Using Prolog Techniques to Guide Program Composition

María D. S. Vargas-Vera

Ph.D.
University of Edinburgh
May 1995

Declaration

Some of the material reported in this thesis has been published:

- M.Vargas-Vera, D. Robertson and R. Inder. Combining Prolog Programs in a Techniques Editing System. *Third International Workshop on Logic Programming Synthesis and Transformation*, Springer Verlag, July 1993. (Research Paper 636, Dept. of Artificial Intelligence, University of Edinburgh, 1993).

- M. Vargas-Vera, W. W. Vasconcelos, and D. Robertson. Building Large-Scale Prolog Programs using a Techniques Editing System. Presented as poster in *International Logic Programming Symposium.* MIT Press, October 1993. (Research Paper 635, Dept. of Artificial Intelligence, University of Edinburgh, 1993).

- A. Bowles, D. Robertson, W. Vasconcelos, M. Vargas-Vera and D. Bental. Applying Prolog Programming Techniques. *International Journal of Human-Computer Studies*, 41(3):329-350, 1994. (Research Paper 641, Dept. of Artificial Intelligence, University of Edinburgh, 1993).

- M.Vargas-Vera and D. Robertson. An Environment for Building Prolog Programs Based on Knowledge about their Construction. The *10th Workshop on Logic Programming (WLP 94)*, University of Zurich, October 1994. (Research Paper 718, Dept. of Artificial Intelligence, University of Edinburgh, 1994).

I declare that this thesis was composed by myself and that the work described in it is my own.

Maria Vargas-Vera

Edinburgh

May 1995

Abstract

It is possible to build complex programs by repeated combination of pairs of simpler programs. However, naive combination often produces programs that are far too inefficient. We would like to have a system that would produce the optimal combination of two programs, and also work with minimal supervision by the user. In this thesis we make a significant step towards such an ideal, with the presentation of an interactive system based on program transformation complemented with knowledge of the program development.

No single method is known that will combine all programs efficiently and so a variety of different combination methods must be used. However, to get good results it is necessary that the methods have access to knowledge about the program structure. To provide this knowledge we have decided to require that the initial programs be constructed in a specialised editor which embodies knowledge of certain standard Prolog practices (*techniques*) to aid the program construction, but more importantly can record pertinent parts of the program development into a structure called the *program history*. This program history contains the initial control flow (*skeleton*) and the techniques that the user applied in the construction of the program. Hence it carries knowledge about the program that would otherwise be very difficult to extract from just the program itself.

The first contribution of this thesis is to recognise that knowledge contained in the *program history* can be used in program transformation, reducing the need for user interaction. The interactive composition system presented can automatically take major decisions, such as the selection of which subgoal should be unfolded or the laws to be applied in order to get a more efficient combined program. Furthermore, a component of our system called the *selection procedure* can decide automatically which is the most suitable combination method by analysing the characteristics of the initial pair of programs as given by their program histories. Approaches that do not use the program history suffer from the problem that it is not always practical to extract the required information about the structure of the program.

Our second contribution is to provide a range of new methods which exploit the program history in order to produce more efficient programs, and to deal with a wider range of combination problems. The new methods not only combine programs with the same control flow, but can also deal with some cases in which the control flows are different. All of these methods are completely automatic with the exception of our "mutant" method in which the combined clause needs to be approved by the user.

The third contribution is to present relevant properties in our composition system. These properties fall into the following three groups: (i) properties which hold after applying each combination method. (ii) properties about the type of program which is obtained after the combination. (iii) properties of the join specification which defines the characteristics of the combined program.

Acknowledgements

I would like to express my gratitude to the following people:

To my supervisor David Robertson, in the AI department, for his encouragement and guidance throughout the period of my studies. Key ideas can be traced from his previous work in the implementation of a techniques editor for novice programmers. To my supervisor Robert Inder, in the Human Communication Research Center, for his comments and suggestions which helped me through the development of this thesis.

To Alan Bundy (AI department) from whom I received suggestions and feedback. To Paul Brna (AI department) for his useful directions in the understanding of Bundy's editor.

To my colleagues in the Department of Artificial Intelligence who provided help in many ways and made my stay in Edinburgh more pleasant: Wamberto Vasconcelos, Andy Bowles, Diana Bental, Brian Ross, Rob Scott, Suresh Manandhar, Alan Black, Renato Bussato, Emilio Agustin, Peter Nowell, John Beaven, Sue Sentance, Keiichi Nakata, Susan Bull, Sheila Rock, Sheila Glasbey, Jeremy Crowe, Edward Carter, Cecilia German, Peter Funk, Chris Gathercole, Judith Good, Andrew Parkes, Elena Perez-Minana, Rolando Carrera and Hon Wing Charles Mak. I specially acknowledge Ian Lewin for his useful comments on my work. Thanks also to Ann Ralls, Michael Keightley and Craig Strachan for the computing support offered in the AI department.

I would like to express my gratitude to Leon Sterling from Case Western Reserve University for his invaluable advice in the early stage of this work. Also my gratitude to Alberto Pettorossi from Universita di Roma Tor Vergata for his useful suggestions about my work; the National University of Mexico for providing financial support during my postgraduates studies in Edinburgh and also to the Department of Artificial Intelligence who gave me support to attend some International conferences on Logic Programming.

Contents

List of Figures

List of Tables

1

Introduction

Program combination can be used to promote the reuse of standard programs more efficiently by allowing complex programs to be built by repeated combination of them. The motivation for this thesis arose from the idea of having an automatic system which assists programmers in the task of combining programs whilst requiring little user interaction.

There have been previous attempts in this direction and these are described below. However, these solutions have generally required considerable interaction from a user with a good understanding of the particular program transformation process being applied. In this thesis we present an approach to the combination of Prolog programs that requires little user interaction.

Our work has been influenced by previous work on the composition problem (mainly based on program transformation techniques) and techniques editing systems (tools which help in program construction). From the program transformation viewpoint the combination problem can be seen as a novel application of transformation rather than their orthodox use in program synthesis. A system for transforming programs expressed as recursion equations is given in [Burstall & Darlington 77], but its use requires intervention of a human with a good understanding of program transformation methods. In procedural languages, there are ways [Horwitz *et al.* 88] to merge programs derived from an initial generic template, however, this approach is restricted to a limited class of programs. In

[Tamaki & Sato 83, Tamaki & Sato 84], an unfold/fold based transformation system was given, but requires user intervention and is restricted to programs with the same flow of control. In [Lakhotia & Sterling 87, Sterling & Lakhotia 88] methods were given for combining Prolog programs with the same basic flow of control (meta-interpreters) but these also require user intervention. The method of [Fuchs & Fromherz 91] employs basic schemata (supplied by an expert) to combine list-processing programs. The method is very efficient, however it can present the user with a difficult choice between possible output schemata. The method in [Proietti & Pettorossi 92] combines logic programs with the same flow of control using basic fold/unfold transformations, but relies on user-guidance and its efficiency depends on the decisions made by the user.

We have also been influenced by ideas of some editing systems which allows user to build Prolog programs by means of a sequence of refinements to the initial program specification. There are several editors which automatically implement this program construction methodology (see Chapter 3). We found which of these editors can provide the high-level description required by our combination system.

Existing combination systems typically require direction by an informed user and are very restricted in the kinds of programs that could successfully be combined.

Given these problems we started out with two basic ideas as to how the situation could be improved.

- If high-level description about the programs were available then we expected to be able to combine a wider range of programs, and also do it more efficiently and produce better output programs.

