to alter or to reuse them. The proposed decomposition of a procedure into its argument contributions allows users to focus on a single argument position and may help them to comprehend a possibly complex piece of software, perhaps written by someone else, or even by the forgetful user himself, and devoid of explanatory comments. The inspection of the sequence of argument slices of a procedure may, furthermore, provide the user with an account of the relationships, or lack of relationships, between the argument positions.

The abstraction of the argument slices provides further support in helping programmers understanding alien code. Higher level representations of argument slices employing predicate descriptors hide implementational details and/or design decisions, providing the user with a cleaner and more informative account of each argument slice. The abstraction of an argument slice can be seen as a representation of the programming practices found, as a declarative description of the relationships discovered or as a mixture of these two.

Meaningful higher level constructs are identified in the abstractions of an argument slice. This can be seen as a *design recovery* of those programming practices employed: particular design decisions are given mnemonic names which reveal, in a more abstract and homogeneous manner, the overall design of the argument slice. For instance, a convenient representation for a class of programming techniques decomposing a singly-recursive data structure is given by the following clause- and mode-annotated argument slice:

	$\mathcal{P}$(A):-	{A/g}	⟨⟨*offer*({A})⟩⟩
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	⟨⟨*offer*({A})⟩⟩
{A/g,B/f,C/f}	*decomp*(A,{B,C}),	{A/g,B/g,C/g}	⟨⟨*offer*({B,C})⟩⟩
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	

The abstract design of an argument slice can be depicted in procedural or declarative terms, or as a mixture of these two: as explained in Section 5.10, our system supports the user-assisted abstraction of an argument slice, in which the user controls the process. In our example above, if *relation* descriptors had been used instead, we would have a declarative account of the design.

The mnemonic predicate descriptors can provide us with a crude form of on-line documentation, in which the procedural meaning of the computation carried out is stated. This facility can be enhanced by having a pre-canned explanatory text in some natural language appropriately inserted describing, in a more verbose fashion, the computations carried out at that point. When the mode-annotations are obtained through concrete interpretation, an accurate and fully automatic form of documentation can be achieved with this proposed enhancement.

7.4 Discussion

The following issues are worthy of further discussion:

7.4.1 Addressed Class of Programming Techniques

Not every Prolog program can have its programming techniques extracted by our proposed method of Chapter 4. Those programs which do not comply with the syntactic constraints laid out in Section 1.5 are obviously not handled by our method: we could rely on an automatic way to translate Prolog programs into their normal form, thus making our work more robust, but useful programming constructs, such as if-then-else's and cuts, are not catered for in our system.

As pointed out in Section 2.9, those procedures without arguments are outside the scope of our approach. Our definition of programming technique is given in terms of argument slices and subgoal relevance to argument positions: if a programming technique uses no argument positions, then our definition fails to capture it.

Our analysis of relevance of a subgoal relies on its variables and their sets of dependencies. Subgoals which do not have arguments (say, if they employ global data structures managed by `assert`'s and `retract`'s) cannot be checked for their relevance to argument slices. Cuts are an important form of such subgoals without variables: it is not possible, with our proposed method, to decide whether a cut is relevant to an argument slice.

Programming techniques spreading over more than one procedure are also outside the scope of our work. Mutual recursion and initialisation calls are examples of important programming practices that cannot be properly handled by our approach. Our proposed model for the dependency of variables within a procedure could, in principle, be expanded to represent inter-procedural relationships of variables, naturally scaling up to cover the variable dependency of a complete program with its many procedures. In Section 7.5.4 below we discuss this in more detail.

Our proposed formalisation for Prolog programming techniques, however, successfully accommodates the majority of the characterisations of Prolog programming practices found in the literature. These characterisations all view programming techniques as argument positions and their relevant subgoals in each clause, very much in accordance with our suggested argument slices.

The *skeletons* and *additions* of Kirschenbaum and colleagues [SS86, KLS89, Lak89, SK93] (Section 2.3) are very much in accordance with our view of individual contributions of argument slices defining a procedure. The examples found in those references can either have their programming techniques extracted (if they are programs showing instances of a technique) or be designed (if it is an informal description of a programming technique).

Gegg-Harrison's Prolog Schemata [GH89, GH91, GH93] (Section 2.4) are also in accordance with our approach: the programs used to generate automatically the schemata could have their programming techniques extracted and incorporated by our system. Our characterisation of programming techniques is more general than that provided by Gegg-Harrison's schemata: not only list processing techniques in initial argument positions are captured (Gegg-Harrison's schemata only focus on such constructs) but also any other data structures (or numeric values, strings, etc.) in any argument positions.

Barker-Plummer's programming clichés [BP90] (Section 2.5) are abstract forms of programming techniques, in our approach. In these abstract techniques, predicate names have been abstracted to second-order symbols: these clichés, similar to Gegg-Harrison's schemata, only address the first argument positions of a program.

The programming techniques detected and represented in the work of Bowles [Bow92] (Section 2.7) are also captured by our approach. Our way to model variable dependency within clauses can be used to infer the *inclusion* and *sharing* relationships between variables, defined in that work.

7.4.2 Psychological Evidence for Argument Slicing

In this work we do not claim any psychological evidence for the argument slicing of procedures. We have not carried out any experiments to verify this issue. However, the pragmatics of Prolog programming and debugging may suggest that the decomposition of a procedure in terms of argument positions and their relationships is a standard practice. Given a problem for which a Prolog program has to be found, one of the first issues that arises is the number of arguments the solution will have in its head goal. Each of these arguments will then have to be provided with their contributions and relationships in each clause [BRB, Dev90, SS86]. The directionality (input or output) for each argument slice is also taken into account when preparing the solution [SS86, Dev90].

When understanding or debugging alien programs, programmers employ information about the directionality and type of each argument position and their contributions and relationships within each clause. If an undocumented program is to be executed and queries have to be provided, the choice of the value of each argument position in the query will inevitably lead to a careful inspection of those subgoals within each clause which are relevant to the argument position.

Psychological evidence was found for the slicing of imperative programs [Wei82]; it was noticed that during the debugging of imperative programs, when programmers work *backwards*, that is, starting from the point in the program where an error first becomes manifest, and then proceed to reason about the sequence of commands that could have led to that error. Although the notion of program slices in [Wei84] is very much different from our notion of argument slices (see Section 4.6) we believe that this result may transfer to Prolog programs.

Any device that helps programmers understand or develop complex pieces of software by focusing on smaller portions of the initial problem has its merits. Even if it had been proved that logic programmers do not employ any notion of slices, the argument slicing approach has enough advantages to be considered as a logic programming methodology on its own and possibly be taught together with other alternatives such as [Dev90].

7.4.3 Declarative and Procedural Meanings of Prolog Constructs

The effects of unification in Prolog programs are explained in some textbooks (*e.g.* [SS86] and [O'K90]) and papers (*e.g.* [KLS89], [Lak89] and [Rob91b]) in terms of common programming practices such as building or decomposing data structures, binding variables, tests, and so on. Prolog programmers can also make use of this more procedural terminology when following the trace of a program, or giving a procedural account of a program.

We have devised a small but expressive repertoire of such procedural terminology. We have also provided an automatic manner to assign these terms to the constructs of Prolog programs. Our proposal, however, naturally accommodates a more implementation-independent and less committed declarative account of programming constructs via the *relation* predicate descriptor. The abstraction and reimplementation have been employed solely on argument slices, but entire mode-annotated procedures may benefit from this approach.

7.4.4 Scaling-up

Our approach addresses Prolog programs from a procedure level. A complete Prolog program can be analysed and have its programming techniques extracted, one procedure at a time. Our approach and its implementation pose no restrictions concerning the number of arguments of the procedure to be analysed.

7.5 Applications and Further Work

We found that some of the concepts developed in our work have other applications beyond those originally intended. In this section we list some of them and also give directions for further work.

7.5.1 Reverse Engineering of Procedures and Programs

The reverse engineering of full procedures and programs can also benefit from our ideas. The study and implementation of an environment allowing its users to make experiments with the reengineering of complete Prolog procedures, which incorporated the concepts of argument slicing and abstraction/reimplementation, seems a prospective direction of research. The abstraction and reimplementation of clause- and mode-annotated argument slices adapt naturally to complete mode-annotated procedures. Our implemented system is, in fact, able to cope with such complete procedures.

The reimplementation of an abstract argument slice gives rise to a new version of the procedure it was extracted from. Additionally, by allowing the user to concentrate on single argument positions, we can envisage a scenario in which an argument slice is replaced by a compatible construct. To illustrate this application, let there be the following *sum*/2 mode-annotated procedure

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

If we suppose that its first mode-annotated argument slice is abstracted then reimplemented as

```
                   𝒫(A):-             {A/g}
          {A/g}      solution(A).     {A/g}
                   𝒫(A):-             {A/g,C/f,D/f}
  {A/g,C/f,D/f}      move(A,D,C),     {A/g,C/g,D/g}
  {A/g,C/g,D/g}      𝒫(D).            {A/g,C/g,D/g}
```

where `move(A,D,C)` is a fact depicting a move from configuration `A` to `D` with an associated cost `C`, and its second argument slice is replaced by

```
                   𝒫(B):-          {B/f}
          {B/f}      B = [].       {B/g}
                   𝒫(B):-          {B/f,C/f,E/f}
  {B/f,C/g,E/f}      𝒫(E),         {B/f,C/g,E/g}
  {B/f,C/g,E/g}      B = [C|E].    {B/g,C/g,E/g}
```

then the following mode-annotated procedure would be obtained

```
                          costs(A,B):-     {A/g,B/f}
              {A/g,B/f}     solution(A),   {A/g,B/f}
              {A/g,B/f}     B = [].        {A/g,B/g}
                          costs(A,B):-     {A/g,B/f,C/f,D/f,E/f}
  {A/g,B/f,C/f,D/f,E/f}     move(A,D,C),   {A/g,B/f,C/g,D/g,E/f}
  {A/g,B/f,C/g,D/g,E/f}     costs(D,E),    {A/g,B/f,C/g,D/g,E/g}
  {A/g,B/f,C/g,D/g,E/g}     B = [C|E].     {A/g,B/g,C/g,D/g,E/g}
```

That is, the initial procedure *sum*/2 which sums the elements of a list was reengineered to obtain procedure *costs*/2 which finds the moves between a given configuration and a final solution and builds a list with the associated costs of the moves.

A useful tool could be devised supporting the reverse engineering of complete procedures (not only its argument slices). This tool could, additionally, supply its users with alternative representations of a complete mode-annotated procedure in terms of predicate descriptors and provide support in reimplementing its abstract versions. The mode-annotated *sum*/2 procedure above, for instance, can be displayed with its procedural meaning, that is,

```
                                𝒫(A,B):-              {A/g,B/f}
                    {A/g,B/f}      test({A}),         {A/g,B/f}
                    {A/g,B/f}      assign(∅,{B}).     {A/g,B/g}
                                𝒫(A,B):-              {A/g,B/f,C/f,D/f,E/f}
      {A/g,B/f,C/f,D/f,E/f}        decomp(A,{C,D}),   {A/g,B/f,C/g,D/g,E/f}
      {A/g,B/f,C/g,D/g,E/f}        𝒫(D,E),            {A/g,B/f,C/g,D/g,E/g}
      {A/g,B/f,C/g,D/g,E/g}        calc({E,C},B).     {A/g,B/g,C/g,D/g,E/g}
```

The tool would then support the reimplementation of each of the predicate descriptors. Given a mode-annotated procedure, our implemented system can be adapted to perform its abstraction (fully automatically or user-assisted) and support be given to its reimplementation.

7.5.2 Debugging of Prolog Programs

A potential application for some of our ideas lies within the field of program debugging. For instance, by keeping track of those programming techniques employed in a given procedure, it is possible to detect inconsistencies between the intended use, represented by the instantiation status of its variables recorded in the mode-annotations of the programming technique, and its actual use in the procedure execution. This application could also address the faulty usage of individual argument slices by the same means.

7.5.3 Tutoring Environments

The framework to abstract design decisions provides us with a number of ways to represent Prolog procedures in different levels of abstraction. The less committed representations of a procedure using predicate descriptors could be employed in a tutoring environment when matching the code devised by a student with its ideal solution. For instance, the common mistake of using the `=` operator for the *is*/2 operator can lead to the following wrong implementation of *sum*/2:

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

The sum itself will not be performed since a binding will actually take place as a result of executing the subgoal `B = E + C`. An abstract version of this mode-annotated procedure, showing the computational effects of each subgoal is given by

	$\mathcal{P}$(A,B):-	{A/g,B/f}
{A/g,B/f}	*test*({A}),	{A/g,B/f}
{A/g,B/f}	*assign*(∅,{B}).	{A/g,B/g}
	$\mathcal{P}$(A,B):-	{A/g,B/f,C/f,D/f,E/f}
{A/g,B/f,C/f,D/f,E/f}	*decomp*(A,{C,D}),	{A/g,B/f,C/g,D/g,E/f}
{A/g,B/f,C/g,D/g,E/f}	$\mathcal{P}$(D,E),	{A/g,B/f,C/g,D/g,E/g}
{A/g,B/f,C/g,D/g,E/g}	*build*({E,C},B).	{A/g,B/g,C/g,D/g,E/g}

The difference between this version and the ideal version becomes apparent: the mismatch between the predicate descriptor *build* and the descriptor *calc* supposed to be used provides useful information in spotting a student's misconception.

The use of argument slices may provide a more robust fashion to analyse programs in a tutoring environment since it would provide more flexibility in the ordering of argument positions. Procedures with disparate argument position orderings could still be compared from the perspective of argument slices.

7.5.4 Techniques Spreading Over More Than One Procedure

Those programming techniques spreading over more than one procedure, *e.g.* initialisation calls and mutually recursive procedures, were not addressed. We would want to extend our formalism to cover these useful programming techniques.

Such programming techniques would consist of larger "chunks" of code comprising the related procedures. For instance, in the following *count*/2 definition:

```
count(L,C):-                    count(L,C):-
   Ac = 0,                         count(L,0,C).
   count(L,Ac,C).
count(L,Ac,C):-                 count([],C,C).
   L = [],
   C = Ac.
count(L,Ac,C):-                 count([_|Xs],Ac,C):-
   L = [_|Xs],                     NAc is Ac + 1,
   NAc is Ac + 1,                  count(Xs,NAc,C).
   count(Xs,NAc,C).
```

A formalisation of the technique employed in the accumulator pair would have to take into account the initial value being assigned to variable `Ac`. The programming technique in the second argument position, for example, could be depicted as

```
Q:-
   Ac = 0,           ⟨⟨offer(Ac)⟩⟩
   P(Ac).

P(Ac).               ⟨⟨offer(Ac)⟩⟩
P(Ac):-              ⟨⟨offer(Ac)⟩⟩
   NAc is Ac + 1,    ⟨⟨offer(NAc)⟩⟩
   P(NAc).
```

The relationship between *count*/2 and *count*/3 with respect to the second argument position is captured by the representation above: the initialisation call $\mathcal{Q}$ employs `Ac` only internally and thus appears without any argument; $\mathcal{P}$ depicts a technique adding one to a variable instantiated to a numeric value and using the new obtained value in its recursive subgoal. In order to properly analyse such constructs, the formalisation of relationships between the variables of a clause (underlying our method for argument slicing) would have to account for possible links created by means of the execution of an auxiliary procedure.

Initialisation calls and mutually recursive procedures are examples of techniques extending over more than one procedure. There are, however, more complex constructs, involving a number of procedures which, on their turn, may employ other procedures and so on. A robust characterisation of these complex programming techniques could eventually be used for analysing full programs.

7.5.5 Adapting our Approach to Other Programming Languages

We have geared our work towards Prolog, incorporating its syntax and execution model. We would want to investigate the adaptability of our approach to program decomposition to other logic programming languages such as Goedel, and to languages of the functional and procedural paradigms.

Our approach should adapt naturally to other logic programming languages. The special built-in predicates offered by the new language would have to be incorporated to our definitions, their possible meanings (*i.e.* whether the predicate always succeeds, whether it is a test, or if it is used to obtain values) being used to analyse their importance within programs. The representation of the execution of a procedure proposed here is robust enough to handle different execution strategies and syntactic restrictions in the programs. The notion of relevance of subgoals, argument slicing and programming techniques should also adapt well.

Programs within the functional and procedural paradigms do not have the potential multiple uses of Prolog programs. In Prolog programs, in order to find out which computation(s) a subgoal is performing, a careful analysis should take place considering the different usages of the procedure within which that subgoal appears. This feature led us into adopting an explicit representation for the usage of a procedure, by means of the instantiation of its variables.

Procedural languages have richer syntax, with larger repertoires of constructs which may span over long portions of a program. A program, for instance, may consist of a single loop with many other commands within it. A representation for such programming practices should allow the initial abstraction of the details of the command (in the case of a loop command it should allow the content of the loop to be considered initially as a single abstract object) and, upon request, its subsequent careful examination. The notion of relevance of a command and the slicing process for procedural languages have been investigated [Wei82, Wei84, RW89, GL91, JR94, RHSR94], but work aimed at formalising procedural programming practices has concentrated on plans and algorithms [Wat92, ROLJ90] used in programs. It would be profitable to provide a purely syntactic account of the programming practices of a procedural language, and observe their use within a techniques editor: it has been pointed out [BB93] that novice programmers may have difficulties in grasping the concept of programming plans, whereas programming techniques are simpler and less numerous. The simple instantiation mechanism of Prolog has two distinct counterparts in some procedural languages, viz. parameter passing by value and reference: it would be worth investigating the contexts of such practices and how they can be represented and distinguished.

Functional languages, on the other hand, have a simple syntax and their constructs do not have the potential multiple uses of logic programs. The argument slicing of a function could then safely rely only on the syntax of its components. The notion of relevance of a command and the characterisation of programming practices by means of descriptors would have to be substantially changed, though. A related work in this area is that of Balmas [Bal95], employing *conceptual schemata* to characterise common computations of LISP functions: these schemata are prototypical versions of computations associated with specific argument positions of the function. Balmas' work is connected to ours in that both use syntactic features of the program in the analysis, whereas [Wat92], [Wil92], [RW90] use the more complex concepts of algorithms and programming plans.

Bibliography

[AK90] H. Ait-Kaci. The WAM: A (Real) Tutorial. Technical Report 5, Paris Research Laboratory, Digital Equipment Centre Technique Europe, 1990.

[Bal95] F. Balmas. A Conceptualisation Model for Programs. Rapport Technique 01-1995-2, Département Informatique, Université Paris 8, 1995.

[BB93] A. W. Bowles and P. Brna. Programming Plans and Programming Techniques. In *World Conference on Artificial Intelligence in Education*, pages 378–385, Edinburgh, Scotland, 1993.

[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.

[BGB91] A. Bundy, G. Grosse, and P. Brna. A Prolog Techniques Recursive Editor. *Instructional Science*, 20(2):135–172, 1991.

[BJCD87] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract Interpretation: Towards the Global Optimisation of Prolog Programs. In *IEEE Symposium on Logic Programming*, pages 192–204, San Francisco, California, August 1987.

[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.

[BP90] D. Barker-Plummer. Cliche Programming in Prolog. In *Second Workshop on Meta-Programming in Logic*, pages 247–256, 1990.

[BRB] A. W. Bowles, D. Robertson, and P. Brna. A Case-Based Reasoning Approach to Supporting Novice Programmers. Submitted to Applied Artificial Intelligence.

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

[BRV+94] A. W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. *International Journal of Human-Computer Studies*, 41(3):329–350, September 1994. Also as Research Paper 641, Dept of Artificial Intelligence, University of Edinburgh.

[Bun88] A. Bundy. Proposal for a Recursive Techniques Editor for Prolog. Research Paper 394, Department of Artificial Intelligence, University of Edinburgh, 1988.

[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.

[CCI90] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: a Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog (3rd. Edition)*. Springer-Verlag, 1987.

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

[Die87] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *IEEE Symposium on Logic Programming*, pages 264–272, San Francisco, California, August 1987.

[DW86] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. In *IEEE Symposium on Logic Programming*, pages 78–88, Salt Lake City, Utah, U.S.A., September 1986.

[FF92] N. E. Fuchs and M. P. J Fromherz. Schema-Based Transformations of Logic Programs. In *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LoPSTr'92)*. Springer-Verlag, 1992.

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

[GH89] T. S. Gegg-Harrison. Basic Prolog Schemata. Technical Report CS-1989-20, Department of Computer Science, Duke University; Durham, North Carolina, U.S.A., September 1989.

[GH91] T. S. Gegg-Harrison. Learning Prolog in a Schema-Based Environment. *Instructional Science*, 20:173–192, 1991.

[GH93] T. S. Gegg-Harrison. Exploiting Program Schemata in a Prolog Tutoring System. Technical Report CS-1993-11, Department of Computer Science, Duke University; Durham, North Carolina, U.S.A., April 1993. Reformatted version of PhD dissertation with the same title and equivalent content.

[GL91] K. B. Gallagher and J. R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[Gol86] A. T. Goldberg. Knowledge-Based Programming: A Survey of Program Design and Construction Techniques. *IEEE Transactions on Software Engineering*, SE-12(7):752–768, July 1986.

[JR94] D. Jackson and E. J. Rollins. A New Model of Program Dependences for Reverse Engineering. In *Proceedings of SIGSOFT'94*, New Orleans, Louisiana, U.S.A., December 1994.

[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.

[KMS94] M. Kirschenbaum, S. Michaylov, and L. Sterling. Skeletons and Techniques as a Normative Approach to Program Development in Logic-Based Languages. OSU-CISRC-5/94-TR25, Department of Computer and Information Science, Ohio State University, Ohio, U.S.A., 1994.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., U.S.A, Second edition, 1988. Prentice-Hall Software Series.

[Lak89] A. Lakhotia. Incorporating 'Programming Techniques' into Prolog Programs. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, Ohio, U.S.A., October 1989.

[Let88] S. I. Letovsky. *Plan Analysis of Programs*. PhD thesis, Department of Computer Science, Yale University, 1988. Published as Technical Report YALE/DCS/RR #662.

[Llo93] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Second, Extended edition, 1993.

[Llo94] J. W. Lloyd. Practical Advantages of Declarative Programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 1994. Invited Paper.

[Loo88a] C.-K. Looi. Analysing Novices' Programs in a Prolog Intelligent Teaching System. In *Proceedings of the European Conference on Artificial Intelligence*, Munich, Germany, August 1988.

[Loo88b] C.-K. Looi. *Automatic Program Analysis in a Prolog Intelligent Teaching System*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1988.

[LS90] A. Lakhotia and L. Sterling. How to Control Unfolding when Specialising Interpreters. In L. Sterling, editor, *The Practice of Prolog*. MIT Press, 1990.

[Mel87] C. Mellish. Abstract Interpretation of Prolog Programs. In *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[Mic94] S. Michaylov. Skeletons and Techniques for the Systematic Development of Constraint Logic Programs. OSU-CISRC-6/94-TR30, Department of Computer and Information Science, Ohio State University, Ohio, U.S.A., 1994.