- Such high-level description was potentially available from Prolog techniques editors (tools for developing programs), by modifying them so as to record the history of the program development (which we call the *program history*). The advantage of using the program history is that it would give information about the program that would otherwise be difficult to re-extract from the finished program.

To investigate these ideas we took Sterling's notion of initial control flows[1], extended it with additional classes and developed it into a program classification scheme.

We then designed combination methods that were typically only to be used for particular classes. We found that this had the advantage that the method could then make the most of the choices necessary in the combination process (choices that the user would otherwise have to make), and also work efficiently for its intended class of programs. The methods were able to do this using the knowledge contained in the program history. Furthermore, the program history itself could be used to classify the program and so select the correct method.

The final system we discuss in this thesis was obtained from the initial system described above by making two main improvements: We refined the classification system and also developed and included new methods that were tailored to make effective use of the program history, (whilst also ensuring that we could still use the program history to select the correct method).

In some cases, when the user had underspecified the requirements of the combined program, the classification of programs (using the high-level description in the program history) implicitly allowed the method to make an informed estimate of what the users likely intentions were, see the example presented later in Section 6.6.1. The methods were also able to combine programs with different flows of control, and for a common class of programs with arithmetic operations the system can deduce which arithmetic laws can be applied in order to get a more optimal combined program. More detailed description of these methods can be found in Section 7.1 and Section 6.10.

Thus, our idealised program construction system shown in Figure 1.1 consists of a Prolog *techniques editor* and a *composition system*. The main modules of the system are represented in squares, the rounded squares are data and the lines denote input/output information. Descriptions of each component of the program construction are as follows:

[1]This basis for the classification scheme was not chosen randomly, but was based on our studies in which we found that knowledge of the flow of control was important in guiding the combination process.

The *techniques editor* has not been built because such systems already exist [Robertson 91, Bowles 93], and could easily be modified to record the choices that the user makes when building a program. In these editors, such choices include selecting the initial control flow (skeleton) and the techniques (standard Prolog practices). These choices are stored in the program history, which therefore contains a high-level description of the program that is close to the way that an expert might think about the program (simply because the editors are designed to use methods that experts might use themselves, but allow users to use them easily).

The *composition system* allows users to construct more complex programs by combining *simpler programs* which have been built by means of a techniques editor as described above. A component of this system is the *selection procedure* which automatically selects a combination method according to the program histories of the simpler programs. The selected method then controls the application of transformation rules (taken from a library), making its decisions according to the information it finds in the program history. We have designed the methods so that when they meet an underspecified combination, then they will tend to do transformations that preserve the spirit of the program histories (and hence their functionalities) rather than the simplest logical alternative based on unfold/fold operations.

In this way the composition system produces as output the combined program and a new history for the combined program. These can be stored and used as input for further combinations.

The main characteristics of our composition system are:

- The system decides the combining method by analysing the pair of programs to be combined (using the program history).
- The user does not need to take major decisions in the composition system such as which clauses need to be unfolded or folded (i.e. the user does not need to know about program transformation).

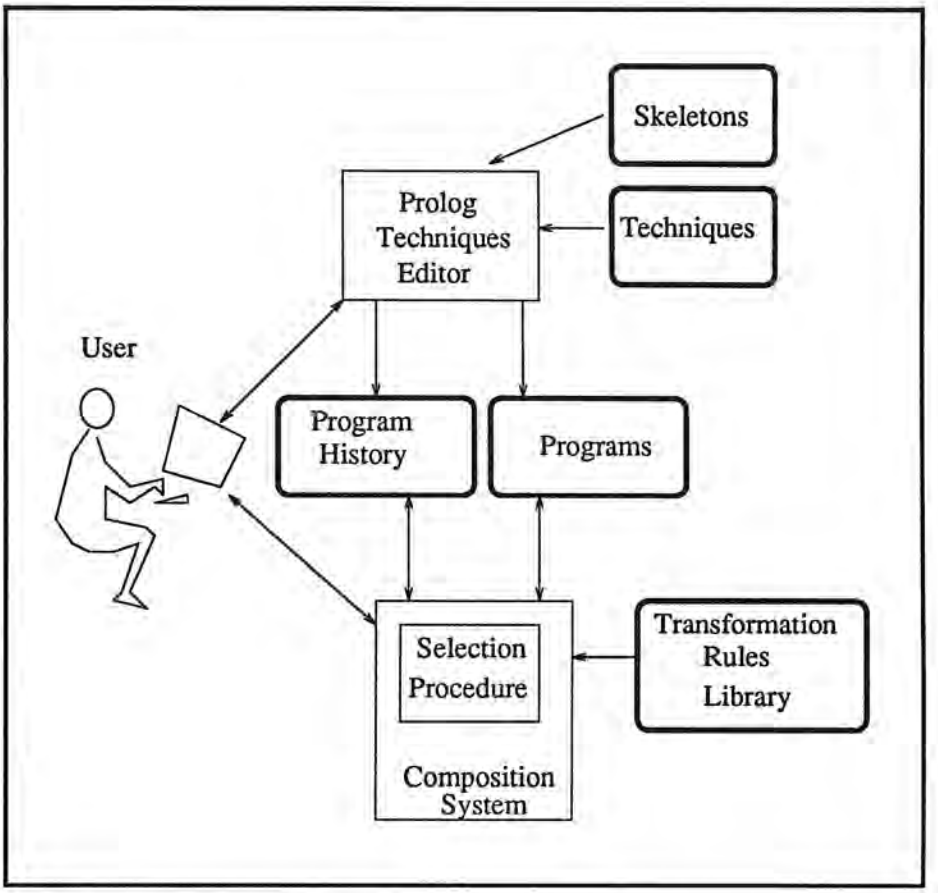

Figure 1.1: A Program Construction Environment

Also we discuss some properties concerning the combination problem. These properties guarantee the soundness of the composition process. We define three different groups of properties which need to be guaranteed.

The first group of properties discuss the equivalence of programs for each of our methods highlighting when the method does not preserve the equivalence (with respect to the join specification if we regard it simply as a Prolog program) but the program is correct according with the intentions of the user. We define a method for each class of program. Each method implemented in the composition system can be seen as a sequence of transformation operations. These methods are implemented based on: unfolding, folding, laws which are valid in an arithmetic domain and meta-folding (which is a modified version of the

fold operation which uses knowledge from the program history). Our composition system uses only these transformation operations and so it is possible to guarantee the soundness of the combined program obtained by each of our combining methods under constraints assumed by each method.

The properties in the second group are called *extension* properties. The relation ***extension*** has the following properties:anti-reflexivity, anti-symmetry and transitivity. These are used in the selection of the combining method.

The third group defines properties (called *composition* properties) that will be useful at the user level when he defines the join specification, for instance commutativity.

1.1 Layout of the Thesis

The layout of the remaining chapters of this thesis is as follows:

Chapter 2 gives a set of definitions used in the different approaches of editing systems with the last part of the chapter presenting the definitions from the program transformation field.

Chapter 3 gives an overview of related work in editing systems which were important in the development of this thesis because they introduce the foundational notions of program schemata and techniques. The main work covered is that of editors based on the program schemata notion such as those of Bundy, Gegg-Harrison, Barker-Plummer and Deville and finally editors based on the skeleton notion such as Kirschenbaum, Robertson and Bowles.

Chapter 4 provides a description of the composition problem in procedural, functional and logic languages. The functional and logic approaches are based on program transformation. Therefore these approaches were useful in the implementation of our composition system which relies on transformation operations and knowledge about the program.

Chapter 5 presents a hierarchy of standard patterns of flow of control which will be used for classifying programs and for the program history. It also describes our set of com-

bining methods and a module called the *selection procedure* which automatically selects a combination method from our set of methods. This selection is performed by taking into consideration features of the initial programs as described on the program history. Such features include flow of control plus techniques information.

Chapter 6 presents a set of methods for combining programs with the same flow of control.

Chapter 7 gives a set of methods for combining programs with different flows of control.

Chapter 8 describes the composition system and its interactive use at each stage of the composition process.

Chapter 9 gives some properties of our composition system. It includes three groups of properties: correctness properties which hold after applying each composition method; properties for the type of program which is obtained at each stage of the combining process and properties for the definition of the join specification.

Chapter 10 summarises our achievements, states current restrictions, gives areas for further study, and presents the contributions of this thesis.

1.2 Conclusions

The basic results of this investigation were:

- Classification of programs in terms of control flows and techniques was a useful classification in that it allowed us to develop program synthesis methods for each class in turn that were far more powerful, and required less user direction, that was possible for programs in general. This is not so surprising, after all, such classifications arose out of experts trying to reason about Prolog programs, and they would try to find useful reasoning methods.

- Storing the program development history is useful. It acts as a set of "meta-information" to the code. In this case this meta-information contained the infor-

mation necessary to classify the program, and hence select the correct combination method. This meta-information could also be used by the combination methods to further reduce the need for user interaction. In particular, the meta-information could be used to infer the user's intentions in requesting the program combination, and so deal with cases of underspecification. So the use of the program history was a powerful way to render program combination more effective.

2

Basic Definitions

In this section we define the terminology which will be used throughout this thesis. The first part provides the definitions which will be used in the description of the techniques editing systems and the second part of this chapter includes the terminology used in the field of program transformation.

2.1 Terminology used in Editing Systems

A *program schema* formalises the control structure of programs independently of the domains involved in the computations [Courcelle 90]. A schema is the abstract representation of a class of programs i.e. it represents a family of similar programs. A schema is obtained from the generalisation of programs, and programs can be obtained from a schema by instantiating its parameters (specialisation). In order to build a program for an application then the user will need to find an instance of a schema that is similar to the application [Partsch & Steinbruggen 83]. Gegg-Harrison presented a set of Prolog schemata as the basic constructs for structured Prolog recursive list processing [Gegg-Harrison 89]. An example of Gegg-Harrison's schemata is the `schema_A`, shown below, which processes a list until the empty list is reached.

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

In the schema, optional arguments and subgoals are denoted by enclosing them within angle brackets and an arbitrary number of arguments is denoted by enclosing them with double angle brackets. Therefore any number of arguments can be replaced in the schema variables `&1,...,&7` while the terms `pre_pred( ≪ &3 ≫,H,≪ &4 ≫)` and `post_pred( ≪ &6 ≫,H,≪ &7 ≫)` may be instantiated to a null element or to any subgoal.

For example we can obtain the program `reverse/2` by applying the substitutions {`reverse/schema_A, L/&1, L/&2, M/&5, M/&6, L/&7`} and instantiating `pre_pred` to null element and `post_pred` to append. This resulting program is normally known as 'naive reverse' because it is quite inefficient. A set of schemata for list processing are described in Appendix A.

```
reverse([],L).
reverse([H|T],L) :- reverse(T,M), enqueue(M,H,L).

enqueue([],E,[E]).
enqueue([H|T1],E,[H|T2]) :- enqueue(T1,E,T2).
```

Skeletons are basic control flow constructs [Kirschenbaum *et al.* 89]. There is a clear difference between a program schema defined above and a skeleton. A skeleton only provides control and does not contain arbitrary numbers of arguments, as in the schema. A skeleton is restricted to one argument which is used to recurse up or down the data structure. In this thesis we will consider skeletons as a special type of program schema.

An example of a skeleton is to traverse a list, where a list is deconstructed by removing head elements until the empty list is reached. The definition of this skeleton is shown below. The predicate name used in the notation is a meta-symbol which can be instantiated to any predicate name and each test *ci* for $i \in \{1,\ldots,n\}$ can be instantiated to any predicate name. This skeleton allows for the general case when the elements of the list need to be distinguished in 'n' different ways.

```
traverse_n([H|T]) :- c1(H), traverse_n(T).
traverse_n([H|T]) :- c2(H), traverse_n(T).
                  ⋮
traverse_n([H|T]) :- cn(H), traverse_n(T).
traverse_n([]).
```

Techniques are defined as programming practices which involve computing arguments and carrying contexts [Sterling & Kirschenbaum 91]. An example of a technique is *count* which always adds one to the value returned from the recursive call. This technique as defined below can only be applied if the data structure which is traversed by the skeleton is a list. The effect of this technique when it is applied to the *traverse_n* skeleton is to add an extra argument (in the skeleton) for computing the number of elements of a list plus an extra arithmetic subgoal in the body of each recursive clause to relate the computation from the body with the final result in the head of the clause. Techniques behave differently for different skeletons, for instance counting the number of elements of a list requires adding a different subgoal (in the body of a clause) compared with counting the number of nodes in a binary tree. A description of a set of techniques is given in Appendix D.

```
traverse_n_count[H|T],N) :- c1(H), traverse_n_count(T,N1), N is N1+1.
traverse_n_count([H|T],N) :- c2(H), traverse_n_count(T,N1), N is N1+1.
                  ⋮
traverse_n_count([H|T],N) :- cn(H), traverse_n_count(T,N1), N is N1+1.
traverse_n_count([],0).
```

An *enhancement* is a program derived from a skeleton by consistently applying the following modifications: adding arguments, adding goals and adding clauses by means of techniques. Sterling and Lakhotia classify enhancements as being of three types: modulations, extensions or mutations [Sterling & Lakhotia 88].

1. *Modulation.* Modulation can be briefly described as follows: when developing a program we often need to change only one part of a program. In order to change that part of the program a distinction must be made between it and the rest of the program. This distinction is made by creating a new procedure from that section of code. The inverse process of taking a procedure and incorporating it directly into

the code is referred to as making the procedure inline. The two processes of creating a new procedure from an existing section of code or conversely making a procedure inline are collectively referred to as modulation.

A program $\mathcal{A}$ is a modulant program of $\mathcal{B}$ and vice-versa, if they can be derived from one another by a sequence of unfold and abstract transformations (data abstraction). For instance, the program shown below can be abstracted to form a new procedure.

```
ancestor(Ancestor,Descendant) :- parent(Ancestor,Descendant).
ancestor(Ancestor,Descendant) :- parent(Ancestor,Person),
            ancestor(Person,Descendant).
parent(luis,david).
parent(sonia,mario).
            :
```

The program `ancestor/2` succeeds if the first argument (`Ancestor`) is an ancestor of the second argument (`Descendant`).

The abstraction process replaces a sequence of subgoals that occur one or more times in the bodies of the clauses (ie. `parent`) of a given program with a new predicate (`parent_rel`), and adds a clause whose head is the new predicate and whose body is the appropriate sequence of subgoals.

```
ancestor(Ancestor,Descendant) :- parent_rel(Ancestor,Descendant).
parent_rel(Ancestor,Descendant) :- parent(Ancestor,Descendant).
parent_rel(Ancestor,Descendant) :- parent(Ancestor,Person),
            parent_rel(Person,Descendant).
```

If we then unfold (see page 16 for details) the definition of `parent_rel/2` in clause number 1, we get back to the initial definition of `ancestor` shown above.