[MU87] H. Mannila and E. Ukkonen. Flow Analysis of Prolog Programs. In *IEEE Symposium on Logic Programming*, pages 205–214, San Francisco, California, August 1987.

[NN90] H. R. Nielson and F. Nielson. Eureka Definitions for Free – Disagreement Points for Fold/Unfold Transformations. In *Proceedings of ESOP'90, Lecture Notes in Computer Science 432*, pages 291–305. Springer-Verlag, August 1990.

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

[PGLS88] N. Prywes, X. Ge, I. Lee, and M Song. Reverse Software Engineering. Technical Report MS-CIS-88-99, Department of Computer and Information Science, University of Pennsylvania, 1988.

[RF92] C. Rich and Y. A. Feldman. Seven Layers of Knowledge Representation and Reasoning in Support of Software Development. *IEEE Transactions on Software Engineering*, 18(6):451–469, June 1992.

[RHSR94] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up Slicing. In *Proceedings of SIGSOFT'94*, New Orleans, Louisiana, U.S.A., December 1994.

[Rob91a] D. Robertson. A Preliminary Description of a Techniques Editor for Prolog. KBS Group Notes 5, Department of Artificial Intelligence, University of Edinburgh, 1991.

[Rob91b] D. Robertson. A Simple Prolog Techniques Editor for Novice Users. In *3rd Annual Conference on Logic Programming*, Edinburgh, Scotland, April 1991. Springer-Verlag.

[Rob91c] D. Robertson. A Simple Prolog Techniques Editor for Novice Users. Research Paper 523, Department of Artificial Intelligence, University of Edinburgh, 1991.

[ROLJ90] S. Rugaber, S. B. Ornburn, and R. J. LeBlanc Jr. Recognising Design Decisions in Programs. *IEEE Software*, 7(1):46–54, 1990.

[Ros89] P. M. Ross. *Advanced Prolog: Techniques and Examples.* Addison-Wesley Publishing Company, 1989.

[RW89] T. Reps and Y. Wuu. The Semantics of Program Slicing and Program Integration. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'89)*, pages 360–374, Barcelona, Spain, March 1989. Lecture Notes in Computer Science, vol. 352.

[RW90] C. Rich and L. M. Wills. Recognising a Program's Design: A Graph-Parsing Approach. *IEEE Software*, 7(1):82–88, 1990.

[SK93] L. Sterling and M. Kirschenbaum. Applying Techniques to Skeletons. In *Constructing Logic Programs.* John Wiley & Sons Ltd, London, England, 1993.

[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.

[Van94] P. Vanneste. *A Reverse Engineering Approach to Novice Program Analysis.* PhD thesis, Katholieke Universiteit Leuven, Belgium, May 1994.

[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. Extracting Prolog Programming Techniques. In *Proceedings of the 11th Brazilian Symposium on Artificial Intelligence*, Fortaleza, Ceará, Brazil, October 1994. Also as Research Paper 715, Dept. of Artificial Intelligence, University of Edinburgh.

[VF95] W. W. Vasconcelos and N. E. Fuchs. Enhanced Schema-Based Transformations for Logic Programs and their Opportunistic Usage in Program Analysis and Optimisation. Technical Report 95.16, Institut für Informatik, May 1995.

[VV95] M. Vargas-Vera. *Using Prolog Techniques to Guide Program Composition.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1995.

[VVRI93a] M. Vargas-Vera, D. Robertson, and R. Inder. An Environment for Combining Prolog Programs. Research Paper 610, Department of Artificial Intelligence, University of Edinburgh, 1993.

[VVRI93b] M. Vargas-Vera, D. Robertson, and R. Inder. A Mathematical Framework for the Problem of Combination of Prolog Programs. Research Paper 612, Department of Artificial Intelligence, University of Edinburgh, 1993.

[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.

[Wat88] R. C. Waters. Program Translation via Abstraction and Reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, August 1988.

[Wat92] R. C. Waters. Cliché-Based Program Editors. Technical Report 91-01a, Mitsubishi Electric Research Laboratories, May 1992.

[Wei82] M. Weiser. Programmers Use Slices When Debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wil92] L. M. Wills. Automated Program Recognition by Graph Parsing. Technical Report 1358, Artificial Intelligence Laboratory, M.I.T., July 1992.

Appendix A

Examples of Extracted Programming Techniques

In this appendix we show some selected examples of programming techniques being extracted. We have chosen 3 classic Prolog programs found in most textbooks [SS86, Ros89, O'K90].

A.1 Programming Techniques of *append*/3 Predicate

Let us suppose the user decides to extract the techniques of the following version of the *append*/3 procedure:

```
append(A,B,C):-
   A = [],
   B = C.
append(A,B,C):-
   A = [D|E],
   C = [D|F],
   append(E,B,F).
```

with respect to query

```
?- append([1,2,3],[4,5],A)
```

providing the following mode-annotated version of *append*/3:

```
                            append(A,B,C):-     {A/g,B/g,C/f}
            {A/g,B/g,C/f}      A = [],          {A/g,B/g,C/f}
            {A/g,B/g,C/f}      B = C.           {A/g,B/g,C/g}
                            append(A,B,C):-     {A/g,B/g,C/f,D/f,E/f,F/f}
{A/g,B/g,C/f,D/f,E/f,F/f}      A = [D|E],       {A/g,B/g,C/f,D/g,E/g,F/f}
{A/g,B/g,C/f,D/g,E/g,F/f}      C = [D|F],       {A/g,B/g,C/i,D/g,E/g,F/f}
{A/g,B/g,C/i,D/g,E/g,F/f}      append(E,B,F).   {A/g,B/g,C/g,D/g,E/g,F/g}
```

and the following clause- and mode-annotated argument slices:

$\hat{\hat{P}}_1 =$

	append(A):-	{A/g}	⟨⟨ *offer*({A}) ⟩⟩
{A/g}	A = [].	{A/g}	
	append(A):-	{A/g,D/f,E/f}	⟨⟨ *offer*({A}) ⟩⟩
{A/g,D/f,E/f}	A = [D|E],	{A/g,D/g,E/g}	⟨⟨ *offer*({D,E}) ⟩⟩
{A/g,D/g,E/g}	append(E).	{A/g,D/g,E/g}	

$\hat{\hat{P}}_2 =$

	append(B).	{B/g}	⟨⟨ *offer*({B}) ⟩⟩
	append(B):-	{B/g}	⟨⟨ *offer*({B}) ⟩⟩
{B/g}	append(B).	{B/g}	

$\hat{\hat{P}}_3 =$

		append(C):-	{B/g,C/f}	
⟨⟨ *required*({B}) ⟩⟩	{B/g,C/f}	B = C.	{B/g,C/g}	⟨⟨ *offer*({C}) ⟩⟩
		append(C):-	{C/f,D/f,F/f}	
⟨⟨ *required*({D}) ⟩⟩	{C/f,D/g,F/f}	C = [D|F],	{C/i,D/g,F/f}	⟨⟨ *offer*({C}) ⟩⟩
	{C/i,D/g,F/f}	append(F).	{C/g,D/g,F/g}	⟨⟨ *offer*({F}) ⟩⟩

The techniques of procedure *append*/3 are:

1. $\mathsf{T}_1 = \langle \hat{\hat{P}}_1 \rangle$, a list-decomposition technique;
2. $\mathsf{T}_2 = \langle \hat{\hat{P}}_2 \rangle$, a technique passing a parameter down the recursive call;
3. $\mathsf{T}_3 = \langle \hat{\hat{P}}_1, \hat{\hat{P}}_2, \hat{\hat{P}}_3 \rangle$, a technique to build a list with the elements supplied by the first technique, followed by the value passed down by the second technique.

Our management-tool provides a facility to help the visualisation of the extracted techniques. The outcome of the extraction process is shown as in Figure A.1 and it is offered the possibility of inspection of the nodes of the tree.

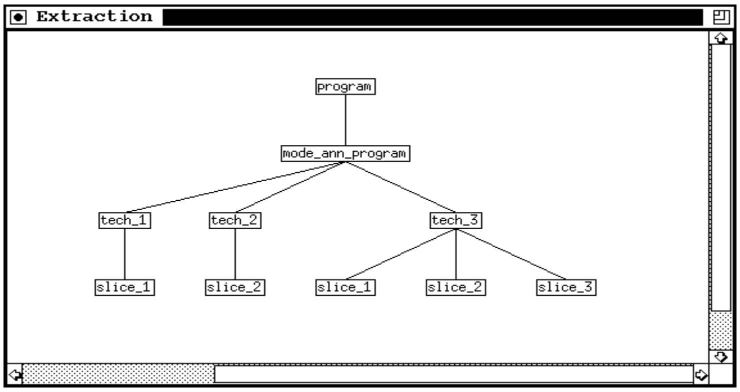


Figure A.1: Simplified Description of the Programming Techniques of *append*/3

A.2 Programming Techniques of *count_sum*/4 Predicate

The following procedure *count_sum*/4

```
count_sum(A,Y,Count,Sum):-
   A = [X|Xs],
   X = Y,
   Count = 0,
   Sum = 0.
count_sum(A,Y,Count,Sum):-
   A = [X|Xs],
   X \== Y,
   count_sum(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, if analysed with respect to query

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

yields the following clause-annotated argument slices (for the sake of brevity the clause annotations with empty sets of variables are omitted):

- $\widehat{P}_1 =$

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

- $\widehat{P}_2 =$

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

- $\widehat{P}_3 =$

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

- $\widehat{P}_4 =$

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

Its 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.

A.3 Programming Techniques of a Fuzzy Expert System

The following fuzzy expert system is an enhanced meta-interpreter [SS86, SL88] which deals with fuzzy inferences and builds a proof of its derivations:

```
fuzzy(S,CF,Xp):-
   fact(S,CF),
   Xp = fact(S,CF).
fuzzy(D,CF,Xp):-
   rule(D,S,CFR),
   fuzzy(S,CFS,XpS),
   CF is CFR * CFS,
   Xp = rule(D,CF,XpS).
fuzzy(Ng,CF,Xp):-
   Ng = not(S),
   fuzzy(S,CFS,XpS),
   CF is 1 - CFS,
   Xp = neg(S,CF,XpS).
fuzzy(Cj,CF,Xp):-
   Cj = S & Ss,
   fuzzy(S,CFS,XpS),
   fuzzy(Ss,CFSs,XpSs),
   min(CFS,CFSs,CF),
   Xp = conj(CF,XpS,XpSs).
fuzzy(Dj,CF,Xp):-
   Dj = S or Ss,
   fuzzy(S,CFS,XpS),
   fuzzy(Ss,CFSs,XpSs),
   max(CFS,CFSs,CF),
   Xp = disj(CF,XpS,XpSs).

% Auxiliary Routines -------------------------------------------------

min(X,Y,Min):- X < Y, Min = X.
min(X,Y,Min):- X >= Y, Min = Y.
max(X,Y,Max):- X > Y, Max = X.
max(X,Y,Max):- X =< Y, Max = Y.

% Knowledge-Base of Diseases and Symptoms ------------------------------

rule(D,S,CF):-
   D = disease(flu),
   S = not(disease(smallpox)) &
         (symptom(temperature,high) or symptom(discharge-from-eyes,present)),
   CF = 0.45.
fact(S,CF):-
   S = symptom(temperature,high),
   CF = 0.5.
       ⋮
```

Predicate *fuzzy*/3 is such that it holds if its second argument is the certainty factor of its first argument, and its third argument has the proof-tree built in the proof of the first argument. If we extract the techniques of procedure *fuzzy*/3 with respect to the query `?- fuzzy(disease(flu),CF,Xp)` we have the following clause-annotated argument slices:

- $\widehat{P}_1 =$

```
  fuzzy(S):-            ⟨⟨ offer({S}) ⟩⟩
     fact(S,CF).        ⟨⟨ offer({CF}) ⟩⟩
  fuzzy(D):-            ⟨⟨ offer({D}) ⟩⟩
     rule(D,S,CFR),     ⟨⟨ offer({S,CFR}) ⟩⟩
     fuzzy(S).
  fuzzy(Ng):-           ⟨⟨ offer({Ng}) ⟩⟩
     Ng = not(S),       ⟨⟨ offer({S}) ⟩⟩
     fuzzy(S).
  fuzzy(Cj):-           ⟨⟨ offer({Cj}) ⟩⟩
     Cj = S & Ss,       ⟨⟨ offer({S,Ss}) ⟩⟩
     fuzzy(S),
     fuzzy(Ss).
  fuzzy(Dj):-           ⟨⟨ offer({Dj}) ⟩⟩
     Dj = S or Ss,      ⟨⟨ offer({S,Ss}) ⟩⟩
     fuzzy(S),
     fuzzy(Ss).
```

- $\widehat{P}_2 =$

```
                        fuzzy(CF):-
⟨⟨ required({S}) ⟩⟩        fact(S,CF).            ⟨⟨ offer({CF}) ⟩⟩
                        fuzzy(CF):-
⟨⟨ required({D}) ⟩⟩        rule(D,S,CFR),         ⟨⟨ offer({S,CFR}) ⟩⟩
                           fuzzy(CFS),            ⟨⟨ offer({CFS}) ⟩⟩
                           CF is CFR * CFS.       ⟨⟨ offer({CF}) ⟩⟩
                        fuzzy(CF):-
                           fuzzy(CFS),            ⟨⟨ offer({CFS}) ⟩⟩
                           CF is 1 - CFS.         ⟨⟨ offer({CF}) ⟩⟩
                        fuzzy(CF):-
                           fuzzy(CFS),            ⟨⟨ offer({CFS}) ⟩⟩
                           fuzzy(CFSs),           ⟨⟨ offer({CFSs}) ⟩⟩
                           min(CFS,CFSs,CF).      ⟨⟨ offer({CF}) ⟩⟩
                        fuzzy(CF):-
                           fuzzy(CFS),            ⟨⟨ offer({CFS}) ⟩⟩
                           fuzzy(CFSs),           ⟨⟨ offer({CFSs}) ⟩⟩
                           max(CFS,CFSs,CF).      ⟨⟨ offer({CF}) ⟩⟩
```

- $\widehat{P}_3 =$