2. *Extension.* Extensions are the result of applying techniques to skeletons. An extension is a special type of enhancement restricted to adding computations which do not affect the control flow of the skeleton. A program $\mathcal{E}$ is an extension of program $\mathcal{P}$ if $\mathcal{E}$ is created by introducing extra computation around the control flow provided by $\mathcal{P}$. The type of operations which can be performed are:

   - Renaming the predicate.

- Adding arguments.
- Reordering the arguments.
- Adding some extra subgoals to the bodies of the clauses which do not change the control flow. (This is guaranteed as the subgoals are added only using those techniques which do not change the flow of control).

In this thesis we are considering extensions in which no reordering of the argument used for traversing the data structure is allowed (i.e. the argument used for traversing the data structure always will be in the first position). The reason for this restriction is that we have recorded in our program history information related the data structure used in each program and we have assumed that the data structure is always in the first position.

A formal definition of extension is given in Chapter 9. An example extension could be to apply the technique *add_carrier* to the *traverse* skeleton which is defined below, where the technique *add_carrier* adds some argument through all clauses in the recursive subgoals (also defined in Appendix D).

```
traverse([X|Xs]):-                          traverse([X|Xs],A) :-
   c_1(X), traverse(Xs).                       c_1(X), traverse(Xs,A).
traverse([X|Xs]) :-                         traverse([X|Xs],A) :-
   c_2(X), traverse(Xs).      add_carrier      c_2(X), traverse(Xs,A).
      .                          ⟹                .
      .                                           .
      .                                           .
traverse([]).                               traverse([],A).
```

3. *Mutation.* A procedure $\mathcal{R}$ is a mutant of procedure $\mathcal{P}$ if $\mathcal{R}$ is derived from $\mathcal{P}$ by performing the following operations:

   - adding one or more clauses which process conditions not checked by the initial program. This new clause should not change the structure of the skeleton (ie. the base case cannot be changed drastically). The new clause should have one data structure pattern that is already defined on the program. In this way the interpretation of the original skeleton is preserved but allows additional behaviour.

- adding subgoals which change the control flow and may conditionally terminate a clause.

For example, a mutation can be produced by adding an extra clause to the piece of code `max/3` shown below. This extra clause is called a mutant clause.

```
max(A,B,A) :- A > B.
max(A,B,B) :- A < B.
```

The extra clause added to program `max/3` checks the maximum when A is equal B. The resulting piece of code is a mutant of the program shown above.

```
max(A,B,A) :- A > B.
max(A,B,B) :- A = B.
max(A,B,B) :- A < B.
```

The *stepwise enhancement* methodology suggests the idea of developing a program by first finding the suitable basic control flow (*'skeleton'*). Once the skeleton has been determined, extra computations are included by applying appropriate *techniques* to produce an *extension* [Kirschenbaum *et al.* 89]. Then extensions can be combined to produce the desired program. The extensions can be regarded as another skeleton, permitting us to repeat the process until the final program has been completed.

2.2 Terminology used in Program Transformation

The *composition* process can be defined as the merging of two programs to form a new program which should have the capabilities of each original program [Berzins 86].

A *definite program clause* is a clause of the form $A :- B_1, \ldots, B_n$ which contains precisely one atomic formula (A) in its consequent. A is called the head and $B_1, \ldots, B_n$ are called the body of the program clause [Lloyd 87].

A *definite program* is a finite set of definite program clauses [Lloyd 87].

A *join specification* is a high level description in which we define the characteristics of a new program that can be generated using two other programs. In the join specification

we define the number of parameters in the new procedure, data input vector of variables from each program, and output vector of variables. Note that this join specification is not a Prolog program. So it is valid to define properties relating the new program such as input/output vector of variables. These input/output variables allow information to be carried between each of the programs to be combined.

A join specification is defined as follows:

Let T be defined as $T \Leftarrow P, Q$ where P and Q are predicates that will be joined; they are called join operands and T is the join target (the resulting join procedure). An example of the join specification is shown as follows:

```
append_len(L1,L2,L3,N) ⇐ append(L1,L2,L3), len(L3,N).
```

where the join target is `append_len(L1,L2,L3,N)`, the first join operand is `append(L1,L2,L3)` and the second join operand is `len(L3,N)`.

In this thesis we distinguish between a *user specification* and *join specification.* A *user specification* is the formal definition of the program that the user wants to build whereas a *join specification* is the actual specification which is used by the composition system for combining programs. The *user specification* can have two operands plus extra subgoals and can be defined as $T \Leftarrow P, Q, F_1, \ldots, F_n$ where P and Q are join operands, T is the join target and $F_1, \ldots, F_n$ are subgoals which use the output results from predicate P and Q for performing new computations. An example of a user specification is defined below.

```
average(L,Av) ⇐ sum(L,Sum),count(L,Count), \+ Count=0, Av is Sum/Count.
```

In the above user specification `average(L,Av)` is the join target, `sum(L,Sum)` is the first operand, `count(L,Count)` is the second operand, $F_1 =$ `\+ Count=0` and $F_2 =$ `Av` *is* `Sum/Count`.

The *meaning* of the program P (denoted by M(P)) is defined as the set goals which can be proved by using the axioms defined in program P i.e. the meaning of the program P

can be defined as follows: M(P) = {G | G is a ground goal such that G is provable in P} [Tamaki & Sato 83, Tamaki & Sato 84].

2.2.1 Transformation Operations

The operation *goal merge* produces from a program P a new program P' in which the identical subgoals are replaced by a single occurrence [Tamaki & Sato 84].

$$P' = (P - \{C\}) \cup \{C'\}$$

where C is a clause in P whose body contains more than one syntactically identical subgoal (exactly the same variables), and C' is the result of merging these occurrences [Tamaki & Sato 84].

The *unfolding* operation replaces a call A in the body of a clause with the body of another clause whose head is unifiable with A [Bossi *et al.* 90]. The *unfolding* operation can be defined as follows: let P be a program and C a clause in P of the form $A \leftarrow Q_1, B, Q_2$ where A and B are atomic formulae and Q_1 and Q_2 are conjunctions of literals. Let $H_1 \leftarrow R_1, \ldots, H_n \leftarrow R_n$ be those clauses in P whose heads H_i unify with B with the mgu's $\theta_1, \ldots, \theta_n$ (the most general unifiers). Unfolding B on C we obtain the clauses $(A \leftarrow Q_1, R_1, Q_2)\theta_1, \ldots, (A \leftarrow Q_1 R_n, Q_2)\theta_n$. By replacing C by these clauses we transform the program P into a new program [Fuchs & Fromherz 91].

For example, we may want to calculate the length of two lists using predicate `append/3` and predicate `len/2` defined below. The predicate `append/3` concatenates two lists and the predicate `len/2` computes the length of a list.

P=C1,C2,C3,C4

C_1 : `append([],Y,Y).`
C_2 : `append([H|T],Y,[H|Z]) :- append(T,Y,Z).`
C_3 : `len([],0).`
C_4 : `len([H|T],Len) :- len(T,Laux), Len is Laux+1.`

The new predicate `append_len/3` can be defined using the following join specification:

C_5 : `append_len(X,Y,N)` $\Leftarrow$ `append(X,Y,Z), len(Z,N).`

The call `append/3` in the body of clause C_5 can be unfolded using the clauses C_1 and C_2, leading to the following clauses C_6 and C_7 as listed below.

```
C6 : append_len([],Y,N) :- len(Y,N).
C7 : append_len([H|T],Y,N) :- append(T,Y,Z), len([H|Z],N).
```

This process can be continued by unfolding `len` in clause C_7 to produce C_8.

```
C8 : append_len([H|T],Y,N) :- append(T,Y,Z), len(Z,Len), N is Len+1.
```

Note that the unfolding process can be continued by unfolding `len` in clause C_6. However, the purpose of this example is only to illustrate how the unfold and fold operation works, and so we have not presented all the stages for obtaining the program `append_len/3`.

The *folding* operation replaces goals Gs in the body of a clause with the head of another clause whose body is unifiable with Gs [Bossi *et al.* 90]. The folding operation can be defined as the inverse operation of the unfolding operation as described in Burstall's work [Burstall & Darlington 77].

The *folding* operation is defined as follows: let P be a program and D is a set of join specifications. Let C be a clause in P of the form $A \leftarrow P, Q\theta, R$ where P, Q and R are conjunctions of literals and θ is a substitution. Let C_1 be a clause of the form $H \leftarrow Q$ in D which is not a instance of C. Folding C using C_1 generates the clause C_2 of the form $A \leftarrow P, H\theta, R$ provided that unfolding C_2 on $H\theta$ with respect to D gives C; C_1 is the only clause in D whose head unifies with $Q\theta$ and θ maps variables which appear in Q, but not in H into distinct variables which do not occur in C_2. Replacing C by C_2 transforms P into a new program [Fuchs & Fromherz 91].

From the previous example the body of C_8 can be folded by using C_5. Therefore we obtain C_9, which is defined as follows:

C_9 : `append_len([H|T],Y,N) :- append_len(T,Y,Len), N is Len + 1.`

We shall see in Section 6.10 that arithmetic rules provide useful transformations in the process of simplifying expressions in the domain of real numbers and natural numbers. These rules are, for instance, identity element for addition, identity element for multiply, commutativity and associativity in real or natural numbers. The set of transformation rules defined in our system is shown as follows. In these rules A,B,C are any real or natural numbers.

$A * 0 = 0$ $\qquad$ $0 * 0 = 0$

$0 + 0 = 0$ $\qquad$ $1 * A = A$

$0 + A = A$ $\qquad$ $A * 1 = A$

$A + 0 = A$ $\qquad$ $1 * 1 = 1$

$0 * A = 0$

The commutativity and associativity rules are defined as follows:

$A + B = B + A$

$A + (B + C) = (A + B) + C$

3

Computer-aided Construction of Logic Programs

In this chapter we describe several editing systems and try to determine which is the most suitable techniques editor for providing us with the information that we require for our composition system. We have divided the editors into two classes: editors based on the notion of skeletons and techniques and editors based on the idea of program schemata. Figure 3.1 shows the classification of editors. In this Figure we have classified the editors into those based on techniques and skeletons (Kirschenbaum, Robertson and Bowles). On the other hand we have the editors based on program schemata (Bundy's editor, editors based on Gegg-Harrison schemata and on Barker-Plummer's cliché and also Deville's methodology which transforms each specification into a logic description (see Section 3.1.4)).

The common objective of each approach described in this chapter is to provide tools which help programmers in Prolog programming. These approaches are closely related to our work, because together they provide a general framework related to program schemata and also they help us in the identification of the kind of techniques editor that we require to be connected with our composition system.

Information about flow of control is widely used in the analysis and manipulation of logic programs. Recently, Fuchs shows that program schemata (which encode information about the flow of control and programming techniques) can be used successfully during the trans-

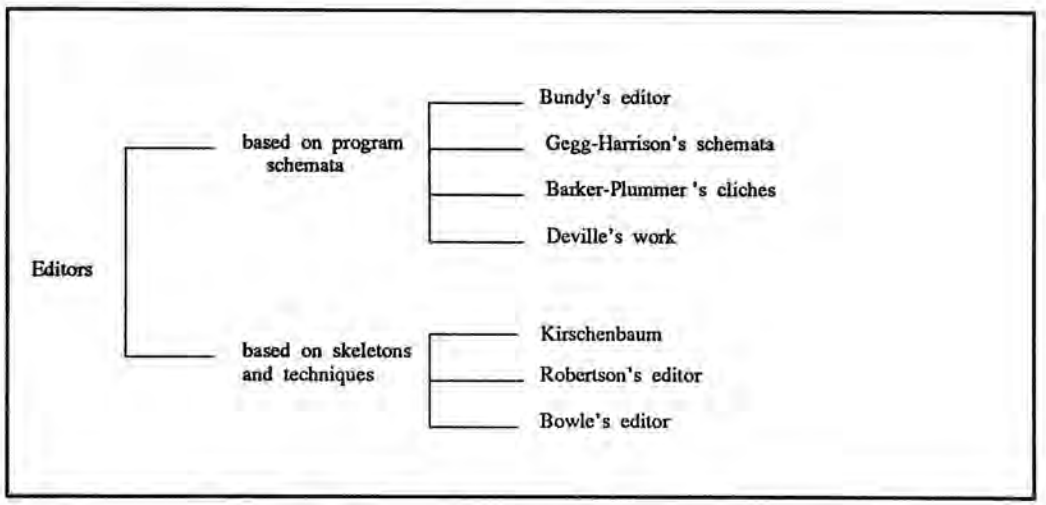

Figure 3.1: Classification of Editors

formation of a program [Fuchs & Fromherz 91]. Other work which shows how useful the information of the flow of control can be is a Prolog debugger implemented at Edinburgh University [Duncan 92]. This debugger enhances explanations of program behaviour by knowing information about the skeleton used in its construction.

The desired techniques editor should allow the user to build a program by using general flows of control which can be tailored to the wishes of the user. The main characteristics which are required from the techniques editor are:

- Information about the flow of control. This information relating to the flow of control used in the program should direct the combining process. Our combination approach, based on program history, takes advantage of the fact that each skeleton defines a different class of programs. Each program constructed by using a skeleton inherits the features of the skeleton. So by knowing which skeleton was used in the program the composition system can start taking decisions about the most suitable combining method for the pair of programs. However, note that the selection is not determined in a unique way by only using the flow of control.
- The information about techniques. This information is considered for optimisation

reasons. If the system can know which set of techniques were used in the program and also the flow of control then it can decide to apply a combining method which produce a more efficient combined program.

- Termination and well definedness of the program. Termination and well-definedness is required in order to guarantee that the combined program, which is the result of the transformation of an initial pair of programs, is well defined and will terminate. This property needs to be satisfied, otherwise ill-formed predicates might be combined.

3.1 Editors Based on Program Schemata

In this chapter we first present the editors based on program schemata, and then present the editors based on the notion of skeleton and techniques.

The different proposals for building programs based on the notion of program schemata do not provide the information that the combination system needs because they do not make a clear separation between the flow of control (skeletons) and the data flow (techniques). On the other hand the editors based on skeletons and techniques are more suitable for our approach to the combination problem. These approaches make a distinction between which is flow of control and data flow.

3.1.1 Bundy's Editor

A study of principal bugs in a Prolog environment [Brna *et al.* 87] has shown that they arise frequently in the creation of recursive procedures. From this fact arose the idea to develop an editor which assists programmers to write programs by combining the partial solutions provided by:

- Structure editors, to ensure that the user writes syntactically correct programs.
- Cross referencers, which alert the user to calls to undefined procedures.

- Procedure stores, which allow reuse of old procedures.
- Formal methods, which ensure only correct programs are written.