```
                          fuzzy(Xp):-
⟨⟨ required({S,CF}) ⟩⟩       Xp = fact(S,CF).              ⟨⟨ offer({Xp}) ⟩⟩
                          fuzzy(Xp):-
   ⟨⟨ required({D}) ⟩⟩       rule(D,S,CFR),                ⟨⟨ offer({S,CFR}) ⟩⟩
                             fuzzy(XpS),                   ⟨⟨ offer({XpS}) ⟩⟩
  ⟨⟨ required({CF}) ⟩⟩       Xp = rule(D,CF,XpS).          ⟨⟨ offer({Xp}) ⟩⟩
                          fuzzy(Xp):-
                             fuzzy(XpS),                   ⟨⟨ offer({XpS}) ⟩⟩
⟨⟨ required({S,CF}) ⟩⟩       Xp = neg(S,CF,XpS).           ⟨⟨ offer({Xp}) ⟩⟩
                          fuzzy(Xp):-
                             fuzzy(XpS),                   ⟨⟨ offer({XpS}) ⟩⟩
                             fuzzy(XpSs),                  ⟨⟨ offer({XpSs}) ⟩⟩
  ⟨⟨ required({CF}) ⟩⟩       Xp = conj(CF,XpS,XpSs).       ⟨⟨ offer({Xp}) ⟩⟩
                          fuzzy(Xp):-
                             fuzzy(XpS),                   ⟨⟨ offer({XpS}) ⟩⟩
                             fuzzy(XpSs),                  ⟨⟨ offer({XpSs}) ⟩⟩
  ⟨⟨ required({CF}) ⟩⟩       Xp = disj(CF,XpS,XpSs).       ⟨⟨ offer({Xp}) ⟩⟩
```

The set $\mathcal{T} = \{\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3\}$ of techniques of *fuzzy*/3 is such that

- $\mathsf{T}_1 = \langle \widehat{P}_1 \rangle$ is a technique defining the flow of execution, by attempting to prove a given formula (symptom, disease, conjunction or disjunction);

- $\mathsf{T}_2 = \langle \widehat{P}_1, \widehat{P}_2 \rangle$ is a technique that calculates the certainty factor of a formula using fuzzy reasoning. It employs values supplied by T_1 above.
- $\mathsf{T}_3 = \langle \widehat{P}_1, \widehat{P}_2 \widehat{P}_3 \rangle$ builds a proof-tree as the proof proceeds, employing values provided by T_1 and T_2 above.

Appendix B

δ-Relations and Examples of Abstractions

In this appendix we show the formal definitions of the δ-relations used during the abstraction and reimplementation of the argument slices, described in Section 5.7.6. We also show examples of argument slices being abstracted.

In δ-relations, when we want to refer to portions of the clause whose details are not relevant we employ vectors of clause- and mode-annotated subgoals $\vec{S}_i$ and $\vec{S}_i^\alpha$ of the following form

$$\vec{S}_i = \boxed{\gamma_{[i,0]}\ \theta_{[i,0]}\ S_{[i,0]}\ \theta'_{[i,0]}\ \ \gamma'_{[i,0]}}, \ldots, \boxed{\gamma_{[i,n_i]}\ \ \theta_{[i,n_i]}\ S_{[i,n_i]}\ \gamma'_{[i,n_i]}\ \ \theta'_{[i,n_i]}}$$
$$\vec{S}_i^\alpha = \boxed{\gamma_{[i,0]}\ \theta^\alpha_{[i,0]}\ S_{[i,0]}\ \theta'^\alpha_{[i,0]}\ \ \gamma'_{[i,0]}}, \ldots, \boxed{\gamma_{[i,n_i]}\ \ \theta^\alpha_{[i,n_i]}\ S_{[i,n_i]}\ \gamma'_{[i,n_i]}\ \ \theta'^\alpha_{[i,n_i]}}$$

such that $\vec{S}_i^\alpha$ corresponds to $\vec{S}_i$ in the right-hand side of the δ-relations, its abstract version, and it is such that only the mode-annotations are altered. Alternatively, $\vec{S}_i$ can be seen as the more specific version of $\vec{S}_i^\alpha$.

B.1 Predicate Symbol Abstraction

The actual predicate symbol employed in the head goal of an argument slice is not relevant. The different contexts within which programming techniques will be used, techniques editors, program combination systems, tracers, etc., are such that only the *format* of each clause, in terms of numbers of recursive calls, is of importance. We shall denote this by replacing the actual predicate p used in the head goal and in the recursive subgoals for a neutral meta-symbol $\mathcal{P}$.

We define a predicate *pred* between two clause- and mode-annotated argument slices $\widehat{\widehat{P}}_i$ and $\widehat{\widehat{P}}{}_i^\alpha$, such that $pred(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}{}_i^\alpha, p)$ holds if $\widehat{\widehat{P}}_i$ is syntactically equivalent to $\widehat{\widehat{P}}{}_i^\alpha$ if p is consistently replaced by $\mathcal{P}$:

$$\begin{array}{lcl} pred(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}{}_i^\alpha, p) & \Leftarrow & \widehat{\widehat{P}}_i = \langle\ \rangle \wedge \widehat{\widehat{P}}{}_i^\alpha = \widehat{\widehat{P}}_i \\ pred(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}{}_i^\alpha, p) & \Leftarrow & \widehat{\widehat{P}}_i = \langle \widehat{\widehat{C}}, \mathsf{C}\rangle \wedge \widehat{\widehat{P}}{}_i^\alpha = \langle \widehat{\widehat{C}}{}^\alpha, \mathsf{C}^\alpha\rangle \wedge \delta(\widehat{\widehat{C}}, \widehat{\widehat{C}}{}^\alpha, p) \wedge pred(\mathsf{C}, \mathsf{C}^\alpha, p) \end{array}$$

The first line describes the case when both argument slices are empty sequences of clauses: $\widehat{\widehat{P}}_i$ and $\widehat{\widehat{P}}_i^\alpha$ are identical. The second clause recursively defines the relationship between $\widehat{\widehat{P}}_i$ and $\widehat{\widehat{P}}_i^\alpha$ in terms of its sequences of clauses: the first clauses $\widehat{\widehat{C}}$ and $\widehat{\widehat{C}}^\alpha$, of $\widehat{\widehat{P}}_i$ and $\widehat{\widehat{P}}_i^\alpha$ respectively, must be δ-related and the rest of the clauses must recursively satisfy *pred*.

The δ-relation employed above is defined in terms of possible templates for the clauses $\widehat{\widehat{C}}$ and $\widehat{\widehat{C}}^\alpha$. The symbol p is the actual predicate symbol of $\widehat{\widehat{P}}_i$, replaced by $\mathcal{P}$ in $\widehat{\widehat{P}}_i^\alpha$. In the definitions below $\vec{S}$ and $\vec{S}_i, 0 \leq i \leq r$ are possibly empty vectors of clause- and mode-annotated non-recursive subgoals; the first definition of δ caters for recursive clauses:

$$\delta\left(\boxed{\begin{array}{lll} & p(x)\texttt{:-} & \theta_0\ \gamma_0 \\ & \vec{S}_0 & \\ \theta_1^r\ \gamma_1^r & p(x_1)\texttt{,} & \theta_1^{r\prime}\ \gamma_1^{r\prime} \\ & \vec{S}_1 & \\ & \vdots & \\ & \vec{S}_{n-1} & \\ \theta_n^r\ \gamma_n^r & p(x_n)\texttt{,} & \theta_n^{r\prime}\ \gamma_n^{r\prime} \\ & \vec{S}_n & \end{array}}, \boxed{\begin{array}{lll} & \mathcal{P}(x)\texttt{:-} & \theta_0\ \gamma_0 \\ & \vec{S}_0 & \\ \theta_1^r\ \gamma_1^r & \mathcal{P}(x_1)\texttt{,} & \theta_1^{r\prime}\ \gamma_1^{r\prime} \\ & \vec{S}_1 & \\ & \vdots & \\ & \vec{S}_{n-1} & \\ \theta_n^r\ \gamma_n^r & \mathcal{P}(x_n)\texttt{,} & \theta_n^{r\prime}\ \gamma_n^{r\prime} \\ & \vec{S}_n & \end{array}}, p\right)$$

The second formulation for δ addresses non-recursive clauses:

$$\delta\left(\boxed{\begin{array}{l} p(x)\texttt{:-}\ \theta_0\ \gamma_0 \\ \quad \vec{S} \end{array}}, \boxed{\begin{array}{l} \mathcal{P}(x)\texttt{:-}\ \theta_0\ \gamma_0 \\ \quad \vec{S} \end{array}}, p\right)$$

B.2 E-form δ-Relations

In this section we formally describe the δ-relations employed to obtain the explicit form (e-form) of argument slices, explained in Section 5.7.6.

Same Variables in the Head Goal and Recursive Subgoal: The use of the same variable in the head goal and in recursive subgoals is a design decision. The programmer decided that the argument position of two subgoals were to be the same, thus sharing an equality relationship. By explicitly using the =/2 operator to describe this design decision, the equality relationship can be abstracted firstly as an *assign* or *bind* then as a *relation* subgoal; during the reimplementation the abstract relationship can be specialised in forms different from the original equality relation between the two argument positions.

When the variable x in the head goal is not free before the recursive subgoal, that is, x is associated with a token different from "**f**" in θ before the recursive call, then the subgoal explicitly binding x and the fresh variable y is inserted *before* the recursive subgoal. The new argument slice with the new clause bearing the inserted subgoal is equivalent to the old one: the recursive subgoal will be invoked with its variables bearing the same instantiation as before:

$$\boxed{\begin{array}{lll} & p(x)\texttt{:-} & \theta_0\ \gamma_0 \\ & \vec{S}_1 & \\ \gamma\ \theta & p(x)\texttt{,} & \theta'\ \gamma' \\ & \vec{S}_2\texttt{.} & \end{array}} \overset{\delta}{\longleftrightarrow} \boxed{\begin{array}{lll} & p(x)\texttt{:-} & \theta_0^\alpha\ \gamma_0 \\ & \vec{S}_1^\alpha & \\ \gamma\ \theta^\alpha & x\ \texttt{=}\ y\texttt{,} & \theta_*\ \gamma_* \\ \gamma\ \theta_* & p(y)\texttt{,} & \theta'^\alpha\ \gamma' \\ & \vec{S}_2^\alpha\texttt{.} & \end{array}} \qquad \text{(B.1)}$$

where

1. $x/T \in \theta, T \neq \mathbf{f}, x/T' \in \theta'$;
2. $\theta_0^\alpha = \theta_0 \cup \{y/\mathbf{f}\}, \theta_{[1,i]}^\alpha = \theta_{[1,i]} \cup \{y/\mathbf{f}\}, \theta'^\alpha_{[1,i]} = \theta'_{[1,i]} \cup \{y/\mathbf{f}\}, 0 \leq i \leq n_1, \theta^\alpha = \theta \cup \{y/\mathbf{f}\}$;
3. $\theta_* = \theta \cup \{y/T\}$;
4. $\theta'^\alpha = \theta' \cup \{y/T'\}, \theta_{[2,i]}^\alpha = \theta_{[2,i]} \cup \{y/T'\}, \theta'^\alpha_{[2,i]} = \theta'_{[2,i]} \cup \{y/T'\}, 0 \leq i \leq n_2$.
5. $\gamma = \langle\!\langle \mathit{required}(\emptyset) \rangle\!\rangle, \gamma' = \langle\!\langle \mathit{offer}(\emptyset) \rangle\!\rangle, \gamma_* = \langle\!\langle \mathit{offer}(\{y\}) \rangle\!\rangle$

The first condition describes the instantiation status x must have: it should not be free before the recursive call where x appears again; x is associated with token T' in θ'. The second condition describes the relationship between the mode-annotations before the analysed recursive call: the mode-annotations $\theta_0^\alpha, \theta_{[1,i]}^\alpha$ and θ^α on the right-hand side consist of $\theta_0, \theta_{[1,i]}$ and θ, respectively, together with the pair $y/\mathbf{f}$ depicting the auxiliary variable y used in the explicit binding "x = y" initially free; the clause-annotations remain the same. The third condition describes the relation between the mode-annotations after the "x = y" subgoal: θ_* is equal to θ together with the pair $\{y/T\}$, where T is the token associated to x before the recursive call (first condition). The fourth condition depicts the relation between the mode-annotations after the analysed recursive call: θ'^α, $\theta_{[2,i]}^\alpha$ and $\theta'^\alpha_{[2,i]}$ consist, respectively, of θ', $\theta_{[2,i]}$ and $\theta'_{[2,i]}$ together with y/T' depicting y with the instantiation mode of x after the recursive call (first condition). The fifth condition shows the values of γ, γ' and γ_*.

When a variable x in the head goal is free before its new occurrence in a recursive subgoal, that is, x is associated with "**f**" in θ, then the subgoal "x = y" is inserted *after* the recursive subgoal. In terms of equivalence of procedural meaning (execution), the position of this new subgoal in relation to the recursive call is irrelevant: the recursive goal will be invoked with its variables bearing the same instantiation modes and contents. If, however, "x = y" is inserted after the recursive call, then the reimplementations of its abstraction may employ y possibly with a changed instantiation status: if it is ground, then any one of the more specialised descriptors (even the stringent descriptor *calc*) can be employed:

$$\boxed{\begin{array}{lll} & p(x)\texttt{:-} & \theta_0\ \gamma_0 \\ & \vec{S}_1 & \\ \gamma\ \theta & p(x)\texttt{,} & \theta'\ \gamma' \\ & \vec{S}_2\texttt{.} & \end{array}} \quad \stackrel{\delta}{\longleftrightarrow} \quad \boxed{\begin{array}{lll} & p(x)\texttt{:-} & \theta_0^\alpha\ \gamma_0 \\ & \vec{S}_1^\alpha & \\ \gamma\ \theta^\alpha & p(y)\texttt{,} & \theta'^\alpha\ \gamma'^\alpha \\ \gamma\ \theta'^\alpha & x\ \texttt{=}\ y\texttt{,} & \theta_*\ \gamma' \\ & \vec{S}_2^\alpha\texttt{.} & \end{array}} \tag{B.2}$$

where

1. $x/\mathbf{f} \in \theta, x/T' \in \theta'$;
2. $\theta_0^\alpha = \theta_0 \cup \{y/\mathbf{f}\}, \theta_{[1,i]}^\alpha = \theta_{[1,i]} \cup \{y/\mathbf{f}\}, \theta'^\alpha_{[1,i]} = \theta'_{[1,i]} \cup \{y/\mathbf{f}\}, 0 \leq i \leq n_1, \theta^\alpha = \theta \cup \{y/\mathbf{f}\}$;
3. $\theta'^\alpha = \theta \cup \{y/T'\}, \theta_* = \theta' \cup \{y/T'\}, \theta_{[2,i]}^\alpha = \theta_{[2,i]} \cup \{y/T'\}, \theta'^\alpha_{[2,i]} = \theta'_{[2,i]} \cup \{y/T'\}, 0 \leq i \leq n_2$.

4. $\gamma = \langle\!\langle \mathit{required}(\emptyset) \rangle\!\rangle, \gamma'^{\alpha} = \langle\!\langle \mathit{offer}(\{y\}) \rangle\!\rangle$

The first condition tells us that this δ-relation holds if x is free before the recursive call where it appears again. The relationship between the mode-annotations before the analysed recursive call in the second item are similar to that of the previous δ-relation: $\theta^{\alpha}_{[1,i]}$ and $\theta'^{\alpha}_{[1,i]}$ consists of, respectively, $\theta_{[1,i]}$ and $\theta'_{[1,i]}$ together with the pair $y/\mathbf{f}$ depicting the auxiliary variable y initially free used in the explicit binding "x = y". The second condition relates the θ's before the recursive subgoal being analysed: the mode-annotations in the right-hand side construct are the corresponding mode-annotations together with the pair $y/\mathbf{f}$ denoting the auxiliary variable y initially free. The third condition describes the relation between the mode-annotations after the analysed recursive subgoal: θ'^{α} is θ added with the pair y/T', where T' is x's mode in θ' (first condition); it also relates $\theta^{\alpha}_{[2,i]}$ and $\theta'^{\alpha}_{[2,i]}$, respectively, with $\theta_{[2,i]}$ and $\theta'_{[2,i]}$ together with the pair y/T'. The last condition describes the formats of γ and γ'^{α}.

Example: The following δ-relation holds between the following clause of the second argument slice of *collect/2* (left-hand side) and its more abstract version:

$$\boxed{\begin{array}{ll} & \mathcal{P}\texttt{(B) :- \{B/f\}} \\ \texttt{\{B/f\}} & \mathcal{P}\texttt{(B).\{B/g\}}\ \langle\!\langle \mathit{offer}(\texttt{\{B\}}) \rangle\!\rangle \end{array}} \quad \overset{\delta}{\longleftrightarrow} \quad \boxed{\begin{array}{ll} & \mathcal{P}\texttt{(B) :- \{B/f,C/f\}} \\ \texttt{\{B/f,C/f\}} & \mathcal{P}\texttt{(C), \{B/f,C/g\}}\ \langle\!\langle \mathit{offer}(\texttt{\{C\}}) \rangle\!\rangle \\ \texttt{\{B/f,C/g\}} & \texttt{B = C.\{B/g,C/g\}}\ \langle\!\langle \mathit{offer}(\texttt{\{B\}}) \rangle\!\rangle \end{array}}$$

Implicit Testing in a Data Structure Decomposition: When a subgoal performing a data structure decomposition is such that one of the variables in its pattern is not associated with token "**f**", then possibly a test is also being implicitly performed. The δ-relation below replaces the variable y of the data structure pattern which is not associated with "**f**" in θ for a fresh variable z associated with "**f**", and then inserts a subgoal of the form "y = z". All the other mode-annotations of the clause have to be altered to accommodate the newly introduced variable z:

$$\boxed{\begin{array}{lll} & H\texttt{:-} & \theta_0\ \gamma_0 \\ & \vec{S}_1 & \\ \gamma\ \theta & x\ \texttt{=}\ f\texttt{(}\vec{V}_1\ y\ \vec{V}_2\texttt{)}, & \theta'\ \gamma' \\ & \vec{S}_2\,\texttt{.} & \end{array}} \quad \overset{\delta}{\longleftrightarrow} \quad \boxed{\begin{array}{lll} & H\texttt{:-} & \theta^{\alpha}_0\ \gamma_0 \\ & \vec{S}^{\alpha}_1 & \\ \gamma^{\alpha}\ \theta^{\alpha} & x\ \texttt{=}\ f\texttt{(}\vec{V}_1\ z\ \vec{V}_2\texttt{)}, & \theta_*\ \gamma'^{\alpha} \\ \gamma_*\ \theta_* & y\ \texttt{=}\ z, & \theta'^{\alpha}\ \gamma'_* \\ & \vec{S}^{\alpha}_2\,\texttt{.} & \end{array}} \qquad \text{(B.3)}$$

where

1. $x/T_x \in \theta, T_x \neq \mathbf{f}, x/T'_x \in \theta', y/T_y \in \theta, T_y \neq \mathbf{f}, y/T'_y \in \theta'$;

2. $T^{\alpha}_x \sqsubseteq T_x, T^{\alpha}_x \sqsubseteq T'_x, T^{\alpha}_y \sqsubseteq T_y, T^{\alpha}_y \sqsubseteq T'_y$;

3. $\theta^{\alpha}_0 = \theta_0 \cup \{z/\mathbf{f}\}, \theta^{\alpha}_{[1,j]} = \theta_{[1,j]} \cup \{z/\mathbf{f}\}, \theta'^{\alpha}_{[1,j]} = \theta'_{[1,j]} \cup \{z/\mathbf{f}\}, 0 \leq j \leq n_1, \theta^{\alpha} = \theta \cup \{z/\mathbf{f}\}$,

4. $\theta_* = (\theta - \{x/T_x\}) \cup \{x/T^{\alpha}_x, z/T^{\alpha}_y\}$;

5. $\theta'^{\alpha} = \theta' \cup \{z/T^{\alpha}_y\}, \theta^{\alpha}_{[2,j]} = \theta_{[2,j]} \cup \{z/T^{\alpha}_y\}, \theta'^{\alpha}_{[2,j]} = \theta'_{[2,j]} \cup \{z/T^{\alpha}_y\}, 0 \leq j \leq n_2$.

6. $\gamma = \langle\!\langle \mathit{required}(Z) \rangle\!\rangle$

   (a) if $y \in Z$ then $\gamma^{\alpha} = \langle\!\langle \mathit{required}(Z') \rangle\!\rangle, Z' = Z - \{y\}, \gamma_* = \langle\!\langle \mathit{required}(\{y\}) \rangle\!\rangle$;

(b) if $y \notin Z$ then $\gamma^\alpha = \gamma, \gamma_* = \langle\!\langle \mathit{required}(\emptyset) \rangle\!\rangle$;

7. $\gamma' = \langle\!\langle \mathit{offer}(V) \rangle\!\rangle, \gamma'^\alpha = \langle\!\langle \mathit{offer}(W) \rangle\!\rangle, W = V \cup \{z\}, \gamma'_* = \langle\!\langle \mathit{offer}(\emptyset) \rangle\!\rangle$

The first item describes the conditions the tokens associated with x and y before and after the subgoal "$x = f(\vec{V}_1\ y\ \vec{V}_2)$" must fulfil. The second condition establishes that T_x^α subsumes T_x and T'_x and T_y^α subsumes T_y and T'_y: the changes in the modes of these variables must be properly captured by this δ-relation. The third condition shows the relations between the mode-annotations before the subgoals under analysis: they are similar, with the addition of the pair $z/\mathbf{f}$ in the mode-annotations of the right-hand side of the δ-relation. The fourth condition describes θ_* in terms of θ: the pair x/T_x is removed from θ and z/T_y^α and x/T_x^α are inserted. The fifth condition defines the relation between those mode-annotations after the analysed subgoals: they are similar, with the exception of an additional element z/T_y^α appearing in those mode-annotations on the right-hand side of the δ-relation. The sixth item describes the relations between γ, γ^α and γ_*: the variable y, if required before the subgoal "$x = f(\vec{V}_1\ y\ \vec{V}_2)$" must be removed from it, since it will be replaced by a fresh variable z and hence is not required any longer, and it must be inserted in γ_*, the clause-annotation preceding the inserted subgoal "y =z"; if y is not in Z then the clause-annotations remain as they were and γ_* has an empty set. The last condition shows the relationships between γ', γ'^α and γ'_*: the fresh variable z is appropriately inserted in γ'^α and an empty set is assigned to γ'_*.

Example: The following δ-relation holds between the following clause of the second argument slice of *prefix/2* and its more abstract version:

```
                             prefix(B):- {B/g,C/f,D/f} ⟨⟨ offer({B}) ⟩⟩
⟨⟨ required({C}) ⟩⟩ {B/g,C/g,D/f}  B = [C|D],{B/g,C/g,D/g}⟨⟨ offer({D}) ⟩⟩
                    {B/g,C/g,D/g}  prefix(D).{B/g,C/g,D/g}
```

$\stackrel{\delta}{\longleftrightarrow}$