Bundy describes an implementation of an editor which ensures not only the correctness of the syntax of the procedures but also the correct use of recursion. In this context *correct* means that the recursive procedure is guaranteed to terminate and to be well defined. By *well defined* Bundy means that the procedures are not under-defined or over-defined [Bundy 88, Bundy *et al.* 91]. The definitions of under-defined and over-defined will be presented later in this section. The editor was designed with the purpose of forcing well definedness and termination of procedures because non-terminating procedures and under or over-defined procedures are a major source of Prolog bugs. Also, the recursion editor provides guidelines in the production of new procedures by allowing the user to use existing procedures or procedure schemata.

The definition of over and under-defined is presented as follows:

- An over-defined procedure consists of an inconsistent set of definitions. One example of an over-defined procedure is the *greatest common divisor* of two integers X and Y denoted as $gcd(X,Y)$. This is defined as follows:

  - $gcd(X,X) = X$
  - $gcd(X,Y) = gcd(X-Y,Y)$

  For example when $X = 2$ we obtain from the first definition of *gcd* that gcd(2,2) = 2 and by using the second definition we get that gcd(2,2)= gcd(0,2). Therefore this definition of *gcd* is over-defined because it provides two results for the same combination of inputs.

- A procedure is under-defined if a legal combination of inputs/outputs has not been considered in its definition. For instance, if we redefine $gcd/2$ as shown below. The new definition of $gcd/2$ has the property of under-definedness for the case $X = Y$.

$$X > Y \ \ then \ \ gcd(X,Y) = gcd(X-Y,Y)$$

$$X < Y \ \ then \ \ gcd(X,Y) = gcd(Y,X)$$

The recursion editor is not capable of defining all terminating and well-defined procedures. The basic design of the editor limits it to procedures that can be shown to terminate by induction on the recursive structure of their arguments. A predicate is defined by structural induction: induction because it is defined recursively, structural because it is controlled by the structure of its argument rather than by numeric values. Non-terminating and over-defined procedures are guaranteed not to be generated by the recursion editor.

The user is presented with a basic schema or previously generated predicate and a menu of editing commands. The user uses these commands to edit the schema or the old predicate into the new predicate. Each time a command is selected and executed the schema will be altered in a way that continues to guarantee its termination and well-definedness. The recursion editor provides two basic schemata: non-recursive and recursive.

The non-recursive schema is defined as follows:

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

where α is the procedure being defined and β is a previous defined predicate. Φ is a parameter and $\rightleftharpoons$ is an equivalence relation.

For the recursive schema the schema for primitive recursion described in Péter's book [Péter 81] was used but was generalised to deal with data structures found in programming. The primitive recursive schema is defined as follows:

$$\mu(\flat, \Phi) \ \rightleftharpoons \ \nu(\Phi)$$
$$\mu(\sharp(\Psi,\Theta),\Phi) \ \rightleftharpoons \ \xi(\Psi,\Theta,\Phi,\mu(\Theta,\Phi))$$

where:

- μ is the recursive procedure being defined;
- ν *and* ξ are the procedures it is being defined in terms of:
- Θ is the recursion variable;
- Φ is the parameter;
- $\flat$ and $\sharp$ are the constructor functions;
- Ψ is the constructor parameter;
- $\rightleftharpoons$ is the equivalence relation.

This schema incorporates the following generalisations to work in Péter's book.

- The data structure of natural numbers has been generalised to one in which the base constructor is $\flat$ rather than 0 and the step constructor or recursive constructor is $\sharp$ rather than successor(s). This recursive schema $\sharp$ has two arguments: Θ is the standard recursion variable and Ψ is an additional parameter to represent constructors such as $[\Psi|\Theta]$ that take non-recursive arguments. Ψ also needs to be a parameter of ξ.
- The relation $=$ has been generalised to any equivalence relation and is represented by the symbol $\rightleftharpoons$.

The editor described by Bundy can be used for building recursive and non-recursive Prolog procedures. This is done in two steps. Firstly pure logic definitions are built using the editing commands and, secondly, these are compiled into Prolog. For example consider the definition of `remove_occurrences/3`. This predicate removes all appearances of a specific item X from the head of a list, for example

`remove_occurrences(1,[1,1,1,1,5,8,10],[5,8,10])` where the first argument is the item that will be removed and the third argument is the remainder of the list without all the initial occurrences of the first argument. This example will be used in section 3.2.1. The first stage is to start with the primitive recursive schema.

$$\mu(\flat, \Phi) \rightleftharpoons \nu(\Phi)$$
$$\mu(\sharp(\Psi, \Theta), \Phi) \rightleftharpoons \xi(\Psi, \Theta, \Phi, \mu(\Theta, \Phi))$$

The second stage is to instantiate the primitive recursive schema with the following substitutions: $\{remove\_occurrences \ / \ \mu, \ [] \ / \ \flat, \ [\_|\_] \ / \ \sharp, \ H \ / \ \Psi, \ T \ / \ \Theta, \ X \ / \ \Phi \ \}$

we then obtain:

$$remove\_occurrences([], X) \rightleftharpoons \nu(X)$$
$$remove\_occurrences([H|T], X) \rightleftharpoons \xi(H, T, X, remove\_occurrences(T, X))$$

remove_ocurrences needs three arguments, so the user should select the command *add recursion argument*. This command adds two new definitions for the new argument. Therefore after the use of the command we obtain:

$$remove\_ocurrences([], X, \flat') \rightleftharpoons \nu_b(X)$$
$$remove\_ocurrences([], X, \sharp'(\Psi', \Theta')) \rightleftharpoons \nu_s(X, \Psi', \Theta', remove\_ocurrences([], X, \Theta')$$
$$remove\_ocurrences([H|T], X, \flat') \rightleftharpoons \xi_b(H, T, X, remove\_ocurrences(T, X, \flat'))$$
$$remove\_ocurrences([H|T], X, \sharp'(\Psi', \Theta')) \rightleftharpoons$$
$$\xi_s(H, T, X, \Psi', \Theta', remove\_ocurrences(T, X, \sharp'(\Psi', \Theta')), remove\_ocurrences([H|T], X, \Theta'),$$
$$remove\_ocurrences(T, X, \Theta'))$$

Renaming the parameters

$\{ [] \ / \ \flat', \ [\_|\_] \ / \ \sharp', \ H \ / \ \Psi', \ T \ / \ \Theta', \ \}$ and $\{ Ys \ / \ \flat' \}$ for the third case.

we get the following definition:

$remove\_ocurrences([], X, []) \rightleftharpoons \nu_b(X)$

$remove\_ocurrences([], X, [H|T]) \rightleftharpoons \nu_s(X, H, T, remove\_ocurrences([], X, T))$

$remove\_ocurrences([H|T], X, Ys) \rightleftharpoons \xi_b(H, T, X, remove\_ocurrences(T, X, Ys))$

$remove\_ocurrences([H|T], X, [H|T]) \rightleftharpoons$

$\xi_s(H, T, X, H, T, remove\_ocurrences(T, X, [H|T]), remove\_ocurrences([H|T], X, T),$
$remove\_ocurrences(T, X, T))$

By removing arguments in the position indicated in the recursive schema, renaming and inserting a new procedure in the body of the procedure which has been defined we get the following definition:

$remove\_ocurrences([], X, []) \rightleftharpoons true$

$remove\_ocurrences([], X, [H|T]) \rightleftharpoons false$

$remove\_ocurrences([H|T], X, Ys) \rightleftharpoons X = H, remove\_ocurrences(T, X, Ys)$

$remove\_ocurrences([H|T], X, [H|T]) \rightleftharpoons X \neq H$

Reordering the arguments, we have:

$remove\_ocurrences(X, [], []) \rightleftharpoons true.$

$remove\_ocurrences(X, [], [H|T]) \rightleftharpoons false.$

$remove\_ocurrences(X, [H|T], Ys) \rightleftharpoons X = H, remove\_ocurrences(X, T, Ys)$

$remove\_ocurrences(X, [H|T], [H|T]) \rightleftharpoons X \neq H$

In short, this editor provides two schemata for construction of recursive and non-recursive programs. This editor cannot supply information about which skeleton (control flow) and techniques were used in the construction of the program because it does not distinguish between flow of control and techniques during the program development. The use of this editor requires considerable expertise on the part of the user; for instance in deciding which parts in the schema need to be removed, where to insert a function, etc.

In conclusion we can say that our composition system requires more than the information that this editor can provide as *program history*. However, this editor guarantees termination and well definedness of the initial pair of programs, which is another required characteristic in our composition system.

3.1.2 Gegg-Harrison's Schemata

Gegg-Harrison attempts to provide the novice programmer with a set of standard program schemata for the construction of Prolog programs in the list processing domain. He defines a language and groups together a set of programs that share common syntactic features into a single *schema*. Such classes of programs exemplify general programming techniques. For example, the method of recursively processing all elements of a list called schema_A (defined in Chapter 2) is a programming technique. Note that in Sterling's terminology schema_A is not a programming technique.

In his classification Gegg-Harrison defines fourteen basic high level schemata that capture the majority of recursive list processing Prolog programs [Gegg-Harrison 91]. The definition of this set of schemata is given in Appendix A. An example of a recursive list processing problem is to move a specified element from one place to another place in a list. Gegg-Harrison also suggests that it could be possible to incorporate this hierarchical notion into a Prolog tutoring system which is currently under development. A set of examples in which we use these schemata for the construction of a program is also shown in Appendix A.

We believe that an editor based on Gegg-Harrison's schemata cannot be used in our environment because, like Bundy's editor, there is no separation between control flow and techniques.

3.1.3 Barker-Plummer's System

Barker-Plummer defines a tool which allows programmers to define procedures by analogy with existing generalised procedures called clichés [Barker-Plummer 90]. This generalised procedure is a high level definition that includes control flow and arguments. The method of programming using clichés allows programmers to write commonly occurring program forms just once, but to reuse them in a variety of ways as they are needed in the program. The instantiation of meta-symbols in a cliché varies from one use of the cliché to another. The following piece of code has been taken from [Barker-Plummer 90] as an example of a cliché. It traverses a list checking when the elements of the list meet the condition defined in predicate Q/n. This example is written using the cliché language proposed by Barker-Plummer.

```
$P/n :- universal($Q/n).
$P([],&Aux).
$P([H|T],&Aux) :- $Q(H,&Aux), $P(T,&Aux).
$end_cliche$
```

The first line of the cliché definition is called the header. The header records the name of the cliché. The name *universal* indicates the name of the main procedure has been defined, where n is the arity of the predicate. Symbols prefixed by $ are *cliché parameters.* These names can be instantiated to any predicate name. The cliché terminator is denoted by `$end_cliche$`. An application of this cliché would be to define the predicate `all_primes/1` which succeeds if the argument is a list of terms satisfying the `prime/1` predicate. This procedure is obtained by instantiating `$Q` to the `prime/1` and `$P` to `all_prime/1`.

By writing the following instruction

```
all_primes/1 :- universal(prime/1).
```

the system will generate the following program by performing the suitable instantiations in the *universal* cliché.

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

The predicate `prime/1` succeeds when its argument is a prime.

This approach taken by Barker-Plumer avoids having to rewrite similar procedures. For example, if we wish to write a procedure which determines whether all members of a list are even numbers, then we would write basically the same procedure as for `all_primes/1` but with `all_even` instead of `all_primes` and replace the predicate `prime/1` with `even/1`.

In short, Barker-Plummer describes a tool which allows the construction of Prolog programs by instantiating existing generalised programs called clichés. These clichés are similar to Gegg-Harrison's schemata, but written in a different notation. This approach cannot provide a record of which skeleton and techniques were applied in the construction of the program. The reason is that the control flow and techniques applied in the program are indistinguishable.

3.1.4 Deville's Methodology

Deville presents a general schema for constructing logic programs [Deville 90] based on induction over the structure of the data objects manipulated by the program. In this methodology the development of a program is decomposed into several steps. The first stage is the elaboration of a specification of the problem. The second stage is the construction of a logic description in pure logic from the specification, independent of any programming language, and finally the third step deals with the derivation of a Prolog program from the logic description.

In the elaboration of a specification, no particular programming language is imposed; the specification of the problem is an informal description of a relation. At this stage the type information provided is the directionality (uses) for which the program has to be correct, and any side-effects are also specified. The second stage is the construction of what is called a *logic description* from the specification of the problem. A logic description is a formula in first order logic. This stage is also independent of the chosen programming language. The logic descriptions can be transformed using fold/unfold transformations. Finally the

last stage is the derivation of a logic program. This process starts with the translation of the logic description into program clauses. At this stage an abstract interpretation or data flow analysis (using type information and directionality) needs to be carried out to determine the order of literals and clauses. An independent termination proof also should be carried out. We consider the main phases of Deville's method in more detail below.

The Elaboration of a Specification

The information that is needed in this description is: type information, directionalities (uses) for which the program has to be correct and, if a side-effect needs to be specified, it will also be described in an extra part of the specification. For example, let us consider the predicate `del(X,L,Rest)` which removes the first occurrence of `X` from the list `L`, giving the list `Rest`. The elaboration for constructing of the predicate `del/3` is shown in Figure 3.2. In the specification, *type information* is explicitly stated in a relational and non-recursive form. The conditions for the *use* of the predicate `del/3` are described in the part called directionality. They specify the allowed forms of actual parameters before the execution of the procedure (*input*) and the form of the parameters after execution is also defined (*output*).

Construction of the Logic Description

The general form of a logic description is described below. This is a closed well-formed formula.

$(\forall X_1 \forall X_2 ... \forall X_n)(p(X_1, X_2, ..., X_n) \iff Def)$

with $n \geq 0$, *Def* is a formula and the symbol $\iff$ is a equivalence relation.

By convention the quantification $\forall X_i$ will be implicit. In the formula *Def*, free variables (different from X_i) are allowed, and they are assumed to be existentially quantified.

The previous definition can be rewritten as follows:

```
procedure del(X,L,Rest)

Type: X: term
      L,Rest: list

Relation
      X is an element of L and Rest is the remainder of L without the
      first occurrence of X in L.

Directionality
      input(ground,ground,ground)      output(ground,ground,ground)
      input(ground,ground,no_ground)   output(ground,ground,ground)
```

Figure 3.2: Specification of the predicate del/3

$$p(X_1, X_2, ..., X_n) \iff \begin{array}{l} C_1 \ \& \ F_1 \\ \vee \ C_2 \ \& \ F_2 \\ \quad . \\ \quad . \\ \quad . \\ \vee \ C_n \ \& \ F_n \end{array}$$

where F_i and C_i are formulae. All the free variables over the C_i and F_i not appearing in $p(\vec{x})$ are existentially quantified over the definition part (right-hand side of the equivalence) of the logic description. Each C_i & F_i deals with one of the various cases of the induction parameter. Therefore each C_i will determine a possible case of the induction parameter, while the corresponding F_i will verify that the relation holds in this particular case.