```
                                  prefix(B):- {B/g,C/f,D/f,E/f}  ⟨⟨ offer({B}) ⟩⟩
                        {B/g,C/g,D/f,E/f}  B = [E|D],{B/g,C/g,D/g,E/g}⟨⟨ offer({D,E}) ⟩⟩
⟨⟨ required({C}) ⟩⟩{B/g,C/g,D/g,E/g}  C = E,     {B/g,C/g,D/g,E/g}
                        {B/g,C/g,D/g,E/g}  prefix(D).{B/g,C/g,D/g,E/g}
```

Implicit Testing on Data Structures being Built: This δ-relation addresses the case when a data structure $f(\vec{V})$ is tested against a non-free variable x: it replaces x for a fresh variable y and then inserts a subgoal y = x performing an explicit test equivalent to that implicitly carried out in the pattern matching:

$$\boxed{\begin{array}{lll} & H\texttt{:-} & \theta_0\ \gamma_0 \\ & \vec{S}_1 & \\ \gamma\ \theta & x\ \texttt{=}\ f(\vec{V})\ \texttt{,} & \theta'\ \gamma' \\ & \vec{S}_2\ \texttt{.} & \end{array}} \quad \stackrel{\delta}{\longleftrightarrow} \quad \boxed{\begin{array}{lll} & H\texttt{:-} & \theta_0^\alpha\ \gamma_0 \\ & \vec{S}_1 & \\ \gamma^\alpha\ \theta^\alpha & y\ \texttt{=}\ f(\vec{V})\ \texttt{,} & \theta'^\alpha\ \gamma'^\alpha \\ \gamma_*\ \theta'^\alpha & y\ \texttt{=}\ x\ \texttt{,} & \theta'_*\ \gamma' \\ & \vec{S}_2\ \texttt{.} & \end{array}} \qquad \text{(B.4)}$$

where

1. $x/T \in \theta, T \neq \mathbf{f}, x/T' \in \theta'$ and for each variable z in $\vec{V}$, it is the case that $\neg \mathit{change}(z, \theta, \theta')$;

2. $T^\alpha \sqsubseteq T, T^\alpha \sqsubseteq T'$;

3. $\theta^\alpha_0 = \theta_0 \cup \{y/\mathbf{f}\}, \theta^\alpha_{[1,i]} = \theta_{[1,i]} \cup \{y/\mathbf{f}\}, {\theta'^\alpha}_{[1,i]} = \theta'_{[1,i]} \cup \{y/\mathbf{f}\}, 0 \leq i \leq n_1, \theta^\alpha = \theta \cup \{y/\mathbf{f}\}$;

4. $\theta'^\alpha = \theta \cup \{y/T^\alpha\}, \theta'_* = (\theta - \{x/T\}) \cup \{y/T^\alpha, x/T^\alpha\}, \theta^\alpha_{[2,i]} = \theta_{[2,i]} \cup \{y/T^\alpha\}, {\theta'}^\alpha_{[2,i]} = \theta'_{[2,i]} \cup \{y/T^\alpha\}, 0 \leq i \leq n_2$.

5. $\gamma = \langle\!\langle \mathit{required}(Z) \rangle\!\rangle$

   (a) if $x \in Z$ then $\gamma^\alpha = \langle\!\langle \mathit{required}(Z') \rangle\!\rangle, Z' = Z - \{x\}, \gamma_* = \langle\!\langle \mathit{required}(\{x\}) \rangle\!\rangle$;

   (b) if $x \notin Z$ then $\gamma^\alpha = \gamma, \gamma_* = \langle\!\langle \mathit{required}(\emptyset) \rangle\!\rangle$;

6. $\gamma'^\alpha = \langle\!\langle \mathit{offer}(\{y\}) \rangle\!\rangle$

These conditions are similar to those of the previous δ-relation. The first item lists the conditions of the tokens associated to x in the mode-annotations before and after the analysed subgoal, as well as the modes of those variables within the data structure pattern. The second condition draws relationships between T^α and T and T'. The third condition shows the relationships between the mode-annotations before the analysed subgoal: those of the left-hand side clause are equal to those of the right-hand side clause if the pair $y/\mathbf{f}$ is appended to them. The fourth condition shows the relationships between the mode-annotations of the subgoals being analysed and those of the subgoals following them. The fifth condition depicts the relationship between γ, γ^α and γ_*. The sixth item depicts γ'^α.

Implicit Testing in Arithmetic Calculations: This δ-relation addresses those subgoals performing the arithmetic calculation of an expression $f(\vec{V})$ via the *is/2* operator and compares the final result with a non-free variable x; complex subgoals of this sort can be replaced by two simpler subgoals, one performing the computation and storing its value in a free variable y and another subgoal explicitly testing y and x:

$$\boxed{\begin{array}{lll} & H\texttt{:-} & \theta_0\ \gamma_0 \\ & \vec{S}_1 & \\ \gamma\ \theta & x\ \texttt{is}\ f(\vec{V})\texttt{,} & \theta'\ \gamma' \\ & \vec{S}_2\texttt{.} & \end{array}} \stackrel{\delta}{\longleftrightarrow} \boxed{\begin{array}{lll} & H\texttt{:-} & \theta^\alpha_0\ \gamma_0 \\ & \vec{S}_1 & \\ \gamma^\alpha\ \theta^\alpha & y\ \texttt{is}\ f(\vec{V})\texttt{,} & \theta'^\alpha\ \gamma'^\alpha \\ \gamma_*\ \theta'^\alpha & y\ \texttt{=}\ x\texttt{,} & \theta'_*\ \gamma' \\ & \vec{S}_2\texttt{.} & \end{array}} \tag{B.5}$$

where

1. $x/T \in \theta, T \neq \mathbf{f}, x/T' \in \theta'$ and for each variable z in $\vec{V}$, it is the case that $\neg \mathit{change}(z, \theta, \theta')$;

2. $T^\alpha \sqsubseteq T, T^\alpha \sqsubseteq T'$;

3. $\theta^\alpha_0 = \theta_0 \cup \{y/\mathbf{f}\}, \theta^\alpha_{[1,i]} = \theta_{[1,i]} \cup \{y/\mathbf{f}\}, {\theta'_{[1,i]}}^\alpha = \theta'_{[1,i]} \cup \{y/\mathbf{f}\}, 0 \leq i \leq n_1, \theta^\alpha = \theta \cup \{y/\mathbf{f}\}$;

4. $\theta'^\alpha = \theta \cup \{y/T^\alpha\}, \theta'_* = (\theta - \{x/T\}) \cup \{y/T^\alpha, x/T^\alpha\}, \theta^\alpha_{[2,i]} = \theta_{[2,i]} \cup \{y/T^\alpha\}, {\theta'}^\alpha_{[2,i]} = \theta'_{[2,i]} \cup \{y/T^\alpha\}, 0 \leq i \leq n_2$.

5. $\gamma = \langle\!\langle \mathit{required}(Z) \rangle\!\rangle$

   (a) if $x \in Z$ then $\gamma^\alpha = \langle\!\langle \mathit{required}(Z') \rangle\!\rangle, Z' = Z - \{x\}, \gamma_* = \langle\!\langle \mathit{required}(\{x\}) \rangle\!\rangle$;

   (b) if $x \notin Z$ then $\gamma^\alpha = \gamma, \gamma_* = \langle\!\langle \mathit{required}(\emptyset) \rangle\!\rangle$;

6. $\gamma'^\alpha = \langle\!\langle \mathit{offer}(\{y\}) \rangle\!\rangle$.

This δ-relation is similar to the previous one, differing only in the predicates *=/2* and *is/2*.

Implicit Testing in $\Im$ Predicates: This δ-relation addresses those subgoals employing an $\Im$ input predicate (*read/1* or *get/1*) which obtains a value and compares it with the value of the non-free variable x it is invoked with, thus implicitly performing a test. Such subgoals are replaced by two simpler subgoals, one performing the data input with a free variable y receiving it, and another explicitly testing x and y:

$$\boxed{\begin{array}{ll} & H\texttt{:-} \quad \theta_0\ \gamma_0 \\ & \vec{S}_1 \\ \gamma\ \theta & \Im(x)\texttt{,}\ \ \theta'\ \gamma' \\ & \vec{S}_2\texttt{.} \end{array}} \quad \stackrel{\delta}{\longleftrightarrow} \quad \boxed{\begin{array}{ll} & H\texttt{:-} \quad \theta_0^\alpha\ \gamma_0 \\ & \vec{S}_1 \\ \gamma^\alpha\ \theta^\alpha & \Im(y)\texttt{,}\ \ \theta'^\alpha\ \gamma'^\alpha \\ \gamma_*\ \theta'^\alpha & y\ \texttt{=}\ x\texttt{,}\ \ \theta'_*\ \gamma' \\ & \vec{S}_2\texttt{.} \end{array}} \tag{B.6}$$

where

1. $x/T \in \theta, T \neq \mathbf{f}, x/T' \in \theta'$;
2. $T^\alpha \sqsubseteq T, T^\alpha \sqsubseteq T'$;
3. $\theta_0^\alpha = \theta_0 \cup \{y/\mathbf{f}\}, \theta_{[1,i]}^\alpha = \theta_{[1,i]} \cup \{y/\mathbf{f}\}, {\theta'_{[1,i]}}^\alpha = \theta'_{[1,i]} \cup \{y/\mathbf{f}\}, 0 \leq i \leq n_1, \theta^\alpha = \theta \cup \{y/\mathbf{f}\}$;
4. $\theta'^\alpha = \theta \cup \{y/T^\alpha\}, \theta'_* = (\theta - \{x/T\}) \cup \{y/T^\alpha, x/T^\alpha\}, \theta_{[2,i]}^\alpha = \theta_{[2,i]} \cup \{y/T^\alpha\}, \theta'^\alpha_{[2,i]} = \theta'_{[2,i]} \cup \{y/T^\alpha\}, 0 \leq i \leq n_2$.
5. $\gamma = \langle\!\langle \mathit{required}(Z) \rangle\!\rangle$
   (a) if $x \in Z$ then $\gamma^\alpha = \langle\!\langle \mathit{required}(Z') \rangle\!\rangle, Z' = Z - \{x\}, \gamma_* = \langle\!\langle \mathit{required}(\{x\}) \rangle\!\rangle$;
   (b) if $x \notin Z$ then $\gamma^\alpha = \gamma, \gamma_* = \langle\!\langle \mathit{required}(\emptyset) \rangle\!\rangle$;
6. $\gamma'^\alpha = \langle\!\langle \mathit{offer}(\{y\}) \rangle\!\rangle$.

Unused Head Goal Arguments: Those variables in the head goal which are not used within a clause may be useful when the procedure is being reimplemented. Those variables can be seen as satisfying a "dummy" *true* subgoal which always succeeds; the δ-relation below inserts a *true/1* subgoal for each variable in the head goal not used within a clause. The inserted *true* subgoals, once abstracted, allow different possible redefinitions:

$$\boxed{\begin{array}{l} p(x)\texttt{:-}\ \theta_0\ \gamma_0 \\ \quad \vec{S} \end{array}} \quad \stackrel{\delta}{\longleftrightarrow} \quad \boxed{\begin{array}{ll} & p(x)\texttt{:-} \quad \theta_0\ \gamma_0 \\ \gamma\ \theta_0 & \mathit{true}(x)\texttt{,}\ \ \theta_0\ \gamma' \\ & \vec{S} \end{array}} \tag{B.7}$$

where

1. x does not occur in any subgoal of vector $\vec{S}$;
2. $\gamma = \langle\!\langle \mathit{required}(\emptyset) \rangle\!\rangle, \gamma' = \langle\!\langle \mathit{offer}(\emptyset) \rangle\!\rangle$.

If the clause has an empty body, then the following alternative formulation of the relation above is applicable:

$$\boxed{p(x)\texttt{.}\ \theta_0\ \gamma_0} \quad \stackrel{\delta}{\longleftrightarrow} \quad \boxed{\begin{array}{ll} & p(x)\texttt{:-} \quad \theta_0\ \gamma_0 \\ \gamma\ \theta_0 & true(x)\texttt{.}\ \theta_0\ \gamma' \end{array}} \tag{B.8}$$

Example: The δ-relation above holds between the following clause of the second argument slice of procedure *prefix/2* (left-hand side) and its more abstract version:

$$\boxed{\mathcal{P}(\texttt{B})\texttt{.}\ \{\texttt{B/g}\}\ \langle\!\langle \mathit{offer}(\{\texttt{B}\})\rangle\!\rangle} \quad \stackrel{\delta}{\longleftrightarrow} \quad \boxed{\begin{array}{ll} & \mathcal{P}(\texttt{B})\texttt{:-} \quad \{\texttt{B/g}\}\ \langle\!\langle \mathit{offer}(\{\texttt{B}\})\rangle\!\rangle \\ \{\texttt{B/g}\} & true(\texttt{B})\texttt{.}\ \{\texttt{B/g}\} \end{array}}$$

The clause on the right-hand side can have its *true* subgoal abstracted and reimplemented as a different form of test.

B.3 Subgoal δ-Relations

In this subsection we define δ-relations relating clause- and mode-annotated subgoals and their abstracted forms. When employed to abstract a subgoal the δ-relation rewrites the subgoal as a descriptor, an informative predicate name conveying the procedural meaning of the subgoal. The clause- and mode-annotations remain unchanged. Every possible combination of predicates and modes was addressed. We have grouped the δ-relations together depending on the descriptor in their right-hand side. The relations presented here are aimed at procedures in their e-form form.

In some cases, a subgoal can be accurately abstracted as one of the descriptors. These subgoals employ system predicates or operators whose computational meaning is well-defined and known in advance. For instance, those subgoals employing operators >*/2*, <*/2*, and so on, are uniquely abstracted as a *test*, despite their mode-annotations. The mode-annotations, in other cases, complement the syntax of the subgoal, ruling out possible meanings.

δ-Relations for the *test* Descriptor: The *test* descriptor stands for all subgoals used as tests, that is, those subgoals that may alter the flow of execution of a procedure by causing a failure. All those system operators $\diamond$ (Section 1.5) different from =*/2* and =..*/2* are necessarily tests, despite their mode-annotations, that is

$$\boxed{\gamma\,\theta\ x \diamond a\ \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ test(\{x\})\ \theta'\ \gamma'} \tag{B.9}$$

where $\diamond \notin \{\texttt{=}, \texttt{=..}\}$. The constant a is an arbitrary construct and is not part of the abstraction. If this relation is used to specify a *test* subgoal, then a must be provided by the user. If two variables x_1 and x_2 are employed in a concrete subgoal employing the same operator $\diamond$ as above, then both must appear in the *test* descriptor:

$$\boxed{\gamma\,\theta\ x_1 \diamond x_2\ \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ test(\{x_1, x_2\})\ \theta'\ \gamma'} \tag{B.10}$$

where $\diamond \notin \{\texttt{=}, \texttt{=..}\}$.

When =*/2* or =..*/2* are used in a subgoal, then its mode-annotations may help to decide which computation is being carried out; if the subgoal is of the form $x \diamond a$ and the tokens associated with variable x do not satisfy the *change* relationship, we shall consider this subgoal as a *test*:

$$\boxed{\gamma\,\theta\; x \diamond a\; \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\;\theta\; test(\{x\})\;\theta'\;\gamma'} \tag{B.11}$$

where $\diamond \in \{\texttt{=}, \texttt{=..}\}$ and $\neg change(x, \theta, \theta')$. One should notice that if $unknown(x, \theta, \theta')$, then this relation is still applicable. If the subgoal is of the form $x_1 \diamond x_2$, then in order to be abstracted as a *test*, both its variables are such that their associated tokens do not satisfy the *change* relationship and are not free:

$$\boxed{\gamma\,\theta\; x_1 \diamond x_2\; \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\;\theta\; test(\{x_1, x_2\})\;\theta'\;\gamma'} \tag{B.12}$$

where $\diamond \in \{\texttt{=}, \texttt{=..}\}, \neg change(x_i, \theta, \theta'), x_i/T \in \theta, T \neq \mathbf{f}, i = 1, 2$: if the variables remain free throughout the subgoal execution, then the *bind* descriptor (Def.: B.21) should be used instead. Both variables appear in the *test* descriptor.

The ¶ system predicates (Section 1.5) are all tests, despite their mode-annotations:

$$\boxed{\gamma\,\theta\;\P(x)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\;\theta\; test(\{x\})\;\theta'\;\gamma'} \tag{B.13}$$

Since we assume that the procedure to be abstracted is in its e-form, we need not consider the subgoals $\boxed{\theta\,\Im(x)\,\theta'}$ and $\boxed{\theta\, x\ \texttt{is}\ f(y_1, \ldots, y_n)\,\theta'}$, $\neg change(x, \theta, \theta')$, because, although they are tests, they would have been changed by the subgoal-inserting relations B.1 to B.8.

δ-Relations for the *assign* Descriptor: The *assign* predicate descriptor abstracts those subgoals that may perform changes in the instantiation status of its variables. The *=/2* and *=../2* operators are of paramount importance as far as assignments and design decisions are concerned, since the procedures dealt with here are in their e-form. If the subgoal is of the form $x \diamond a$ and the tokens associated with variable x do not satisfy the *fixed* relationship, we shall consider this subgoal as an assignment:

$$\boxed{\gamma\,\theta\; x \diamond a\; \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\;\theta\; assign(\emptyset, \{x\})\;\theta'\;\gamma'} \tag{B.14}$$

where $\diamond \in \{\texttt{=}, \texttt{=..}\}$ and $\neg fixed(x, \theta, \theta')$. If the subgoal is of the form $x_1 \diamond x_2$, then in order to be abstracted as an *assign*, one of its variables x_i is such that its associated tokens do not satisfy the *fixed* relationship:

$$\boxed{\gamma\,\theta\; x_1 \diamond x_2\; \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\;\theta\; assign(\{x_j\}, \{x_i\})\;\theta'\;\gamma'} \tag{B.15}$$

where $\diamond \in \{\texttt{=}, \texttt{=..}\}$ and $\neg fixed(x_i, \theta, \theta'), i \neq j$. This relation does not pose any constraint on x_j, apart from it being different from x_i. The system predicates $\Im$ (Section 1.5) are also assignments

$$\boxed{\gamma\,\theta\;\Im(x)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\;\theta\; assign(\emptyset, \{x\})\;\theta'\;\gamma'} \tag{B.16}$$

where $\neg fixed(x, \theta, \theta')$; since the procedures are in their e-form, it is the case that $\Im$ subgoals always satisfy this constraint.

δ-Relations for Other Descriptors: The remaining descriptors have their δ-relations explained here. The *decomp* descriptor abstracts those subgoals of the form "$x = f(\vec{V})$" such that x does not satisfy the *change* relationship, that is,

$$\boxed{\gamma\,\theta\; x = f(y_1, \ldots, y_n)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\; decomp(x, \{y_1, \ldots, y_n\})\;\theta'\,\gamma'} \qquad \text{(B.17)}$$

where $\neg change(x, \theta, \theta')$. The same subgoal may be abstracted as the *build* descriptor, if x does not satisfy the *fixed* relationship, that is,

$$\boxed{\gamma\,\theta\; x = f(y_1, \ldots, y_n)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\; build(\{y_1, \ldots, y_n\}, x)\;\theta'\,\gamma'} \qquad \text{(B.18)}$$

where $\neg fixed(x, \theta, \theta')$. If, instead of the *=/2* operator, the arithmetic operator *is/2* is used, then it is the case that "x `is` $f(\vec{V})$" is abstracted as the *calc* descriptor:

$$\boxed{\gamma\,\theta\; x \texttt{ is } f(y_1, \ldots, y_n)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\; calc(\{y_1, \ldots, y_n\}, x)\;\theta'\,\gamma'} \qquad \text{(B.19)}$$

where $\neg fixed(x, \theta, \theta')$. Those subgoals employing user-defined predicates are simply abstracted onto the *user-pred* descriptor:

$$\boxed{\gamma\,\theta\; p(x_1, \ldots, x_n)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\; user\text{-}pred(\{x_1, \ldots, x_n\})\;\theta'\,\gamma'} \qquad \text{(B.20)}$$

where $\neg system(p^n)$. If two variables x_1 and x_2 are associated with tokens "**f**" in the mode-annotations before and after a subgoal, then this subgoal can be abstracted as the *bind* descriptor:

$$\boxed{\gamma\,\theta\; x_1 = x_2\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\; bind(x_1, x_2)\;\theta'\,\gamma'} \qquad \text{(B.21)}$$

where $x_i/\mathbf{f} \in \theta, x_i/\mathbf{f} \in \theta', i = 1, 2$.

δ-Relations for the *relation* Descriptor: The predicate descriptors, conveying the procedural meaning of the subgoals, can be further abstracted as the *relation* descriptor, providing an even less committed account of the design decisions of the program. The *relation* descriptor stands for a generic declarative description of the subgoals: its sets of variables and their mode-annotations preserve the procedural aspect of the subgoal and serve as constraints in future reimplementations.

The δ-relations above may have further restrictions as to how the variables in each descriptor are grouped together: this depends on the tokens associated with each variable. The *test* descriptor relates to *relation* in a straightforward way:

$$\boxed{\gamma\,\theta\; test(W)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\; relation(W)\;\theta'\,\gamma'} \qquad \text{(B.22)}$$

where $x \in W, \neg change(x, \theta, \theta')$; *test* subgoals should not change the instantiation status of their variables W. The arguments x and V of *decomp* are merged as W in *relation*:

$$\boxed{\gamma\,\theta\; decomp(x, V)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\; relation(W)\;\theta'\,\gamma'} \qquad \text{(B.23)}$$

where $W = V \cup \{x\}, \neg change(x, \theta, \theta')$, and $y \in V, \neg fixed(y, \theta, \theta')$; the variables $y \in V$ have their values changed and x remains unchanged. In *build*, the converse holds:

$$\boxed{\gamma\,\theta\; build(V, x)\;\theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\; relation(W)\;\theta'\,\gamma'} \qquad \text{(B.24)}$$

where $W = V \cup \{x\}, \neg\mathit{fixed}(x, \theta, \theta'), y \in V, \neg\mathit{change}(y, \theta, \theta')$; the variables $y \in V$ remain unchanged while x has its value changed. A similar δ-relation holds between *calc* and *relation*:

$$\boxed{\gamma\,\theta\;\mathit{calc}(V, x)\;\theta'\,\gamma'} \overset{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\;\mathit{relation}(W)\;\theta'\,\gamma'} \tag{B.25}$$

where $W = V \cup \{x\}, \neg\mathit{fixed}(x, \theta, \theta'), y \in V, \neg\mathit{change}(y, \theta, \theta')$. The sets of variables of the *assign* descriptor are merged:

$$\boxed{\gamma\,\theta\;\mathit{assign}(V_1, V_2)\;\theta'\,\gamma'} \overset{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\;\mathit{relation}(W)\;\theta'\,\gamma'} \tag{B.26}$$

where $W = V_1 \cup V_2, x \in V_1, \neg\mathit{change}(x, \theta, \theta'), y \in V_2, \neg\mathit{fixed}(y, \theta, \theta')$; the set V_1 contains those variables which do not change, whereas V_2 contains those variables which do not remain fixed. The *bind* descriptor is such that the following δ-relation holds:

$$\boxed{\gamma\,\theta\;\mathit{bind}(x_1, x_2)\;\theta'\,\gamma'} \overset{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\;\mathit{relation}(\{x_1, x_2\})\;\theta'\,\gamma'} \tag{B.27}$$

where $x_i/\mathbf{f} \in \theta, x_i/\mathbf{f} \in \theta'$. The following δ-relation holds for the *user-pred* descriptor:

$$\boxed{\gamma\,\theta\;\mathit{user\text{-}pred}(W)\;\theta'\,\gamma'} \overset{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\;\mathit{relation}(W)\;\theta'\,\gamma'} \tag{B.28}$$

Finally, the relation below holds for those *true* predicates inserted in the e-form:

$$\boxed{\gamma\,\theta\;\mathit{true}(x)\;\theta'\,\gamma'} \overset{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\;\mathit{relation}(\{x\})\;\theta'\,\gamma'} \tag{B.29}$$

δ-Relation between *relation* Descriptors: two consecutive *relation* subgoals are related to a single *relation* subgoal whose set of variables is given by the union of the sets of variables of the consecutive subgoals, that is,

$$\boxed{\begin{array}{l}\langle\!\langle \mathit{required}(V_1)\rangle\!\rangle\;\theta\;\mathit{relation}(W_1)\;\theta_*\langle\!\langle \mathit{offer}(Z_1)\rangle\!\rangle\\ \langle\!\langle \mathit{required}(V_2)\rangle\!\rangle\;\theta_*\mathit{relation}(W_2)\;\theta'\langle\!\langle \mathit{offer}(Z_2)\rangle\!\rangle\end{array}} \overset{\delta}{\longleftrightarrow} \boxed{\langle\!\langle \mathit{required}(V)\rangle\!\rangle\;\theta\;\mathit{relation}(W)\;\theta'\;\langle\!\langle \mathit{offer}(Z)\rangle\!\rangle} \tag{B.30}$$

where $W = W_1 \cup W_2, V_1 = V \cap W_1, V_2 = V \cap W_2, x \in W, x/T \in \theta, x/T' \in \theta_*, x/T'' \in \theta', T'' \preceq T' \preceq T$. The $\preceq$ relation is as described in Section 3.6.2; it poses additional requirements on the mode-annotations of the variables and the way they can be divided into two subsets with their own mode-annotations. This δ-relation merges the *relation* subgoals when used to abstract the subgoals of a clause, or splits a *relation* subgoal into a sequence of other *relation* subgoals, if used to reimplement the subgoals of a clause.

B.4 Convergence to the Procedural Abstraction

If $\Re^p$ is used to automatically abstract a clause- and mode-annotated argument slice in its e-form $\widehat{\widehat{P}}_i^{[\alpha,e]}$ then the procedural abstraction $\widehat{\widehat{P}}_i^{[\alpha,p]}$ of that component is guaranteed to be eventually reached and is such that $\Re^p$ will not abstract it further. $\widehat{\widehat{P}}_i^{[\alpha,p]}$ has no duplicate clauses and each of its clauses is of the form $\mathcal{P}(x)$:- $\theta'\,\gamma'\,\vec{S}$, $\vec{S}$ being a non-empty vector of clause- and mode-annotated subgoals $\gamma\,\theta\;\mathcal{P}(y)\;\theta'\,\gamma'$ or $\gamma\,\theta\;q(\vec{V})\;\theta'\,\gamma'$ where q is a predicate descriptor different from *relation*. It is possible that there are no recursive subgoals $\mathcal{P}(y)$, but there is at least one subgoal $q(\vec{V})$ in $\vec{S}$.

The $\Re^p$ relation is only applied to those clause and mode-annotated argument slices in their explicit form version. In order to prove that the procedural abstraction is eventually reached, we must first consider the format the e-form version of components have. Our proof will initially address the definition of the *e-form* relation and the format of its second argument. We then employ this intermediate result to sketch a proof that the procedural abstraction is eventually reached.

B.4.1 The *e-form* Relation

The *e-form* predicate, when provided with argument slice $\widehat{\widehat{P}}_i^{[\alpha,0]}$, obtains the construct $\widehat{\widehat{P}}_i^{[\alpha,e]}$, consisting of $\widehat{\widehat{P}}_i^{[\alpha,0]}$ in its explicit form. Its definition, taken from Section 5.7.6, is

$$\begin{array}{rcl} \textit{e-form}(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_i) & \Longleftarrow & \widehat{\widehat{P}}_i = \langle \vec{C}\, \widehat{\widehat{C}}\, \vec{C}' \rangle \wedge \neg(\widehat{\widehat{C}} \overset{\delta}{\longleftrightarrow} \widehat{\widehat{C}}^e) \\ \textit{e-form}(\widehat{\widehat{P}}_i, \widehat{\widehat{P}}_i^{[\alpha,e]}) & \Longleftarrow & \widehat{\widehat{P}}_i = \langle \vec{C}\, \widehat{\widehat{C}}\, \vec{C}' \rangle \wedge \widehat{\widehat{P}}'_i = \langle \vec{C}\, \widehat{\widehat{C}}^e\, \vec{C}' \rangle \wedge \\ & & (\widehat{\widehat{C}} \overset{\delta}{\longleftrightarrow} \widehat{\widehat{C}}^e) \wedge \textit{e-form}(\widehat{\widehat{P}}'_i, \widehat{\widehat{P}}_i^{[\alpha,e]}) \end{array}$$

The employed notation hides the implementational details, enabling us to focus on more relevant issues. The definition above, in its first clause, states that the predicate will have found the e-form of an argument slice, *i.e.* the second argument position is assigned a value, when there are no more e-form δ-relations (Section B.2) applicable to any clause in the argument slice provided as the first argument. The second clause states that if there is an e-form δ-relation which applies to one of the clauses of the given argument slice in the first argument, then a temporary construct $\widehat{\widehat{P}}'_i$ is built replacing the clause $\widehat{\widehat{C}}$ for the newly obtained clause $\widehat{\widehat{C}}^e$ and recursively employed to obtain the final e-form $\widehat{\widehat{P}}_i^{[\alpha,e]}$ of the given argument slice.

The non-deterministic nature of this definition implies that all possibilities will be tried until there are no more applicable e-form δ-relations. If there is any clause which matches one of these relations then it will be found and properly replaced by its e-form.

Since we are dealing with argument slices obtained from actual Prolog procedures, it is always the case that in every clause there must be relationships (explicit or not) between the variable in its head goal and each of the variables in its recursive calls. Implicit relationships between variables are repeated occurrences of the same symbol in different recursive subgoals. Explicit relationships involve non-recursive subgoals and are considered below. As far as the syntactic features of the most abstract argument slice are concerned, these relations make sure that there is at least one non-recursive subgoal in the body of each clause; for instance, if there is a clause- and mode-annotated clause in the argument slice with an empty body, that is, of the form

$$\boxed{\mathtt{p(}x\mathtt{)\ .}\ \theta_0\ \gamma_0}$$

then δ-relation B.8 will replace it for

$$\boxed{\begin{array}{rl} \mathtt{p(}x\mathtt{)\,:\!-} & \theta_0\ \gamma_0 \\ \gamma\ \theta_0\ \ \mathit{true}\mathtt{(}x\mathtt{)\ .} & \theta_0\ \gamma' \end{array}}$$

Clauses of the form

$$
\boxed{\begin{array}{rl}
 & \mathtt{p(}x\mathtt{)\,:\!-} \quad \theta_0\ \gamma_0 \\
 & \vec{S}_1 \\
\gamma\ \theta & \mathtt{p(}x\mathtt{)\,,}\ \theta'\ \gamma' \\
 & \vec{S}_2\,\mathtt{.}
\end{array}}
$$

are replaced by (Def.: B.1)

$$
\boxed{\begin{array}{rl}
 & \mathtt{p(}x\mathtt{)\,:\!-} \quad \theta_0^{\alpha}\ \gamma_0 \\
 & \vec{S}_1^{\alpha} \\
\gamma\ \theta^{\alpha} & x\ \mathtt{=}\ y\mathtt{\,,}\ \theta_*\ \gamma_* \\
\gamma\ \theta_* & \mathtt{p(}y\mathtt{)\,,}\ \ \theta'^{\alpha}\ \gamma' \\
 & \vec{S}_2^{\alpha}\,\mathtt{.}
\end{array}}
$$

or by (Def.: B.2)

$$
\boxed{\begin{array}{rl}
 & \mathtt{p(}x\mathtt{)\,:\!-} \quad \theta_0^{\alpha}\ \gamma_0 \\
 & \vec{S}_1^{\alpha} \\
\gamma\ \theta^{\alpha} & \mathtt{p(}y\mathtt{)\,,}\ \ \theta'^{\alpha}\ \gamma'^{\alpha} \\
\gamma\ \theta'^{\alpha} & x\ \mathtt{=}\ y\mathtt{\,,}\ \theta_*\ \gamma' \\
 & \vec{S}_2^{\alpha}\,\mathtt{.}
\end{array}}
$$

These last two cases assure us that at least one *relation* subgoal will always be found in a clause: those clauses where head goal variables are replicated in recursive goals are replaced by clauses where this replication is made via an explicit association using the operator *=/2*. This, together with the rest of the informal proof below, explains the fact that there must be at least one *relation* descriptor in the body of the clauses of the most abstract argument slice.

The e-form δ-relations also assure us that complex subgoals are replaced by equivalent sequences of simpler subgoals. This guarantees that each non-recursive subgoal has an appropriate procedural predicate descriptor different from *relation* being assigned to it by means of $\Re^p$, as explained below.

B.4.2 The $\Re^p$ Relation

The $\Re^p$ predicate relates, similarly, two argument slices $\hat{\hat{P}}_i$ and $\hat{\hat{P}}_i^{\alpha}$, such that the latter is the procedural abstraction of the former. The definition of $\Re^p$, introduced in Section 5.7.6, is

$$
\begin{array}{rcl}
\Re^p(\hat{\hat{P}}_i, \hat{\hat{P}}_i^{[\alpha,p]}) & \Longleftarrow & \Re^p_C(\hat{\hat{P}}_i, \hat{\hat{P}}'_i) \wedge \Re^p(\hat{\hat{P}}'_i, \hat{\hat{P}}_i^{[\alpha,p]}) \\
\Re^p(\hat{\hat{P}}_i, \hat{\hat{P}}_i^{[\alpha,p]}) & \Longleftarrow & \neg\Re^p_C(\hat{\hat{P}}_i, \hat{\hat{P}}'_i) \wedge \hat{\hat{P}}_i = \hat{\hat{P}}_i^{[\alpha,p]}
\end{array}
$$

The first clause recursively relates $\hat{\hat{P}}_i$, a clause- and mode-annotated argument slice in its e-form, to $\hat{\hat{P}}_i^{[\alpha,p]}$ by means of an intermediate construct $\hat{\hat{P}}'_i$ obtained via the auxiliary predicate $\Re^p_C$. $\Re^p_C$ holds if, and only if, $\hat{\hat{P}}'_i$ differs from $\hat{\hat{P}}_i$ in exactly one clause: this clause has a counterpart in $\hat{\hat{P}}_i$ but has one of its subgoals replaced by a descriptor different from *relation*, that is, one of of the descriptors bearing a procedural meaning. $\Re^p_C$ is defined below. The second clause states that an argument slice $\hat{\hat{P}}_i$ is $\Re^p$-related to another argument slice $\hat{\hat{P}}_i^{[\alpha,p]}$ if it is not possible to find an intermediate argument slice $\hat{\hat{P}}_i'$ that fulfils the $\Re^p_C$ relation; in this case, $\hat{\hat{P}}_i$ and $\hat{\hat{P}}_i^{[\alpha,p]}$ are bound to be the same. The definition of $\Re^p$ formalises, in a non-deterministic way, that $\Re^p_C$ will be applied to the clauses of $\hat{\hat{P}}_i$ exhaustively, until it is not applicable anymore.

The correctness and termination of $\Re^p$ relies thus on the definition of $\Re^p_C$. $\Re^p$ terminates if $\Re^p_C$ eventually fails. We must then prove that $\Re^p_C$ is always successful in mapping two clause- and mode-annotated argument slices $\hat{\hat{P}}_i$ and $\hat{\hat{P}}_i'$ such that they differ in exactly one clause; if it is not possible to find $\hat{\hat{P}}_i'$ then $\Re^p_C$ should fail. The difference between these argument slices is either a repeated clause that is found $\hat{\hat{P}}_i$ and has been removed in $\hat{\hat{P}}_i'$ or a clause in $\hat{\hat{P}}_i$ that has one of its subgoals abstracted as a predicate descriptor different from *relation*. We address the correctness and termination of $\Re^p_C$ below.

B.4.3 The $\Re^p_C$ Relation

The predicate $\Re^p$ employs the auxiliary predicate $\Re^p_C$, whose definition (also introduced in Section 5.7.6) is

$$\begin{array}{lcl}
\Re^p_C(\hat{\hat{P}}_i, \hat{\hat{P}}_i^\alpha) & \Leftarrow & \hat{\hat{P}}_i = \langle \vec{C}_1\, \hat{\hat{C}}\, \vec{C}_2\, \hat{\hat{C}}'\, \hat{\hat{C}}\, \vec{C}_3 \rangle \wedge \hat{\hat{P}}_i^\alpha = \langle \vec{C}_1\, \hat{\hat{C}}\, \vec{C}_2\, \hat{\hat{C}}'\, \vec{C}_3 \rangle \\
\Re^p_C(\hat{\hat{P}}_i, \hat{\hat{P}}_i^\alpha) & \Leftarrow & \hat{\hat{P}}_i = \langle \vec{C}_1\, \hat{\hat{C}}\, \vec{C}_2 \rangle \wedge \hat{\hat{P}}_i^\alpha = \langle \vec{C}_1\, \hat{\hat{C}}^\alpha\, \vec{C}_2 \rangle \wedge \Re^p_S(\hat{\hat{C}}, \hat{\hat{C}}^\alpha)
\end{array}$$

The first clause relates a given argument slice $\hat{\hat{P}}_i$ with $\hat{\hat{P}}_i^\alpha$ if the former has a pair of repeated clauses $\hat{\hat{C}}$: the second occurrence of this clause is removed from $\hat{\hat{P}}_i$ yielding $\hat{\hat{P}}_i^\alpha$. This explains the lack of repeated clauses in the procedural abstraction of an argument slice. The second clause picks up one clause $\hat{\hat{C}}$ from $\hat{\hat{P}}_i$ and finds its abstract form $\hat{\hat{C}}^\alpha$ via $\Re^p_S$; $\Re^p_S$ abstracts subgoals in the body of $\hat{\hat{C}}$ by relating via some δ-relation a clause- and mode-annotated subgoal of $\hat{\hat{C}}$ to a more abstract format of it.

The definition of $\Re^p_C$ ensures that if there is a pair of repeated clauses in $\hat{\hat{P}}_i$ then it will provide the component $\hat{\hat{P}}_i^\alpha$ consisting of $\hat{\hat{P}}_i$ without the second occurrence of the repeated clause; it also ensures that if there is one clause $\hat{\hat{C}}$ in $\hat{\hat{P}}_i$ that satisfies $\Re^p_S$ then this clause will be eventually used to obtain $\hat{\hat{P}}_i^\alpha$. $\Re^p_C$ fails if there are no repeated clauses in $\hat{\hat{P}}_i$ or if it cannot relate one of its clauses to a more abstract form via $\Re^p_S$.

B.4.4 The $\Re^p_S$ Relation and its Subset of δ-Relations

$\Re^p_S$ relates two clause- and mode-annotated clauses when one of their subgoals are δ-related:

$$\begin{array}{lcl}
\Re^p_S(\hat{\hat{C}}, \hat{\hat{C}}^\alpha) & \Leftarrow & \hat{\hat{C}} = H \;\text{:-}\; \theta\gamma\, \langle \vec{S}_1 \hat{\hat{S}} \vec{S}_2 \rangle \wedge \hat{\hat{C}}^\alpha = H \;\text{:-}\; \theta\gamma\, \langle \vec{S}_1 \hat{\hat{S}}^\alpha \vec{S}_2 \rangle \wedge \\
& & \hat{\hat{S}} \xleftrightarrow{\delta} \hat{\hat{S}}^\alpha
\end{array}$$

$\Re^p_S$ holds when $\widehat{\widehat{S}}$ in $\widehat{\widehat{C}}$ is δ-related to $\widehat{\widehat{S}}^\alpha$ in $\widehat{\widehat{C}}^\alpha$. The δ-relations mentioned in the definition are only those mapping actual clause- and mode-annotated Prolog subgoals to those descriptors different from *relation*, that is, δ-relations B.9 to B.21. $\Re^p_S$ fails when there are no subgoals $\widehat{\widehat{S}}$ in $\widehat{\widehat{C}}$ fulfilling one of the δ-relations B.9 to B.21; if there is one clause- and mode-annotated subgoal matching the left-hand side of one of the δ-relations B.9 to B.21 then the non-deterministic definition of $\Re^p_S$ guarantees it will be eventually considered.

The subset of δ-relations B.9 to B.21 must ensure that all possible clause- and mode-annotated Prolog subgoals are addressed. The definitions of *e-form*, $\Re^p$, $\Re^p_C$ and $\Re^p_S$ guarantee that every clause- and mode-annotated Prolog subgoal comprising the bodies of the clauses in the given argument slice $\widehat{\widehat{P}}_i$ is eventually considered and abstracted using a δ-relation:

- Subgoals employing system operators $\diamond$ (Section 1.5) different from $=/2$ and $=../2$ are abstracted by δ-relation B.9, if only constants are employed, or by B.10, if variables are also being used;
- Subgoals employing system predicate ¶ (Section 1.5) are abstracted by B.13;
- Subgoals of the form $x \diamond a, \diamond \in \{\texttt{=}, \texttt{=..}\}$ are abstracted by B.11 or by B.14, depending on whether or not x changes its mode;
- Subgoals of the form $x_1 \diamond x_2, \diamond \in \{\texttt{=}, \texttt{=..}\}$ are abstracted by B.12 or by B.15, depending on whether or not x_i changes its mode; if x_i remains associated with **f** throughout the subgoal execution then the subgoal is abstracted by B.21;
- Subgoals employing system predicates $\Im$ (Section 1.5) are abstracted by B.16: since the argument slices are in their e-form, it is guaranteed that $\Im$ will only appear with its variable satisfying the *change* relation;
- Subgoals of the form $x \texttt{=} f(\vec{V})$" are abstracted by B.17 or by B.18, depending on whether x changes or not its mode: since the argument slice is in its e-form before the abstraction takes place, those subgoals performing complex bindings between subterms of x and variables in $f(\vec{V})$ are replaced for a sequence of simpler subgoals all of which match one of the δ-relations described;
- Subgoals of the form $x \texttt{ is } f(\vec{V})$ are abstracted by B.19: the initial processing of argument slices into their e-form guarantees that subgoals of this sort will always instantiate x;
- Subgoals employing user-defined predicates are abstracted by B.20.

The cases above address all the clause- and mode-annotated Prolog subgoals found in our studied class of programs as defined in Section 1.5. They guarantee that there is a unique δ-relation for each combination of Prolog subgoal and mode-annotations.

B.5 Convergence to the Most Abstract Argument Slice

Similarly, if $\Re^*$ is used to automatically abstract a procedural abstraction $\widehat{\widehat{P}}_i^{[\alpha,p]}$ of a clause- and mode-annotated argument slice then its most abstract version $\widehat{\widehat{P}}_i^{[\alpha,*]}$ is eventually reached and it is such that $\Re^*$ will not abstract it any further. $\widehat{\widehat{P}}_i^{[\alpha,*]}$ has no duplicate clauses and each of its clauses is such that:

1. its body consists only of clause- and mode-annotated recursive calls and *relation* subgoals;
2. there are no two consecutive *relation* subgoals;
3. there is at least one *relation* subgoal;

That is, its clauses are of the form $\mathcal{P}(x)$:- $\theta' \gamma'$ $\vec{S}$, $\vec{S}$ being a non-empty vector of clause- and mode-annotated subgoals $\gamma\,\theta$ $\mathcal{P}(y)$ $\theta'\,\gamma'$ or $\gamma\,\theta$ $relation(\vec{V})$ $\theta'\,\gamma'$, with at least one subgoal of the latter kind and no two consecutive occurrence of such *relation* subgoals. If there are no recursive subgoals $\mathcal{P}(y)$ then $\vec{S}$ consists of a single $relation(\vec{V})$ subgoal.

The proof of the convergence of $\Re^*$ is similar to that of $\Re^p$. $\Re^*$ and $\Re^p$ differ in the kind of clause- and mode-annotated argument slice supplied for abstraction: in the former, a procedural abstraction must be provided; in the latter, an argument slice in its e-form is the object of analysis.

$\Re^*$ is defined in Section 5.7.6 in a similar fashion as $\Re^p$, with an equivalent predicate $\Re^*_C$ replacing $\Re^p_C$ and an equivalent $\Re^*_S$ replacing $\Re^p_S$. Its fundamental distinction is the subset of available δ-relations: it consists of relations B.22 to B.29, mapping descriptors to the more abstract *relation* descriptor.

The δ-relations B.22 to B.29 relate each of the predicate descriptors obtained in $\Re^p$ to a more abstract *relation* descriptor.

Given that every clause- and mode-annotated descriptor subgoal has a δ-relation that abstracts it as a *relation* descriptor, consecutive pairs of *relation* subgoals are appropriately merged as a single *relation* subgoal by relation B.30, and since the relations $\Re^*_C$ and $\Re^*_S$ ensure us that every subgoal of each clause is eventually considered, then our informal proof is completed. Our rationale then is:

- every combination of subgoals and modes of clause- and mode-annotated argument slices in their e-form is eventually abstracted as a predicate descriptor;
- every predicate descriptor eventually gets abstracted as a *relation*;
- every pair of consecutive *relation* predicate descriptors is eventually abstracted as a single *relation* predicate.

B.6 Examples of Abstractions

In this section we show examples of extracted argument slices being abstracted, according to our framework employing the repertoire of δ-relations given previously.

B.6.1 Abstraction of Argument Slices of *count_sum*/4

Given the following definition of procedure *count_sum*/4

```
count_sum(A,Y,Count,Sum):-
   A = [X|Xs],
   X = Y,
   Count = 0,
   Sum = 0.
count_sum(A,Y,Count,Sum):-
   A = [X|Xs],
   X \== Y,
   count_sum(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, if analysed with respect to query `count_sum([1,2,3,end,4],end,C,S)` is such that its first argument slice yields the following clause-annotated abstract versions (for the sake of brevity the clause annotations with empty sets of variables are omitted):

$\widehat{P}_1^{[\alpha,*]}$

	$\mathcal{P}$(A):-	$\langle\!\langle \mathit{offer}(\{\texttt{A}\})\rangle\!\rangle$
$\langle\!\langle \mathit{required}(\{\texttt{Y}\})\rangle\!\rangle$	$\mathit{relation}(\{\texttt{A},\texttt{X},\texttt{Xs},\texttt{Y}\})$.	$\langle\!\langle \mathit{offer}(\{\texttt{X},\texttt{Xs}\})\rangle\!\rangle$
	$\mathcal{P}$(A):-	$\langle\!\langle \mathit{offer}(\{\texttt{A}\})\rangle\!\rangle$
$\langle\!\langle \mathit{required}(\{\texttt{Y}\})\rangle\!\rangle$	$\mathit{relation}(\{\texttt{A},\texttt{X},\texttt{Xs},\texttt{Y}\})$.	$\langle\!\langle \mathit{offer}(\{\texttt{X},\texttt{Xs}\})\rangle\!\rangle$
	$\mathcal{P}$(Xs).	

$\widehat{P}_1^{[\alpha,p]}$

	$\mathcal{P}$(A):-	$\langle\!\langle \mathit{offer}(\{\texttt{A}\})\rangle\!\rangle$
	$\mathit{decomp}(\texttt{A},\{\texttt{X},\texttt{Xs}\})$,	$\langle\!\langle \mathit{offer}(\{\texttt{X},\texttt{Xs}\})\rangle\!\rangle$
$\langle\!\langle \mathit{required}(\{\texttt{Y}\})\rangle\!\rangle$	$\mathit{test}(\{\texttt{X},\texttt{Y}\})$.	
	$\mathcal{P}$(A):-	$\langle\!\langle \mathit{offer}(\{\texttt{A}\})\rangle\!\rangle$
	$\mathit{decomp}(\texttt{A},\{\texttt{X},\texttt{Xs}\})$,	$\langle\!\langle \mathit{offer}(\{\texttt{X},\texttt{Xs}\})\rangle\!\rangle$
$\langle\!\langle \mathit{required}(\{\texttt{Y}\})\rangle\!\rangle$	$\mathit{test}(\{\texttt{X},\texttt{Y}\})$,	
	$\mathcal{P}$(Xs).	

$\widehat{P}_1^{[\alpha,0]}$

	$\mathcal{P}$(A):-	$\langle\!\langle \mathit{offer}(\{\texttt{A}\})\rangle\!\rangle$
	A = [X|Xs],	$\langle\!\langle \mathit{offer}(\{\texttt{X},\texttt{Xs}\})\rangle\!\rangle$
$\langle\!\langle \mathit{required}(\{\texttt{Y}\})\rangle\!\rangle$	X = Y.	
	$\mathcal{P}$(A):-	$\langle\!\langle \mathit{offer}(\{\texttt{A}\})\rangle\!\rangle$
	A = [X|Xs],	$\langle\!\langle \mathit{offer}(\{\texttt{X},\texttt{Xs}\})\rangle\!\rangle$
$\langle\!\langle \mathit{required}(\{\texttt{Y}\})\rangle\!\rangle$	X \== Y,	
	$\mathcal{P}$(Xs).	

The clause-annotated abstractions of the second argument slice are

$\widehat{P}_2^{[\alpha,*]}$

```
P(Y):-                  ⟨⟨offer({Y})⟩⟩
  relation(Y).
P(Y):-                  ⟨⟨offer({Y})⟩⟩
  relation({Y,Z}),      ⟨⟨offer({Z})⟩⟩
  P(Z).
```

$\widehat{P}_2^{[\alpha,p]}$

```
P(Y):-                  ⟨⟨offer({Y})⟩⟩
  test(Y).
P(Y):-                  ⟨⟨offer({Y})⟩⟩
  assign({Y},{Z}),      ⟨⟨offer({Z})⟩⟩
  P(Z).
```

$\widehat{P}_2^{[\alpha,e]}$

```
P(Y):-        ⟨⟨offer({Y})⟩⟩
  true(Y).
P(Y):-        ⟨⟨offer({Y})⟩⟩
  Z = Y,      ⟨⟨offer({Z})⟩⟩
  P(Z).
```

$\widehat{P}_2^{[\alpha,0]}$

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

The clause-annotated abstractions of the third argument slice are

$\widehat{P}_3^{[\alpha,*]}$

```
P(Count):-
  relation({Count}).                  ⟨⟨offer({Count})⟩⟩
P(Count):-
  P(RestCount),                       ⟨⟨offer({RestCount})⟩⟩
  relation({Count,RestCount}).        ⟨⟨offer({Count})⟩⟩
```

$\widehat{P}_3^{[\alpha,p]}$

```
P(Count):-
  assign(∅,{Count}).                  ⟨⟨offer({Count})⟩⟩
P(Count):-
  P(RestCount),                       ⟨⟨offer({RestCount})⟩⟩
  calc({RestCount},Count).            ⟨⟨offer({Count})⟩⟩
```

$\widehat{P}_3^{[\alpha,0]}$

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

The clause-annotated abstractions of the fourth argument slice are

$\widehat{P}_4^{[\alpha,*]}$

	$\mathcal{P}$(Sum):-	
	relation({Sum}).	$\langle\!\langle$ *offer*({Sum}) $\rangle\!\rangle$
	$\mathcal{P}$(Sum):-	
	$\mathcal{P}$(RestSum),	$\langle\!\langle$ *offer*({RestSum}) $\rangle\!\rangle$
$\langle\!\langle$ *required*({X}) $\rangle\!\rangle$	*relation*({RestSum,X,Sum}).	$\langle\!\langle$ *offer*({Sum}) $\rangle\!\rangle$

$\widehat{P}_4^{[\alpha,p]}$

	$\mathcal{P}$(Sum):-	
	assign(∅,{Sum}).	$\langle\!\langle$ *offer*({Sum}) $\rangle\!\rangle$
	$\mathcal{P}$(Sum):-	
	$\mathcal{P}$(RestSum),	$\langle\!\langle$ *offer*({RestSum}) $\rangle\!\rangle$
$\langle\!\langle$ *required*({X}) $\rangle\!\rangle$	*calc*({RestSum,X},Sum).	$\langle\!\langle$ *offer*({Sum}) $\rangle\!\rangle$

$\widehat{P}_4^{[\alpha,0]}$

	$\mathcal{P}$(Sum):-	
	Sum = 0.	$\langle\!\langle$ *offer*({Sum}) $\rangle\!\rangle$
	$\mathcal{P}$(Sum):-	
	$\mathcal{P}$(RestSum),	$\langle\!\langle$ *offer*({RestSum}) $\rangle\!\rangle$
$\langle\!\langle$ *required*({X}) $\rangle\!\rangle$	Sum is RestSum + X.	$\langle\!\langle$ *offer*({Sum}) $\rangle\!\rangle$

B.6.2 Abstraction of Argument Slices of *dutch*/4

Let there be the following definition of predicate *dutch*/4, taken from [SS86]:

```
dutch(C,R,W,B):-
   C = [],
   R = [],
   W = [],
   B = [].
dutch(C,R,W,B):-
   C = [X|Cs],
   X = red(_),
   R = [X|Rs],
   dutch(Cs,Rs,W,B).
dutch(C,R,W,B):-
   C = [X|Cs],
   X = white(_),
   W = [X|Ws],
   dutch(Cs,R,Ws,B).
dutch(C,R,W,B):-
   C = [X|Cs],
   X = blue(_),
   B = [X|Bs],
   dutch(Cs,R,W,Bs).
```

This predicate is a possible solution to the Dutch flag problem, collecting the different occurrences of the colours red, white and blue, maintaining their relative ordering. If it is analysed with respect to the query

```
dutch([blue(1),red(1),white(2),red(2),blue(2),white(1)],R,W,B).
```

we obtain the following mode-annotated program:

```
                                  dutch(C,R,W,B):-      {C/g,R/f,W/f,B/f}
           {C/g,R/f,W/f,B/f}        C = [],             {C/g,R/f,W/f,B/f}
           {C/g,R/f,W/f,B/f}        R = [],             {C/g,R/g,W/f,B/f}
           {C/g,R/g,W/f,B/f}        W = [],             {C/g,R/g,W/g,B/f}
           {C/g,R/g,W/g,B/f}        B = [].             {C/g,R/g,W/g,B/g}
                                  dutch(C,R,W,B):-      {C/g,R/f,W/f,B/f,X/f,Cs/f,Rs/f}
 {C/g,R/f,W/f,B/f,X/f,Cs/f,Rs/f}    C = [X|Cs],         {C/g,R/f,W/f,B/f,X/g,Cs/g,Rs/f}
 {C/g,R/f,W/f,B/f,X/g,Cs/g,Rs/f}    X = red(_),         {C/g,R/f,W/f,B/f,X/g,Cs/g,Rs/f}
 {C/g,R/f,W/f,B/f,X/g,Cs/g,Rs/f}    R = [X|Rs],         {C/g,R/i,W/f,B/f,X/g,Cs/g,Rs/f}
 {C/g,R/i,W/f,B/f,X/g,Cs/g,Rs/f}    dutch(Cs,Rs,W,B).   {C/g,R/g,W/g,B/g,X/g,Cs/g,Rs/g}
                                  dutch(C,R,W,B):-      {C/g,R/f,W/f,B/f,X/f,Cs/f,Ws/f}
 {C/g,R/f,W/f,B/f,X/f,Cs/f,Ws/f}    C = [X|Cs],         {C/g,R/f,W/f,B/f,X/g,Cs/g,Ws/f}
 {C/g,R/f,W/f,B/f,X/g,Cs/g,Ws/f}    X = white(_),       {C/g,R/f,W/f,B/f,X/g,Cs/g,Ws/f}
 {C/g,R/f,W/f,B/f,X/g,Cs/g,Ws/f}    W = [X|Ws],         {C/g,R/f,W/i,B/f,X/g,Cs/g,Ws/f}
 {C/g,R/f,W/f,B/f,X/g,Cs/g,Ws/f}    dutch(Cs,R,Ws,B).   {C/g,R/g,W/g,B/g,X/g,Cs/g,Bs/g}
                                  dutch(C,R,W,B):-      {C/g,R/f,W/f,B/f,X/f,Cs/f,Bs/f}
 {C/g,R/f,W/f,B/f,X/f,Cs/f,Bs/f}    C = [X|Cs],         {C/g,R/f,W/f,B/f,X/g,Cs/g,Bs/f}
 {C/g,R/f,W/f,B/f,X/g,Cs/g,Bs/f}    X = blue(_),        {C/g,R/f,W/f,B/f,X/g,Cs/g,Bs/f}
 {C/g,R/f,W/f,B/f,X/g,Cs/g,Bs/f}    B = [X|Bs],         {C/g,R/f,W/f,B/i,X/g,Cs/g,Bs/f}
 {C/g,R/f,W/f,B/i,X/g,Cs/g,Bs/f}    dutch(Cs,R,W,Bs).   {C/g,R/g,W/g,B/g,X/g,Cs/g,Bs/g}
```

Its first clause- and mode-annotated argument slice $\widehat{\widetilde{P}}_1$ is of the form

```
                 dutch(C):-      {C/g}              ⟨⟨ offer({C}) ⟩⟩
       {C/g}       C = [].       {C/g}
                 dutch(C):-      {C/g,X/f,Cs/f}     ⟨⟨ offer({C}) ⟩⟩
{C/g,X/f,Cs/f}     C = [X|Cs],   {C/g,X/g,Cs/g}     ⟨⟨ offer({X,Cs}) ⟩⟩
{C/g,X/g,Cs/g}     X = red(_),   {C/g,X/g,Cs/g}
{C/g,X/g,Cs/g}     dutch(Cs).    {C/g,X/g,Cs/g}
                 dutch(C):-      {C/g,X/f,Cs/f}     ⟨⟨ offer({C}) ⟩⟩
{C/g,X/f,Cs/f}     C = [X|Cs],   {C/g,X/g,Cs/g}     ⟨⟨ offer({X,Cs}) ⟩⟩
{C/g,X/g,Cs/g}     X = white(_), {C/g,X/g,Cs/g}
{C/g,X/g,Cs/g}     dutch(Cs).    {C/g,X/g,Cs/g}
                 dutch(C):-      {C/g,X/f,Cs/f}     ⟨⟨ offer({C}) ⟩⟩
{C/g,X/f,Cs/f}     C = [X|Cs],   {C/g,X/g,Cs/g}     ⟨⟨ offer({X,Cs}) ⟩⟩
{C/g,X/g,Cs/g}     X = blue(_),  {C/g,X/g,Cs/g}
{C/g,X/g,Cs/g}     dutch(Cs).    {C/g,X/g,Cs/g}
```

It yields the following abstractions:

$\widehat{P}_4^{[\alpha,*]}$		$\mathcal{P}$(C):-	{C/g}	⟪ *offer*({C}) ⟫
	{C/g}	*relation*({C}).	{C/g}	
		$\mathcal{P}$(C):-	{C/g,X/f,Cs/f}	⟪ *offer*({C}) ⟫
	{C/g,X/f,Cs/f}	*relation*({C,X,Cs}),	{C/g,X/g,Cs/g}	⟪ *offer*({X,Cs}) ⟫
	{C/g,X/g,Cs/g}	$\mathcal{P}$(Cs).	{C/g,X/g,Cs/g}	

$\widehat{P}_4^{[\alpha,p]}$		$\mathcal{P}$(C):-	{C/g}	⟪ *offer*({C}) ⟫
	{C/g}	*test*({C}).	{C/g}	
		$\mathcal{P}$(C):-	{C/g,X/f,Cs/f}	⟪ *offer*({C}) ⟫
	{C/g,X/f,Cs/f}	*decomp*(C,{X,Cs},	{C/g,X/g,Cs/g}	⟪ *offer*({X,Cs}) ⟫
	{C/g,X/g,Cs/g}	*test*({X}),	{C/g,X/g,Cs/g}	
	{C/g,X/g,Cs/g}	$\mathcal{P}$(Cs).	{C/g,X/g,Cs/g}	

$\widehat{P}_4^{[\alpha,0]}$		$\mathcal{P}$(C):-	{C/g}	⟪ *offer*({C}) ⟫
	{C/g}	C = [].	{C/g}	
		$\mathcal{P}$(C):-	{C/g,X/f,Cs/f}	⟪ *offer*({C}) ⟫
	{C/g,X/f,Cs/f}	C = [X|Cs],	{C/g,X/g,Cs/g}	⟪ *offer*({X,Cs}) ⟫
	{C/g,X/g,Cs/g}	X = red(_),	{C/g,X/g,Cs/g}	
	{C/g,X/g,Cs/g}	$\mathcal{P}$(Cs).	{C/g,X/g,Cs/g}	
		$\mathcal{P}$(C):-	{C/g,X/f,Cs/f}	⟪ *offer*({C}) ⟫
	{C/g,X/f,Cs/f}	C = [X|Cs],	{C/g,X/g,Cs/g}	⟪ *offer*({X,Cs}) ⟫
	{C/g,X/g,Cs/g}	X = white(_),	{C/g,X/g,Cs/g}	
	{C/g,X/g,Cs/g}	$\mathcal{P}$(Cs).	{C/g,X/g,Cs/g}	
		$\mathcal{P}$(C):-	{C/g,X/f,Cs/f}	⟪ *offer*({C}) ⟫
	{C/g,X/f,Cs/f}	C = [X|Cs],	{C/g,X/g,Cs/g}	⟪ *offer*({X,Cs}) ⟫
	{C/g,X/g,Cs/g}	X = blue(_),	{C/g,X/g,Cs/g}	
	{C/g,X/g,Cs/g}	$\mathcal{P}$(Cs).	{C/g,X/g,Cs/g}	

Appendix C

Examples of Designed Programming Techniques

In this appendix we show some examples of programming techniques being designed by means of the service offered in our knowledge management tool. We show examples of argument slices being devised and then used to define more complex constructs via the binding of the variables in their clause-annotations.

Our tool supports the design of argument slices, the building blocks of programming techniques. They can be devised starting from a very abstract template and have it gradually defined, or by reusing existing abstract components (that is, argument slices with predicate descriptors) and refining them.

Alternatively, programming techniques can be defined using existing extracted argument slices chosen from the library and linked in some different manner to other argument slices.

A repertoire of commands to manipulate argument slices is made available. These commands can either be applied to an argument slice template or to an abstract version (*i.e.* with at least one predicate descriptor in one of its clauses) of an existing argument slice in the library. The δ-relations comprise our design commands, with an interactive element to narrow down the possibly infinite options during the specialisation of abstract constructs.

C.1 Designing a Programming Technique to Decompose Data Structures

Our implemented tool supports the design of programming techniques by offering its users a repertoire of commands to define argument slices, the building blocks of programming techniques, and to link them together by means of variable sharing.

In this example, let us assume that the user chose to define an argument slice from "scratch", using the most abstract clause template. Let us also assume that the user wants to define a programming technique traversing a list data structure, testing its elements until the empty list is found, that is, skeleton `traverse_n`, proposed by Kirschenbaum and colleagues [SS86, KLS89, Lak89, SK93] and shown in Section 2.3.

The screen dump of Figure C.1 shows two windows: the bigger one, named "`Design History`", shows the design tree currently available; the smaller one, named "`Command`", provides a simple form of interface, allowing its users to type in the identifier of components of the tree and commands. The user interface checks the correctness of the

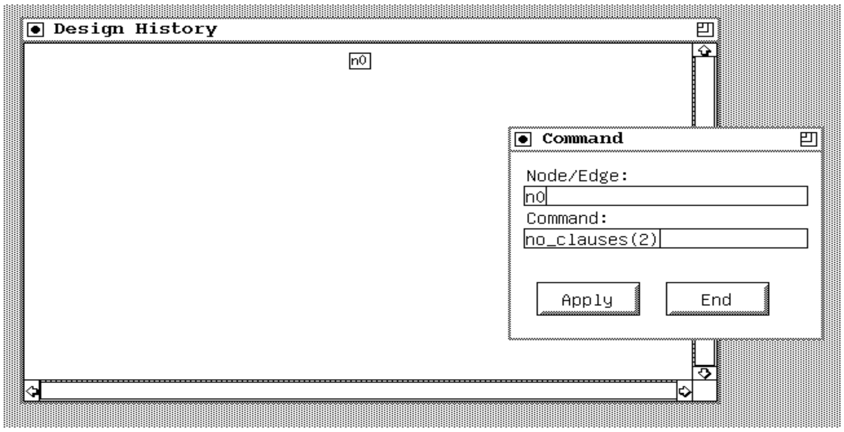


Figure C.1: Initial Window of Design Tool showing the Initial Template `n0`

syntax of the typed commands and if their application to the specified component is appropriate. The user can inspect the contents of any of the nodes or edges of the tree at any point, by specifying the component and typing the command "`show`", then pressing the button "`Apply`".

In the screen dump above the design tree consisting of a single root note `n0`. The content of node `n0` is a sequence of clauses $\langle C_1, \ldots, C_n \rangle$ where each C_i is a most abstract clause template of the form

$$\boxed{\begin{array}{l} \mathcal{P}\texttt{(A):-} \\ \quad \mathit{relation}\texttt{(}V_0\texttt{),} \\ \quad \mathcal{P}\texttt{(A1),} \\ \quad \mathit{relation}\texttt{(}V_1\texttt{),} \\ \qquad \vdots \\ \quad \mathcal{P}\texttt{(An),} \\ \quad \mathit{relation}\texttt{(}V_n\texttt{).} \end{array}}$$

The user must initially define the number of clauses the initial template is to have. This can be done by specifying node `n0` in the Command Window, typing the command `no_clauses(`n`)`, where n is an integer greater than zero, and clicking on the Apply button. In Figure C.1 above we see the design tree in the moment before the command `no_clause(2)` is applied. Its outcome is shown in Figure C.2: the edge `e1` linking nodes `n0` and `n1` stores the command used to transform the former into the latter. The content of node `n1` is roughly the same as `n0`, with the exception that in the latter the actual number of clauses is known. At this point the user can specify the number of recursive calls of each clause, customising the most abstract clause template for each

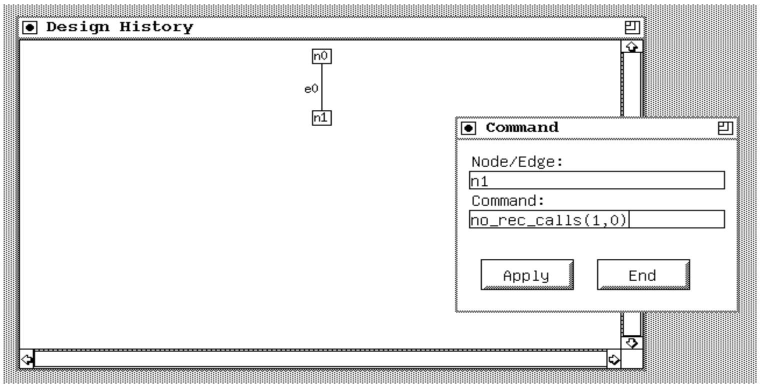


Figure C.2: Design Tree after application of Command `no_clauses(2)`

clause of `n1`. Given our initial assumption that the user wishes to define a technique to traverse a singly-recursive data structure testing each of its elements until its base case is reached, the user should make a distinction between the base case clause, without a recursive call, and the recursive clause, with a single recursive call. In Figure C.2 we see in the Command Window the specification of component `n1` and the command `no_rec_calls(1,0)`, indicating that the number of recursive calls of the first clause is 0. Following the application of this command, the user supplies the new node `n2` and the command `no_rec_calls(2,1)`, indicating that the number of recursive calls of the second clause of `n2` is to be defined as 1. The outcome of these operations is depicted in Figure C.3 below: the Design History Window shows the four current nodes

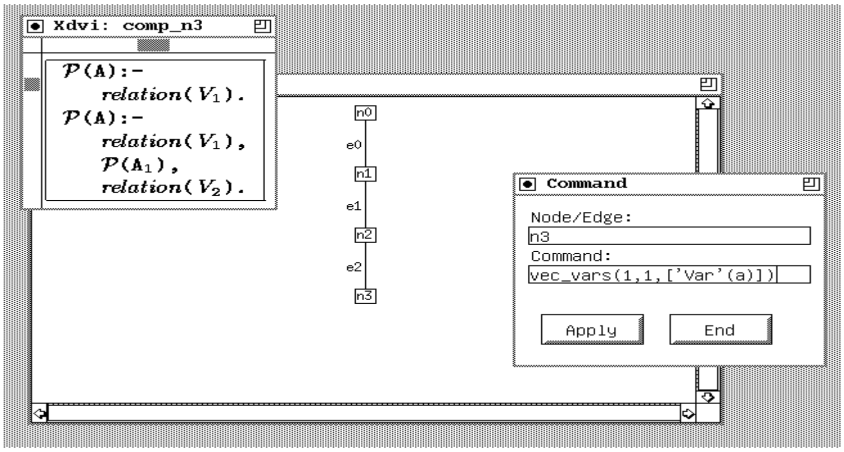


Figure C.3: Design Tree after specification of Recursive Calls

and the commands employed to define them. In that figure we can also see a small window "`Xdvi: comp_n3`", showing the content of `n3`, displayed after the application of command `show` to node `n3`.

Node `n3` can now have its vectors of variables in the *relation* predicate descriptors defined. The user should have in mind the purpose of each clause: their vectors of variables must consist of all those variables that are likely to appear. It is not possible, after this point in the design process, to create new variables symbols within the argument slice. In our particular example, the data structure the user aims at is a list such that its base case is the empty list and its recursive case is the decomposition of the list into its head and tail. The commands **vec_vars(**c**,**s**,**n**)** where c is the clause number, s is the subgoal number and n is the number of fresh variables to be created, or **vec_vars(**c**,**s**,**$\{v_1, \ldots, v_n\}$**)** where c and s are as above and $\{v_1, \ldots, v_n\}$ are specific variable symbols supplied by the user, provide means to define such vectors of variables. In our example, the user has applied the command vec_vars(1,1,{A}) to n3 obtaining n4, and applied vec_vars(1,1,{A,B,C}) to n4, yielding n5, of the form

```
P(A):-
    relation({A}).
P(A):-
    relation({A,B,C}),
    P(C).
```

So far there has been no mention to clause or mode-annotations. It is only at this point, after the vectors of variables are defined that these issues are tackled. The annotations cannot be obtained fully automatically, since there are many possibilities depending on the intended use of the constructs. However, when the user specifies this intended use, that is, if the argument slice is an *input* or an *output* argument slice (Section 5.5.2) the clause- and mode-annotations can be automatically inferred. Our technique currently being devised aims at traversing a data structure, which means that the component is to be used with its head argument instantiated. The user can then inform this by means of the intended_use(input) command, which if applied to node n5 above yields the following node n6:

```
                 P(A):-                 {A/i}
         {A/i}       relation({A}).     {A/i}           ⟨⟨ offer({A}) ⟩⟩
                 P(A):-                 {A/i,B/f,C/f}
 {A/i,B/f,C/f}       relation({A,B,C}), {A/i,B/?,C/?}   ⟨⟨ offer({A,B,C}) ⟩⟩
 {A/i,B/?,C/?}       P(C).              {A/i,B/?,C/i}
```

The annotations automatically obtained are not precise, but they are correct with respect to the intended use of the argument slice. If, alternatively, the user had chosen that the intended use of the argument slice was as an output contribution to a technique (command intended_use(output)) then the following annotations would be obtained

```
                 P(A):-                 {A/f}
         {A/f}       relation({A}).     {A/?}           ⟨⟨ offer({A}) ⟩⟩
                 P(A):-                 {A/f,B/f,C/f}
 {A/f,B/f,C/f}       relation({A,B,C}), {A/?,B/?,C/?}   ⟨⟨ offer({A,B,C}) ⟩⟩
 {A/?,B/?,C/?}       P(C).              {A/?,B/?,C/?}
```

The user is given the opportunity to provide more precise mode-annotations to refine those automatically obtained. The new instantiation mode T' associated with variable x has to be a specialised form of the existing token T, that is, T must subsume T', $T \sqsubseteq T'$ (Def. 3.6.1). In our specific example the user can change the tokens "**i**" associated with the head variable A to the more specific token "**g**". The mode-annotations of variables B and C after the *relation* subgoal can also be changed to the more specific "**g**" token associated with the subcomponents of the data structure after the decomposition (*relation*, still to be specified) takes place. The following clause- and mode-annotated argument slice can be obtained:

	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$ *offer*({A}) $\rangle\!\rangle$
{A/g}	*relation*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	$\langle\!\langle$ *offer*({A}) $\rangle\!\rangle$
{A/g,B/f,C/f}	*relation*({A,B,C}),	{A/g,B/g,C/g}	$\langle\!\langle$ *offer*({B,C}) $\rangle\!\rangle$
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	

The user can now proceed to the gradual definition of the predicate descriptors in the argument slice. The design tool provides support to this task by showing, upon request, those δ-relations applicable to specific subgoals. If the user specifies the component above and types in "`show_inst(1,1)`" (this command stands for "show the instances of subgoal 1 of clause 1), then the following is displayed:

The δ-relations applicable to goal 1,1 are:

$$\boxed{\gamma\,\theta\ \mathit{test}(W)\ \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\ \mathit{relation}(W)\ \theta'\,\gamma'}$$

$$\boxed{\gamma\,\theta\ \mathit{user\text{-}pred}(W)\ \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\ \mathit{relation}(W)\ \theta'\,\gamma'}$$

$$\boxed{\gamma\,\theta\ \mathit{true}(x)\ \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\ \mathit{relation}(\{x\})\ \theta'\,\gamma'}$$

$$\boxed{\begin{array}{c}\gamma_1\ \theta\ \mathit{relation}(W_1)\ \theta_*\ \gamma_1' \\ \gamma_2\ \theta_*\mathit{relation}(W_2)\ \theta'\ \gamma_2'\end{array}} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ \mathit{relation}(W)\ \theta'\ \gamma'}$$

Given that our argument slice is aimed at traversing a list until the empty list is found, the base-case clause should address the end of the traversal, that is, it should test if the end of the list has been reached. The user should then employ the first rule to specialise *relation* as *test*, thus obtaining

	$\mathcal{P}$(A):-	{A/g}	$\langle\!\langle$ *offer*({A}) $\rangle\!\rangle$
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	$\langle\!\langle$ *offer*({A}) $\rangle\!\rangle$
{A/g,B/f,C/f}	*relation*({A,B,C}),	{A/g,B/g,C/g}	$\langle\!\langle$ *offer*({B,C}) $\rangle\!\rangle$
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	

Similarly, upon typing "`show_inst(2,1)`", the following would be shown

The δ-relations applicable to goal 2,1 are:

$$\boxed{\gamma\,\theta\ \mathit{decomp}(x,V)\ \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\ \mathit{relation}(W)\ \theta'\,\gamma'}$$

$$\boxed{\gamma\,\theta\ \mathit{assign}(V_1,V_2)\ \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\ \mathit{relation}(W)\ \theta'\,\gamma'}$$

$$\boxed{\gamma\,\theta\ \mathit{user\text{-}pred}(W)\ \theta'\,\gamma'} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\,\theta\ \mathit{relation}(W)\ \theta'\,\gamma'}$$

$$\boxed{\begin{array}{c}\gamma_1\ \theta\ \mathit{relation}(W_1)\ \theta_*\ \gamma_1' \\ \gamma_2\ \theta_*\mathit{relation}(W_2)\ \theta'\ \gamma_2'\end{array}} \stackrel{\delta}{\longleftrightarrow} \boxed{\gamma\ \theta\ \mathit{relation}(W)\ \theta'\ \gamma'}$$

The `B` variable symbol is the non-recursive part (head) of the list being decomposed; `C` is the recursive part (body) of the list. The traversal of a list should allow tests to be performed on `B` as the decomposition is performed: this means that another *relation* subgoal using `B` must follow the current *relation* subgoal. This change can be performed by means of the fourth rule above, the details being specified by the user.

The component below is thus obtained:

	$\mathcal{P}$(A):-	{A/g}	⟪ *offer*({A}) ⟫
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	⟪ *offer*({A}) ⟫
{A/g,B/f,C/f}	*relation*({A,B,C}),	{A/g,B/g,C/g}	⟪ *offer*({B,C}) ⟫
{A/g,B/f,C/f}	*relation*({B}),	{A/g,B/g,C/g}	
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	

The first *relation* subgoal of the second clause is a data structure decomposition (descriptor *decomp*) and the second *relation* is a *test*. The appropriate specialisation of these constructs would thus yield:

	$\mathcal{P}$(A):-	{A/g}	⟪ *offer*({A}) ⟫
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	⟪ *offer*({A}) ⟫
{A/g,B/f,C/f}	*decomp*(A,{B,C}),	{A/g,B/g,C/g}	⟪ *offer*({B,C}) ⟫
{A/g,B/f,C/f}	*test*({B}),	{A/g,B/g,C/g}	
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	

that is, a procedural abstraction of a programming technique performing a traversal of a singly-recursive data structure the details of which are still to be specified. In our particular example, when the specific details of a list data structure are provided and incorporated to the component above, it yields

	$\mathcal{P}$(A):-	{A/g}	⟪ *offer*({A}) ⟫
{A/g}	A = [].	{A/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	⟪ *offer*({A}) ⟫
{A/g,B/f,C/f}	A = [B|C],	{A/g,B/g,C/g}	⟪ *offer*({B,C}) ⟫
{A/g,B/f,C/f}	*test*({B}),	{A/g,B/g,C/g}	
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	

At this point we have a robust formalisation for the `traverse_n` skeleton: the *test* predicate still to be specialised provides an economic way to represent the many possible tests, including the absence of any tests (*test* is replaced by *true*). If there is a need for a traversal technique that addresses the integer and non-integer components of a list as the decomposition takes place, the user could copy the second clause and specialise the *test* predicates differently, thus obtaining

	$\mathcal{P}$(A):-	{A/g}	⟪ *offer*({A}) ⟫
{A/g}	A = [].	{A/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	⟪ *offer*({A}) ⟫
{A/g,B/f,C/f}	A = [B|C],	{A/g,B/g,C/g}	⟪ *offer*({B,C}) ⟫
{A/g,B/f,C/f}	integer(B),	{A/g,B/g,C/g}	
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	⟪ *offer*({A}) ⟫
{A/g,B/f,C/f}	A = [B|C],	{A/g,B/g,C/g}	⟪ *offer*({B,C}) ⟫
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	

The *test* of the second clause was specialised as *true*. The procedural abstraction can be used to define techniques to perform a traversal of other singly-recursive data structures different from lists. For instance, the following argument slice

	$\mathcal{P}$(A):-	{A/g}	⟪ *offer*({A}) ⟫
{A/g}	A = foo1.	{A/g}	
	$\mathcal{P}$(A):-	{A/g}	⟪ *offer*({A}) ⟫
{A/g}	A = foo2.	{A/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	⟪ *offer*({A}) ⟫
{A/g,B/f,C/f}	A = baz(C,B),	{A/g,B/g,C/g}	⟪ *offer*({B,C}) ⟫
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	

Performs the traversal of a data structure `baz(`y`,`x`)` where y is the recursive part of it. Its base cases are terms of the form `foo1` and `foo2`.

Each of the argument slices produced above are programming techniques on their own, since they do not have *required* variables to be linked to *offer* variables of other argument slices (cf. Section 4.5). These techniques can be used to define the flow of execution of a procedure in techniques-based editors.

C.2 Designing a Technique to Compute Values from Singly-Recursive Procedures

In this example we show the definition of a programming technique relating the argument slices defined previously and a new output argument slice. The new output argument slice computes values as the flow of control of a singly-recursive procedure proceeds. This new technique is defined by firstly preparing an output argument slice and then appropriately linking its *required* values to those values offered in the argument slices above.

The output argument slice is prepared with its usage in mind. Let us assume that we are preparing a technique to build a list with a value `B` provided by another argument slice. This means our argument slice will be singly recursive: at each recursive step, use a value `B` supplied by another (singly-recursive) argument slice and partially build a list, by making `B` its head and recursively preparing its tail along the same lines. When the flow of execution stops then the list should be empty.

Let us suppose that the user decided to use the output argument slice shown above, that is

	$\mathcal{P}$`(A):-`	`{A/f}`	
`{A/f}`	*relation*`({A}).`	`{A/?}`	⟨⟨ *offer*`({A})` ⟩⟩
	$\mathcal{P}$`(A):-`	`{A/f,B/f,C/f}`	
`{A/f,B/f,C/f}`	*relation*`({A,B,C}),`	`{A/?,B/?,C/?}`	⟨⟨ *offer*`({A,B,C})` ⟩⟩
`{A/?,B/?,C/?}`	$\mathcal{P}$`(C).`	`{A/?,B/?,C/?}`	

Its mode-annotations can be refined, bearing in mind that its head variable `A` is initially free and should eventually get instantiated within the clause: this argument slice will compute a value and assign it to `A`. Variable `B`, required from another argument slice, should be instantiated before it is related to `A`. `C`, used in the recursive subgoal, is related to `A`, but maintains its free instantiation status since it should be assigned the partial computation of the recursive subgoal, eventually becoming ground. After the recursive subgoal is executed, `C` should become instantiated and so should `A`, given that `C` was the "loose end" of the partial computation assigned to `A`. The following refined argument slice, incorporating these considerations, can thus be obtained:

		$\mathcal{P}$`(A):-`	`{A/f}`	
	`{A/f}`	*relation*`({A}).`	`{A/g}`	⟨⟨ *offer*`({A})` ⟩⟩
		$\mathcal{P}$`(A):-`	`{A/f,B/f,C/f}`	
⟨⟨ *required*`({B})` ⟩⟩	`{A/f,B/g,C/f}`	*relation*`({A,B,C}),`	`{A/i,B/g,C/f}`	⟨⟨ *offer*`({A})` ⟩⟩
	`{A/i,B/g,C/f}`	$\mathcal{P}$`(C).`	`{A/g,B/g,C/g}`	⟨⟨ *offer*`({C})` ⟩⟩

The *relation* subgoal in the first clause is instantiating `A`. The *relation* subgoal in the first clause is employing the required variable `B` to perform a partial computation of `A`; the same subgoal relates `A` to the free variable `C` used recursively.

The *relation* subgoal of the first clause is next instantiated to *assign*: it will assign the empty list to `A`, at the end of the computation. The *relation* subgoal of the second clause is instantiated to the *build* descriptor, since it builds in `A` a list partially instantiated. The result of such instantiations is the construct

		$\mathcal{P}$(A):-	{A/f}	
	{A/f}	*assign*(∅,{A}).	{A/g}	⟨⟨ *offer*({A}) ⟩⟩
		$\mathcal{P}$(A):-	{A/f,B/f,C/f}	
⟨⟨ *required*({B}) ⟩⟩	{A/f,B/g,C/f}	*build*({B,C},A),	{A/i,B/g,C/f}	⟨⟨ *offer*({A}) ⟩⟩
	{A/i,B/g,C/f}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	⟨⟨ *offer*({C}) ⟩⟩

This procedural abstraction provides a robust representation of an argument slice building a singly-recursive data structure as the flow of execution proceeds. At this point the user may want to link the required variable `B` to an *offer* variable of another compatible argument slice, as described in Section 6.3. A good choice of such argument slice is the procedural abstraction of our previous example, that is

	$\mathcal{P}$(A):-	{A/g}	⟨⟨ *offer*({A}) ⟩⟩
{A/g}	*test*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,B/f,C/f}	⟨⟨ *offer*({A}) ⟩⟩
{A/g,B/f,C/f}	*decomp*(A,{B,C}),	{A/g,B/g,C/g}	⟨⟨ *offer*({B,C}) ⟩⟩
{A/g,B/f,C/f}	*test*({B}),	{A/g,B/g,C/g}	
{A/g,B/g,C/g}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	

The required `B` variable of our current example can be linked to the offered `B` variable of the component above since the argument slices are compatible. If the user decides to link these argument slices at this level of abstraction, an useful characterisation of a *class* of programming techniques is obtained, relating two procedural abstractions (cf. Section 6.3.3). A number of different programming techniques can be automatically defined via the different ways each of the argument slices can be specialised separately.

The predicate descriptors of our output argument slice await instantiation. The *assign* descriptor should assign the empty list to `A`; the *bind* descriptor should assign to `A` a list consisting of head `B` and body `C`. The following component is obtained after replacing these subgoals:

		$\mathcal{P}$(A):-	{A/f}	
	{A/f}	A = [].	{A/g}	⟨⟨ *offer*({A}) ⟩⟩
		$\mathcal{P}$(A):-	{A/f,B/f,C/f}	
⟨⟨ *required*({B}) ⟩⟩	{A/f,B/g,C/f}	A = [B|C],	{A/i,B/g,C/f}	⟨⟨ *offer*({A}) ⟩⟩
	{A/i,B/g,C/f}	$\mathcal{P}$(C).	{A/g,B/g,C/g}	⟨⟨ *offer*({C}) ⟩⟩

Within a techniques editor, an initial procedure devised using the traversal technique specialised to the alternative `baz(C,B)` data structure, that is

```
𝒫(A):-
   A = foo1.
𝒫(A):-
   A = foo2.
𝒫(A):-
   A = baz(C,B),
   𝒫(C).
```

can be enhanced by applying our technique to build a list devised above, yielding the procedure

```
𝒫(A,A1):-
   A = foo1,
   A1 = [].
𝒫(A,A1):-
   A = foo2,
   A1 = [].
𝒫(A,A1):-
   A = baz(C,B),
   A1 = [B|C1],
   𝒫(C,C1).
```

An altogether different candidate argument slice for the programming technique of this example would be the left-recursive construct

		𝒫(A):-	{A/f}	
	{A/f}	*relation*({A}).	{A/g}	⟪ *offer*({A}) ⟫
		𝒫(A):-	{A/f,B/f,C/f}	
	{A/f,B/f,C/f}	𝒫(C),	{A/f,B/f,C/g}	⟪ *offer*({C}) ⟫
⟪ *required*({B}) ⟫	{A/f,B/g,C/g}	*relation*({A,B,C}).	{A/g,B/g,C/g}	⟪ *offer*({A}) ⟫

The instantiation status of those variables related by the *relation* subgoal in the second clause would allow the specialisation of that subgoal as a calculation (descriptor *calc*), as well as a data structure being built.

Appendix D

Working Example

In this appendix we describe a working example, showing how our system can be used to help humans re-engineering pieces of software. Let us suppose that the following program is the object of our study:

```
evaluate(A,B,C):-
   A = var(D),
   value(D,B,C).
evaluate(A,B,C):-
   number(A),
   C = A.
evaluate(A,B,C):-
   A = D + E,
   evaluate(D,B,F),
   evaluate(E,B,G),
   C is F + G.
evaluate(A,B,C):-
   A = D - E,
   evaluate(D,B,F),
   evaluate(E,B,G),
   C is F - G.
evaluate(A,B,C):-
   A = D * E,
   evaluate(D,B,F),
   evaluate(E,B,G),
   C is F * G.
evaluate(A,B,C):-
   A = D/E,
   evaluate(D,B,F),
   evaluate(E,B,G),
   C is F/G.

value(A,B,C):-
   member(bind(A,C),B).
value(A,B,C):-
   C = 0.

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

The predicate *evaluate*/3 holds if its first argument is an arithmetic expression, its second argument is a list of variable bindings and its third argument is the result of evaluating the arithmetic expression with respect to the list of bindings. The expression does not make use of Prolog variables: it uses instead a Prolog term `var(`x`)` to refer to a variable x. The list of bindings consists of terms of the form `bind(`x`,`n`)` where x is the name of a variable and n is a number associated with it. When a variable has no binding then a 0 is by default associated with it.

D.1 Extraction of Programming Techniques

The initial stage of our working example consists of extracting the programming techniques of a chosen predicate. Our system supports the examination of the predicates of a given program: the user chooses the focused predicate and the query/queries with respect to which it is to be analysed. In our working example, we suppose that the user wants to examine the techniques of predicate *evaluate*/3 with respect to its usage when solving a query of the form

```
?- evaluate((var(x) + 3) * 9/var(y),[bind(x,2),bind(y,3)],Res).
```

The user is also offered a choice of employing an abstract or a concrete interpreter during the techniques' extraction. If the user provides the program and queries above and chooses to employ an abstract interpreter, the system produces the output window shown in Figure D.1. The tree in the figure shows the argument slices of the

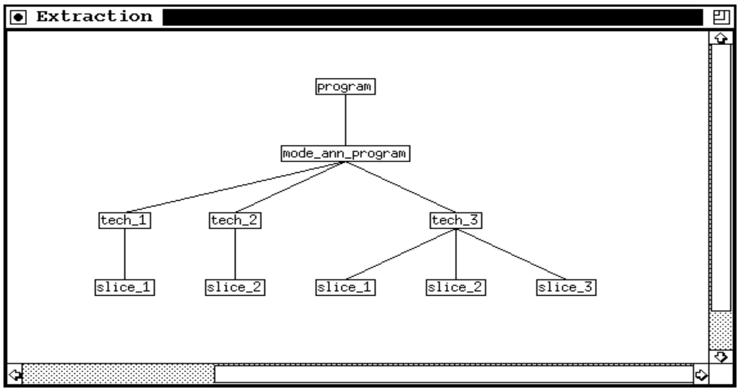


Figure D.1: Simplified description of *evaluate*/3 Programming Techniques

predicate, and how they are related as programming techniques: the first and second argument slices are programming techniques on their own, and the third argument slice is being used to define a technique together with the other argument slices. The inter-relationships between the different argument positions of a predicate are formally characterised by our concept of programming techniques.

The first argument slice is a technique to traverse an arithmetic expression; the second argument slice defines a technique to carry a value (a list of variable bindings, in this case) down recursive calls; and the appropriate combination of all argument slices defines a technique to compute in the third argument position the value of the equation in the first argument using the bindings in the second argument.

Each of the nodes of the tree can be examined by the user. They show the components of our analysis with their clause- and mode-annotations. If the user decides to view the mode-annotated version of the program, then the system displays the following construct:

```
                                   evaluate(A,B,C):-      {A/g,B/g,C/f,D/f}
                 {A/g,B/g,C/f,D/f}   A = var(D),          {A/g,B/g,C/f,D/g}
                 {A/g,B/g,C/f,D/g}   value(D,B,C).        {A/g,B/g,C/g,D/g}
                                   evaluate(A,B,C):-      {A/g,B/g,C/f}
                     {A/g,B/g,C/f}   number(A),           {A/g,B/g,C/f}
                     {A/g,B/g,C/f}   C = A.               {A/g,B/g,C/g}
                                   evaluate(A,B,C):-      {A/g,B/g,C/f,D/f,E/f,F/f,G/f}
     {A/g,B/g,C/f,D/f,E/f,F/f,G/f}   A = D + E,           {A/g,B/g,C/f,D/g,E/g,F/f,G/f}
     {A/g,B/g,C/f,D/g,E/g,F/f,G/f}   evaluate(D,B,F),     {A/g,B/g,C/f,D/g,E/g,F/g,G/f}
     {A/g,B/g,C/f,D/g,E/g,F/g,G/f}   evaluate(E,B,G),     {A/g,B/g,C/f,D/g,E/g,F/g,G/g}
     {A/g,B/g,C/f,D/g,E/g,F/g,G/g}   C is F + G.          {A/g,B/g,C/g,D/g,E/g,F/g,G/g}
                                   evaluate(A,B,C):-      {A/g,B/g,C/f,D/f,E/f,F/f,G/f}
     {A/g,B/g,C/f,D/f,E/f,F/f,G/f}   A = D - E,           {A/g,B/g,C/f,D/g,E/g,F/f,G/f}
     {A/g,B/g,C/f,D/g,E/g,F/f,G/f}   evaluate(D,B,F),     {A/g,B/g,C/f,D/g,E/g,F/g,G/f}
     {A/g,B/g,C/f,D/g,E/g,F/g,G/f}   evaluate(E,B,G),     {A/g,B/g,C/f,D/g,E/g,F/g,G/g}
     {A/g,B/g,C/f,D/g,E/g,F/g,G/g}   C is F - G.          {A/g,B/g,C/g,D/g,E/g,F/g,G/g}
                                   evaluate(A,B,C):-      {A/g,B/g,C/f,D/f,E/f,F/f,G/f}
     {A/g,B/g,C/f,D/f,E/f,F/f,G/f}   A = D * E,           {A/g,B/g,C/f,D/g,E/g,F/f,G/f}
     {A/g,B/g,C/f,D/g,E/g,F/f,G/f}   evaluate(D,B,F),     {A/g,B/g,C/f,D/g,E/g,F/g,G/f}
     {A/g,B/g,C/f,D/g,E/g,F/g,G/f}   evaluate(E,B,G),     {A/g,B/g,C/f,D/g,E/g,F/g,G/g}
     {A/g,B/g,C/f,D/g,E/g,F/g,G/g}   C is F * G.          {A/g,B/g,C/g,D/g,E/g,F/g,G/g}
                                   evaluate(A,B,C):-      {A/g,B/g,C/f,D/f,E/f,F/f,G/f}
     {A/g,B/g,C/f,D/f,E/f,F/f,G/f}   A = D/E,             {A/g,B/g,C/f,D/g,E/g,F/f,G/f}
     {A/g,B/g,C/f,D/g,E/g,F/f,G/f}   evaluate(D,B,F),     {A/g,B/g,C/f,D/g,E/g,F/g,G/f}
     {A/g,B/g,C/f,D/g,E/g,F/g,G/f}   evaluate(E,B,G),     {A/g,B/g,C/f,D/g,E/g,F/g,G/g}
     {A/g,B/g,C/f,D/g,E/g,F/g,G/g}   C is F/G.            {A/g,B/g,C/g,D/g,E/g,F/g,G/g}
```

The first clause- and mode-annotated argument slice is the construct

```
                 evaluate(A):-     {A/g,D/f}       ⟨⟨ offer({A}) ⟩⟩
     {A/g,D/f}     A = var(D).     {A/g,D/g}       ⟨⟨ offer({D}) ⟩⟩
                 evaluate(A):-     {A/g}           ⟨⟨ offer({A}) ⟩⟩
         {A/g}     number(A).      {A/g}
                 evaluate(A):-     {A/g,D/f,E/f}   ⟨⟨ offer({A}) ⟩⟩
 {A/g,D/f,E/f}     A = D + E,      {A/g,D/g,E/g}   ⟨⟨ offer({D,E}) ⟩⟩
 {A/g,D/g,E/g}     evaluate(D),    {A/g,D/g,E/g}
 {A/g,D/g,E/g}     evaluate(E).    {A/g,D/g,E/g}
                 evaluate(A):-     {A/g,D/f,E/f}   ⟨⟨ offer({A}) ⟩⟩
 {A/g,D/f,E/f}     A = D - E,      {A/g,D/g,E/g}   ⟨⟨ offer({D,E}) ⟩⟩
 {A/g,D/g,E/g}     evaluate(D),    {A/g,D/g,E/g}
 {A/g,D/g,E/g}     evaluate(E).    {A/g,D/g,E/g}
                 evaluate(A):-     {A/g,D/f,E/f}   ⟨⟨ offer({A}) ⟩⟩
 {A/g,D/f,E/f}     A = D * E,      {A/g,D/g,E/g}   ⟨⟨ offer({D,E}) ⟩⟩
 {A/g,D/g,E/g}     evaluate(D),    {A/g,D/g,E/g}
 {A/g,D/g,E/g}     evaluate(E).    {A/g,D/g,E/g}
                 evaluate(A):-     {A/g,D/f,E/f}   ⟨⟨ offer({A}) ⟩⟩
 {A/g,D/f,E/f}     A = D/E,        {A/g,D/g,E/g}   ⟨⟨ offer({D,E}) ⟩⟩
 {A/g,D/g,E/g}     evaluate(D),    {A/g,D/g,E/g}
 {A/g,D/g,E/g}     evaluate(E).    {A/g,D/g,E/g}
```

This argument slice recursively decomposes an arithmetic expression consisting of any number of nestings of the infix operators `+`, `-`, `*` and `/`, until its basic components, `var(X)` and numbers, are reached.

The second clause- and mode-annotated argument slice is

```
          evaluate(B).       {B/g}   ⟨⟨ offer({B}) ⟩⟩
          evaluate(B).       {B/g}   ⟨⟨ offer({B}) ⟩⟩
          evaluate(B):-      {B/g}   ⟨⟨ offer({B}) ⟩⟩
{B/g}        evaluate(B),    {B/g}
{B/g}        evaluate(B).    {B/g}
          evaluate(B):-      {B/g}   ⟨⟨ offer({B}) ⟩⟩
{B/g}        evaluate(B),    {B/g}
{B/g}        evaluate(B).    {B/g}
          evaluate(B):-      {B/g}   ⟨⟨ offer({B}) ⟩⟩
{B/g}        evaluate(B),    {B/g}
{B/g}        evaluate(B).    {B/g}
          evaluate(B):-      {B/g}   ⟨⟨ offer({B}) ⟩⟩
{B/g}        evaluate(B),    {B/g}
{B/g}        evaluate(B).    {B/g}
```

It simply carries a value down the recursive calls. Finally the third clause- and mode-annotated argument slice is

```
                                         evaluate(C):-      {B/g,C/f,D/f}
⟨⟨ required({B,D}) ⟩⟩  {B/g,C/f,D/g}        value(D,B,C).   {B/g,C/g,D/g}    ⟨⟨ offer({C}) ⟩⟩
                                         evaluate(C):-      {A/g,C/f}
  ⟨⟨ required({A}) ⟩⟩      {A/g,C/f}        C = A.          {A/g,C/g}        ⟨⟨ offer({C}) ⟩⟩
                                         evaluate(C):-      {C/f,F/f,G/f}
                       {C/f,F/f,G/f}        evaluate(F),    {C/f,F/g,G/f}    ⟨⟨ offer({F}) ⟩⟩
                       {C/f,F/g,G/f}        evaluate(G),    {C/f,F/g,G/g}    ⟨⟨ offer({G}) ⟩⟩
                       {C/f,F/g,G/g}        C is F + G.     {C/g,F/g,G/g}    ⟨⟨ offer({C}) ⟩⟩
                                         evaluate(C):-      {C/f,F/f,G/f}
                       {C/f,F/f,G/f}        evaluate(F),    {C/f,F/g,G/f}    ⟨⟨ offer({F}) ⟩⟩
                       {C/f,F/g,G/f}        evaluate(G),    {C/f,F/g,G/g}    ⟨⟨ offer({G}) ⟩⟩
                       {C/f,F/g,G/g}        C is F - G.     {C/g,F/g,G/g}    ⟨⟨ offer({C}) ⟩⟩
                                         evaluate(C):-      {C/f,F/f,G/f}
                       {C/f,F/f,G/f}        evaluate(F),    {C/f,F/g,G/f}    ⟨⟨ offer({F}) ⟩⟩
                       {C/f,F/g,G/f}        evaluate(G),    {C/f,F/g,G/g}    ⟨⟨ offer({G}) ⟩⟩
                       {C/f,F/g,G/g}        C is F * G.     {C/g,F/g,G/g}    ⟨⟨ offer({C}) ⟩⟩
                                         evaluate(C):-      {C/f,F/f,G/f}
                       {C/f,F/f,G/f}        evaluate(F),    {C/f,F/g,G/f}    ⟨⟨ offer({F}) ⟩⟩
                       {C/f,F/g,G/f}        evaluate(G),    {C/f,F/g,G/g}    ⟨⟨ offer({G}) ⟩⟩
                       {C/f,F/g,G/g}        C is F/G.       {C/g,F/g,G/g}    ⟨⟨ offer({C}) ⟩⟩
```

It assigns a value to its head variable `C`, either by using the *value*/3 predicate with required values from other slices (first clause), by assigning a required `A` value (second clause), or by carrying out an appropriate calculation by means of the *is*/2 built-in, employing values obtained in recursive calls.

D.2 Storing Extracted Techniques

The user can choose which of the extracted techniques above are to be inserted in the library of our system. Each chosen technique is inserted in the library together with its more abstract versions in which design decisions are progressively concealed to a generic template.

The abstraction and insertion processes are fully automated and require no user intervention or supervision. The user can view the abstraction tree that depicts the gradual generalisation of the design decisions of each argument slice comprising a programming technique. The abstraction of the argument slices comprising the techniques of our example yields the tree depicted in Figure D.2. Each of the components of the tree can be inspected at the user's request. The components of the tree are given shortened

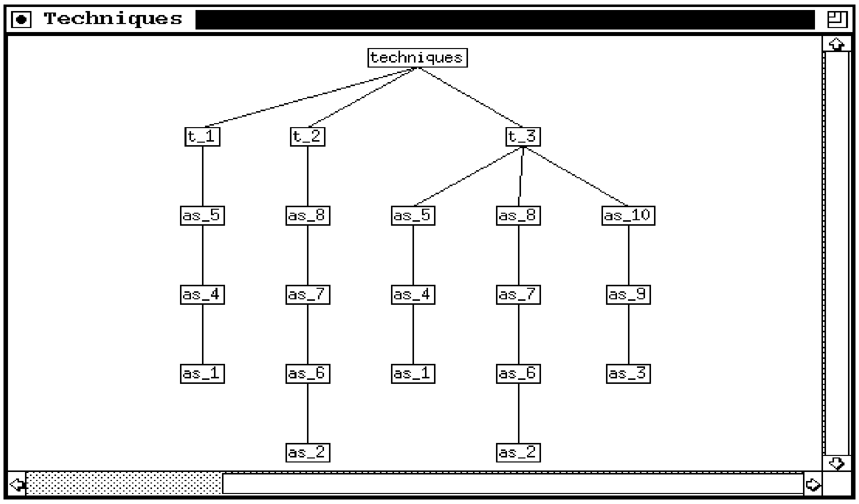


Figure D.2: Programming Techniques and their Abstract Argument Slices

names so as to make it easier to fit them as nodes of a tree, also providing an economic way to address the actual content of the node.

Node `as_5` stands for the most abstract version of the first clause- and mode-annotated argument slice. It is of the form

	$\mathcal{P}$(A):-	{A/g,D/f}	⟪ *offer*({A}) ⟫
{A/g,D/f}	*relation*({A,D}).	{A/g,D/g}	⟪ *offer*({D}) ⟫
	$\mathcal{P}$(A):-	{A/g}	⟪ *offer*({A}) ⟫
{A/g}	*relation*({A}).	{A/g}	
	$\mathcal{P}$(A):-	{A/g,D/f,E/f}	⟪ *offer*({A}) ⟫
{A/g,D/f,E/f}	*relation*({A,D,E}),	{A/g,D/g,E/g}	⟪ *offer*({D,E}) ⟫
{A/g,D/g,E/g}	$\mathcal{P}$(D),	{A/g,D/g,E/g}	
{A/g,D/g,E/g}	$\mathcal{P}$(E).	{A/g,D/g,E/g}	

Those clauses that were abstracted to the same generic format were collapsed together, thus rendering a simpler argument slice. Node `as_10` is the most abstract form of the third argument slice. It is of the form

		$\mathcal{P}$(C):-	{B/g,C/f,D/f}	
⟪ *required*({B,D}) ⟫	{B/g,C/f,D/g}	*relation*({B,C,D}).	{B/g,C/g,D/g}	⟪ *offer*({C}) ⟫
		$\mathcal{P}$(C):-	{A/g,C/f}	
⟪ *required*({A}) ⟫	{A/g,C/f}	*relation*({C,A}).	{A/g,C/g}	⟪ *offer*({C}) ⟫
		$\mathcal{P}$(C):-	{C/f,F/f,G/f}	
	{C/f,F/f,G/f}	$\mathcal{P}$(F),	{C/f,F/g,G/f}	⟪ *offer*({F}) ⟫
	{C/f,F/g,G/f}	$\mathcal{P}$(G),	{C/f,F/g,G/g}	⟪ *offer*({G}) ⟫
	{C/f,F/g,G/g}	*relation*({C,F,G}).	{C/g,F/g,G/g}	⟪ *offer*({C}) ⟫

The links between the argument slices, in the form of clause-annotations, are preserved during the abstraction process. Programming techniques are still properly depicted by means of its argument slices in more abstract versions. Technique `t_3` is the proper combination of the abstract versions of its argument slices.

The initial clause- and mode-annotated argument slices, their abstracted versions and the techniques are appropriately stored in our library, as described in Chapter 5, upon the user's request.

D.3 Reusing Argument Slices and Defining New Programming Techniques

The argument slices in our library can be examined and used as a starting point for the definition of new components. The δ-relations used to abstract the design decisions of the argument slices (Section 5.7) are also employed in this task. In the argument slices below we have omitted the clause- and mode-annotations for the sake of conciseness. Although we do not use them in our explanation below, they are essential to the abstraction and reimplementation process.

The procedural abstraction `as_4` of the first argument slice of our example is the construct

$$\boxed{\begin{array}{l}\mathcal{P}\texttt{(A):-}\\ \quad \mathit{decomp}\texttt{(A,}\{\texttt{D}\}\texttt{).}\\ \mathcal{P}\texttt{(A):-}\\ \quad \mathit{test}\texttt{(}\{\texttt{A}\}\texttt{).}\\ \mathcal{P}\texttt{(A):-}\\ \quad \mathit{decomp}\texttt{(A,}\{\texttt{D,E}\}\texttt{),}\\ \quad \mathcal{P}\texttt{(D),}\\ \quad \mathcal{P}\texttt{(E).}\end{array}}$$

A slightly more abstract form of this construct can be obtained, in which the *test* descriptor of the second clause is further abstracted as a *relation* descriptor:

$$\boxed{\begin{array}{l}\mathcal{P}\texttt{(A):-}\\ \quad \mathit{decomp}\texttt{(A,}\{\texttt{D}\}\texttt{).}\\ \mathcal{P}\texttt{(A):-}\\ \quad \mathit{relation}\texttt{(}\{\texttt{A}\}\texttt{).}\\ \mathcal{P}\texttt{(A):-}\\ \quad \mathit{decomp}\texttt{(A,}\{\texttt{D,E}\}\texttt{),}\\ \quad \mathcal{P}\texttt{(D),}\\ \quad \mathcal{P}\texttt{(E).}\end{array}}$$

The user can redesign these higher-level descriptions of subgoals with the intention of adapting the argument slice to other purposes. Our design service (Chapter 6) supports the gradual specification of abstract subgoals, ruling out incorrect specialisations and supplying the set of applicable specialisations (δ-relations). Let us assume that the user is adapting the argument slice above to a data structure representing a logic expression consisting of operators `and` and `or`, constants `t` and `f` and a term `prop(`x`)` to denote atomic formulae, for instance,

```
(t and prop(p)) or (f and prop(q))
```

Since there are two possible ways to recursively decompose the formulae (either as a conjunction or as a disjunction), the user would first have to copy the third clause, thus obtaining:

```
𝒫(A):-
  decomp(A,{D}).
𝒫(A):-
  relation({A}).
𝒫(A):-
  decomp(A,{D,E}),
  𝒫(D),
  𝒫(E).
𝒫(A):-
  decomp(A,{D,E}),
  𝒫(D),
  𝒫(E).
```

The *decomp* descriptors must be specialised one at a time, as $x = f(\vec{V})$ constructs supplied by the user. In order to address the possible cases of the targeted data structure, the following instantiations should be performed:

1. *decomp*(A,{D}) in the first clause is instantiated as `A = prop(D)` (atomic formulae);
2. *decomp*(A,{D,E}) in the third clause is instantiated as `A = D and E`;
3. *decomp*(A,{D,E}) in the fourth clause is instantiated as `A = D or E`.

The last two instantiations are interchangeable. The result of applying these instantiations to the abstract slice above is

```
𝒫(A):-
  A = prop(D).
𝒫(A):-
  relation({A}).
𝒫(A):-
  A = D and E,
  𝒫(D),
  𝒫(E).
𝒫(A):-
  A = D or E,
  𝒫(D),
  𝒫(E).
```

The *relation* descriptor still awaits instantiation. It is suitable to address the truth-values `t` and `f` of formulae. Let us assume that there exists a user-defined predicate *constant*/1 that tests if a given value is equal to `t` or `f`. The user can then specialise the *relation*: it must initially be specialised as *user-pred* and then as the *constant*/1 predicate. The result of this specialisation is the new argument slice:

```
𝒫(A):-
  A = prop(D).
𝒫(A):-
  constant(A).
𝒫(A):-
  A = D and E,
  𝒫(D),
  𝒫(E).
𝒫(A):-
  A = D or E,
  𝒫(D),
  𝒫(E).
```

This argument slice defines a programming technique traversing propositional formulae represented using the data structure mentioned above.

The practice of reusing existing components to define new, similar constructs relies on the user's expertise to choose an appropriate starting point to carry out the alterations. The initial component must bear some features that favour its choice, such as the number of recursive calls, modes of variables, existing programming practices (as depicted by the predicate descriptors), and so on. Since the original first argument slice is a technique on its own, this new component is also a technique traversing a propositional formula.

Let us suppose the user wants to prepare a technique to obtain the truth-value of the formulae traversed by the argument slice devised above. The atomic formulae will have their associated truth-values stored in a list of bindings, as the variables in the arithmetic expressions of the initial program. The argument slice responsible for the preparation of the truth value can be designed by adapting the third argument slice of our initial program. Let us suppose, for instance, that the user looks up the procedural abstraction of the third argument slice (node `as_9`), of the form

```
𝒫(C):-
  user_pred({B,C,D}).
𝒫(C):-
  assign({A},{C}).
𝒫(C):-
  𝒫(F),
  𝒫(G),
  calc({F,G},C).
```

The *calc* descriptor in the last clause renders the construct inadequate because the *is*/2 predicate is not able to cope with truth values. The user can, however, abstract that particular subgoal slightly further, obtaining a *relation* descriptor in its place:

```
𝒫(C):-
  user_pred({B,C,D}).
𝒫(C):-
  assign({A},{C}).
𝒫(C):-
  𝒫(F),
  𝒫(G),
  relation({C,F,G}).
```

The user may then customise this construct differently, employing our design tool. Since there are two distinct recursive cases (`and` and `or` operators), the user must first duplicate the third clause, thus obtaining the construct

```
𝒫(C):-
  user_pred({B,C,D}).
𝒫(C):-
  assign({A},{C}).
𝒫(C):-
  𝒫(F),
  𝒫(G),
  relation({C,F,G}).
𝒫(C):-
  𝒫(F),
  𝒫(G),
  relation({C,F,G}).
```

The predicate descriptors await instantiation. The *relation* subgoals are responsible for assigning a value to `C`, using the values of `F` and `G` (this information is given by the mode-annotations, not shown here). In the problem being considered, *relation* uses the truth-value of the subformulae `F` and `G` to obtain the truth-value of their conjunction or disjunction. To cover the many different cases, the user may use a procedure defined as

```
truth_and(A,B,C):-
   A = t,
   B = t,
   C = t.
truth_and(A,B,C):-
   C = f.
truth_or(A,B,C):-
   A = t,
   C = t.
truth_or(A,B,C):-
   B = t,
   C = t.
truth_or(A,B,C):-
   C = f.
```

and instantiate the *relation* descriptors firstly as *user-pred*, then as *truth_and*/3 and *truth_or*/3, thus obtaining

```
𝒫(C):-
  user_pred({B,C,D}).
𝒫(C):-
  assign({A},{C}).
𝒫(C):-
  𝒫(F),
  𝒫(G),
  truth_and(F,G,C).
𝒫(C):-
  𝒫(F),
  𝒫(G),
  truth_or(F,G,C).
```

The remaining base cases consist of obtaining the truth value of atomic formulae and the constants. To cope with the atomic formulae, the *user-pred* descriptor of the first clause can be instantiated to a similar *value*/3 procedure as in the original argument slice, that is,

```
value(A,B,C):-
   member(bind(A,C),B).
value(A,B,C):-
   C = f.

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

This decision assumes that the truth-values associated with the atomic formulae are stored as a list of pairs `bind(`x`,`v`)`, where x is the propositional formula and v is the truth-value `t` or `f` associated with it. The outcome of this decision is the argument slice

```
𝒫(C):-
  value(D,B,C).
𝒫(C):-
  assign({A},{C}).
𝒫(C):-
  𝒫(F),
  𝒫(G),
  truth_and(F,G,C).
𝒫(C):-
  𝒫(F),
  𝒫(G),
  truth_or(F,G,C).
```

Finally, the *assign* descriptor must be specified, addressing the constant values `t` and `f`. This base case is solved by having the `A` value supplied as a constant, assigned to `C`, that is

```
𝒫(C):-
  value(D,B,C).
𝒫(C):-
  C = A.
𝒫(C):-
  𝒫(F),
  𝒫(G),
  truth_and(F,G,C).
𝒫(C):-
  𝒫(F),
  𝒫(G),
  truth_or(F,G,C).
```

The links between the original components are maintained throughout the manipulations above. The variables `A`, `D` and `B` referred to in the first and second clauses of the new argument slice above are those of the first argument slice (`A` and `D`) and the second argument slice (`B`). A new technique was automatically defined by having each of its components separately re-engineered.

The kinds of syntactic transformations the δ-relations perform require that the original and new components must share fundamental properties, such as the number of recursive calls and the instantiation mode of its variables. Thus the user trying to reuse and adapt existing components should choose an appropriate candidate as a starting point.

The newly defined programming technique can be put to use in a techniques-based program development environment in which a procedure is developed via the successive application of techniques. Starting with a technique defining the flow of control of the execution, enhancements are gradually added to it, until a final procedure is obtained. These enhancements are other techniques which may or may not have links to the initial technique. If the techniques above and their new argument slices are put to use in a techniques-editing system, the following single-argument procedure can be initially defined

```
truth_value(A):-
   A = prop(D).
truth_value(A):-
   constant(A).
truth_value(A):-
   A = D and E,
   truth_value(D),
   truth_value(E).
truth_value(A):-
   A = D or E,
   truth_value(D),
   truth_value(E).
```

It defines the flow of execution (the *skeleton* [SS86, KLS89, Lak89, SK93]) of the procedure being devised. This initial procedure can be extended by applying the technique `t_2` to carry a value down the recursive calls, thus obtaining

```
truth_value(A,B):-
   A = prop(D).
truth_value(A,B):-
   constant(A).
truth_value(A,B):-
   A = D and E,
   truth_value(D,B),
   truth_value(E,B).
truth_value(A,B):-
   A = D or E,
   truth_value(D,B),
   truth_value(E,B).
```

We can apply the technique to obtain the truth-value of the formula being manipulated to this procedure, thus obtaining

```
truth_value(A,B,C):-
   A = prop(D),
   value(D,B,C).
truth_value(A,B,C):-
   constant(A),
   C = A.
truth_value(A,B,C):-
   A = D and E,
   truth_value(D,B,F),
   truth_value(E,B,G),
   truth_and(F,G,C).
truth_value(A,B,C):-
   A = D or E,
   truth_value(D,B,F),
   truth_value(E,B,G),
   truth_or(F,G,C).
```

The insertion of subgoals in the appropriate places and variable binding is only possible because each argument slice has information concerning its required and offered variables (the clause-annotations). The auxiliary procedures are defined as

```
value(A,B,C):-
   member(bind(A,C),B).
value(A,B,C):-
   C = f.

member(A,B):-
   B = [A|C].
member(A,B):-
   B = [C|D],
   member(A,D).

constant(A):-
   A = t.
constant(A):-
   A = f.

truth_and(A,B,C):-
   A = t,
   B = t,
   C = t.
truth_and(A,B,C):-
   C = f.
truth_or(A,B,C):-
   A = t,
   C = t.
truth_or(A,B,C):-
   B = t,
   C = t.
truth_or(A,B,C):-
   C = f.
```

Appendix E

Glossary and Auxiliary Definitions

In this appendix we list the notational shorthands and conventions used throughout the dissertation.

E.1 Notation

E.1.1 Programs, Clauses and Subgoals

We need some notation to express the manipulation of mode-annotated clauses within programs and mode-annotate subgoals within clauses. We shall employ *vectors* to refer to a possibly empty sequence of mode-annotate clauses in a program, mode-annotate subgoals in a clause, and variables within a subgoal: $\vec{C}$ stands for a possibly empty sequence $\widetilde{C}_0, \ldots, \widetilde{C}_n$ where each $\widetilde{C}_i$ is a mode-annotated clause, $\vec{S}$ stands for a possibly empty sequence $\widetilde{S}_0, \ldots, \widetilde{S}_n$ where each $\widetilde{S}_i$ is a mode-annotated subgoal, and $\vec{V}$ stands for a possibly empty sequence $x_0, \ldots, x_n$ of Prolog variables.

According to this notation the pattern $\widetilde{P} = \vec{C}$ describes a mode-annotated program $\widetilde{P}$ consisting of a possibly empty vector of mode-annotated clauses. Program and clause patterns can be more sophisticated, such as $\widetilde{P} = \vec{C}\,\widetilde{C}'$ in which a reference to the last clause $\widetilde{C}$ of $\widetilde{P}$ is made. Further elaborations can be conceived, such as $\widetilde{P} = \vec{C}_1\,\widetilde{C}\,\vec{C}_2\,\widetilde{C}'\,\vec{C}$ describing a mode-annotated program consisting of at least two mode-annotated clauses $\widetilde{C}$ and $\widetilde{C}'$. Similar constructions can be devised to describe mode-annotated clauses and their subgoals and variables of subgoals.

The vector notation provides us with a clean and economic manner to address specific parts of a mode-annotated program or clause, without having to explicitly refer to their implementational details. For instance, the conjunction $(\widetilde{P} = \vec{C}_1\widetilde{C}\vec{C}_2) \wedge test(\widetilde{C})$ which holds if there is a mode-annotated clause $\widetilde{C}$ in $\widetilde{P}$ satisfying predicate *test* is such that there are no references whatsoever as to how $\widetilde{P}$ has been represented/implemented nor as to how $\widetilde{P}$ has had its clauses examined until $\widetilde{C}$, satisfying *test*, had been found.

Sometimes in the δ-relations we must refer to portions of the clause whose details are not relevant. In such cases we employ vectors of clause- and mode-annotated subgoals $\vec{S_i}$ and $\vec{S_i^\alpha}$ of the following form

$$\vec{S_i} = \boxed{\gamma_{[i,0]}\ \theta_{[i,0]}\ S_{[i,0]}\ \theta'_{[i,0]}\ \ \gamma'_{[i,0]}}, \ldots, \boxed{\gamma_{[i,n_i]}\ \ \theta_{[i,n_i]}\ S_{[i,n_i]}\ \gamma'_{[i,n_i]}\ \ \theta'_{[i,n_i]}}$$

$$\vec{S_i^\alpha} = \boxed{\gamma_{[i,0]}\ \theta^\alpha_{[i,0]}\ S_{[i,0]}\ {\theta'}^\alpha_{[i,0]}\ \ \gamma'_{[i,0]}}, \ldots, \boxed{\gamma_{[i,n_i]}\ \ \theta^\alpha_{[i,n_i]}\ S_{[i,n_i]}\ \gamma'_{[i,n_i]}\ \ {\theta'}^\alpha_{[i,n_i]}}$$

such that $\vec{S_i^\alpha}$ corresponds to $\vec{S_i}$ in the right-hand side of the δ-relations, its abstract version, and it is such that only the mode-annotations are altered. Alternatively, $\vec{S_i}$ can be seen as the more specific version of $\vec{S_i^\alpha}$.

E.1.2 Declarative Definitions

In this presentation we have sometimes used a declarative approach to describe our proposed solutions (*e.g.* Definitions (refs)). A first-order Horn-clause formalism has been employed to that extent, of the form

$$L \Longleftarrow R_1 \wedge \cdots \wedge R_n$$

where L and $R_1, \ldots, R_n$ are first-order terms. Such formulae can be understood as in conventional logic programming: L holds if $R_1, \ldots, R_n$ also hold. Our declarative definitions can safely be understood as Prolog programs — different symbols like $\Longleftarrow$, $\wedge$ and $\vee$ have been used so as to make a distinction between the Prolog programs these definitions are aimed at, the notation employed to refer to Prolog constructs and the definitions themselves.

E.2 Sequences

In order to represent a collection of objects such that the ordering of its components is important we will employ *sequences*. They are defined as follows:

Definition E.2.1 A sequence S is defined recursively as:

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

$\mathsf{S} = \langle obj, \mathsf{S}'\rangle$, where *obj* is any data item, is a sequence, if S' is a sequence.

E.3 Auxiliary Predicates

We define two auxiliary relations on sequences. The first relation can be used to obtain the position (in terms of a natural number) of a data item in a sequence, being defined as

Definition E.3.1 Given $\mathsf{S} = \langle obj_1, \ldots, obj_n \rangle$ and a data item obj_j, the *position* n of obj_j with respect to S, $pos(obj_j, \mathsf{S}, n)$, is such that:

$$\begin{array}{rcl} pos(obj_j, \langle obj_j, \mathsf{S}' \rangle, 1). & & \\ pos(obj_j, \langle obj_i, \mathsf{S}' \rangle, m+1) & \Longleftarrow & obj_j \neq obj_i \wedge pos(obj_j, \mathsf{S}', m). \end{array}$$

The *pos* relation can also be applied to a clause to find the position of a subgoal S in its body and hence we need to add the following:

$$pos(S, H \text{ :- } \mathsf{B}, pos) \Longleftarrow pos(S, \mathsf{B}, pos)$$

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

Definition E.3.2 Given two subsequences T and N of a sequence S and a data item *obj*, the relation $insertion(obj, \mathsf{S}, \mathsf{T}, \mathsf{N})$ holds if

$$\begin{array}{rcl} insertion(obj, \mathsf{S}, \langle\, \rangle, \langle obj \rangle). & & \\ insertion(obj, \mathsf{S}, \langle obj, T' \rangle, \langle obj, T' \rangle). & & \\ insertion(obj, \mathsf{S}, \langle obj', \mathsf{T}' \rangle, \langle obj, obj', \mathsf{T}' \rangle) & \Longleftarrow & pos(obj, \mathsf{S}, n) \wedge pos(obj', \mathsf{S}, n') \wedge \\ & & n < n'. \\ insertion(obj, \mathsf{S}, \langle obj', \mathsf{T}' \rangle, \langle obj', \mathsf{N}' \rangle) & \Longleftarrow & pos(obj, \mathsf{S}, n) \wedge pos(obj', \mathsf{S}, n') \wedge \\ & & n \geq n' \wedge insertion(obj, \mathsf{S}, \mathsf{T}', \mathsf{N}'). \end{array}$$

The definition of *insertion* compares the *position* n' of the first element obj' of the sequence T (with respect to the initial sequence S) and the *position* n of *obj*: if n is less than n' (third case) the new sequence N is T with *obj* inserted before obj'; otherwise (fourth case) obj' 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 *obj* is already found.