Following with the example `del/3` described in Figure 3.2, the logic description is constructed in several stages:

1. Choice of an induction parameter: Let L be the induction parameter.

2. Choice of a well-founded relation: $l_1 < l_2$ if only if l_1 is a proper suffix of l_2.

3. Structural forms of an induction parameter (L) are given in C_1 and C_2.

   - C_1: L=[]
   - C_2: L=[H | T]

4. Construction of the structural cases

   - For $L = []$ it is not possible to have the list *Rest*, which is the list L without the first occurrence of X. Therefore F1 is $false$.
   - For $L = [H|T]$ there are two possibilities depending on whether or not $H = X$.
     - For $H = X$ then $Rest = T$ because T is the list L without the first occurrence of X.
     - For $H \neq X$ then *Rest* must be of the form $[H|Temp]$ where $Temp$ is the list T without the first occurrence of X.

   We obtain F2:

   $(H = X \;\&\; Rest = T \;\vee\; H \neq X \;\&\; del(X, T, Temp) \;\&\; Rest = [H|Temp])$

According to the well-founded relation $T < L$ when L is any ground list since T is a proper suffix of L. Finally the constructed logic description is shown below.

$$
\begin{aligned}
del(X, L, Rest) \iff & \\
& L = [] \;\&\; false \\
\vee \;\; & L = [H|T] \;\&\; \quad (H = X \;\&\; Rest = T \\
& \qquad\qquad \vee H \neq X \;\&\; del(X, T, Temp) \;\&\; Rest = [H|Temp])
\end{aligned}
$$

where the variables H,T and $Temp$ are existentially quantified on the right hand side of this description.

Generation of the Logic Program

A logic program can be derived from the logic description. The logic description is firstly translated into program clauses. For our example, the resulting clauses are:

$del(X, L, Rest)$:- $L = [H|T]$, $H = X$, $Rest = T$.
$del(X, L, Rest)$:- $L = [H|T]$, $Rest = [H|Temp]$, $del(X, T, Temp)$, $not(H = X)$.

The previous Prolog procedure can then be transformed into a more efficient version where the parameter values that appear on the right-hand side have been substituted into the left-hand side.

$del(X, [X|T], T)$.
$del(X, [H|T], [H|Temp])$:- $del(X, T, Temp)$, $not(H = X)$.

This is a proposed methodology for the synthesis of programs which works using a single schema. Therefore it cannot provide the *program history* information that we require in our composition system. In order to get this specific information an extraction of the components (skeleton and the set of techniques) of the program is required.

3.2 Editors based on Skeletons and Techniques

In this section we present three approaches for building Prolog programs based in the idea of skeleton and techniques as defined originally by Kirschenbaum et. al [Kirschenbaum *et al.* 89]. Kirschenbaum et al. describe a methodology for building Prolog programs using a skeleton and applying a sequence of techniques. This proposal makes a separation between the flow of control and data flow. Therefore valuable information about features of the program can be obtained from an editor which records pertinent parts of the program development, for example, the initial control flow (*skeleton*) and the *techniques* that the user applied in the construction of the program. The structural knowledge about the program (obtained from an editor of this kind) might be difficult to extract from just the program itself. Usually this knowledge is discarded, but in this thesis we illustrate that in fact it can be very useful, or even vital, to retain it.

By using an editor of this kind the user starts the program construction process firstly by finding the *skeleton* (perhaps from a menu of standard skeletons). At this stage the techniques editor might record the control flow which was selected. Secondly, the user chooses

a *technique* which can be easily recorded. The composition system also uses information about how the initial subgoals in the skeleton were transformed. So for each clause in our skeleton, the techniques editor needs to record the transformed subgoals (predicate name and variable names). In the case where a new clause is added to the initial flow of control, this might be recorded in the list of mutant clauses in the program history. Also, this techniques editor must be provided with a set of editing commands to allow flexibility in the addition of new clauses or subgoals in programs already constructed, the renaming of predicate names and variables, and the reordering of arguments.

Robertson's work is an implementation of an editor based on the notion of skeletons and techniques as defined by Kirschenbaum et al. and finally Bowle's editor also is an editor based on the notion of skeleton and techniques with slightly different notion of techniques. We will conclude that an editor based on skeletons and techniques could provide the information that the composition system requires such as information about the flow of control and the data flow. Currently, none of the two Edinburgh implemented versions of the techniques editor [Robertson 91, Bowles 93] provide us with the knowledge required for our composition system. However, an extended version of the techniques editor implemented by Robertson could be used to provide the required knowledge about the program.

3.2.1 Kirschenbaum et al.'s Approach

Kirschenbaum et al. [Kirschenbaum *et al.* 89] present the idea of stepwise enhancement as a methodology to be used during program development to produce Prolog structured programs. This method consists of developing a program by finding the suitable basic control flow (skeleton). Once the skeleton has been determined, extra computations are included by applying appropriate techniques to produce an extension. Separate extensions can be combined to produce the desired program. The extensions can be regarded as another skeleton permitting us to repeat the process until the final program has been developed.

Sterling et al. [Kirschenbaum *et al.* 89, Sterling & Kirschenbaum 91] attempted to iden-

tify and classify skeletons and they found that slight changes in the specification of the program, such as changing the base case, leads in general to different skeletons. They claimed that this volatile nature of skeletons makes it difficult for programmers to build programs by using program schemata (in which skeleton and techniques are indistinguishable). So, they said that a better approach is to have programmers decide the control flow (skeleton) for their application and then using standard techniques, enhance the skeleton.

The most simple examples of skeletons which process lists are traverse, search, short_traverse. The first skeleton traverses the entire list always. The second will either traverse the entire list or stop when a condition is met and the third skeleton traverses the list until what is being searched for has been found, and fails otherwise. In Appendix C there is a set of skeletons defined in Kirschenbaum et al. [Kirschenbaum *et al.* 89]. This list does not mean to be complete but shows three important categories for building Prolog programs such as meta-interpreters, parsers and skeletons used for manipulating recursive data structures.

Also Brna et.al. [Brna *et al.* 91] have collected a set of programming clichés from textbooks and by interviewing Prolog experts and they called them *programming techniques.* However Kirschenbaum et al. differ in this classification because they consider these clichés of flow of control as skeletons.

The following example is taken from [Kirschenbaum *et al.* 89]. Suppose we want to obtain statistics such as the `mode` (most common element) and `mean` (arithmetic mean) of a list of numbers during just one pass through the list. The `mean` can be computed knowing the sum and the number of elements in the list which can be easily found using any method of traversal. To simplify the example, we will assume that the input list is sorted. This means the list can be easily broken down into sublists, with all elements in a given sublist being equal, and no sublists having any elements in common. Finding the mode is then just a matter of finding the sublist with maximum length. Furthermore, we can efficiently find the sum of all elements by summing the contributions for each sublist. The contribution from a sublist is just its length (which we already need for the mode calculation anyway)

multiplied by an element of the sublist.

To build the program, we firstly prepare the basic_skeleton which will traverse the list be effectively breaking it into the sublists. We then specialise this code in three distinct ways to calculate each of the `mode`, `sum` (sum of all elements), and `len` (number of elements) in the list. Finally we will combine the three resulting programs into a single program.

In order to prepare the basic skeleton we will build two intermediate skeletons: remove_occurrences/3 and remove_len/4 which are described as follows:

The code remove_occurrences removes a block of occurrences from the head of a list within a sequence. For instance, after executing `remove_occurrences(1,[1,1,1,1,2,3,4],X)` X will be instantiated to X=[2,3,4]. Note that the predicate `remove_occurrences/3` only removes a block of occurrences from the head of the list, otherwise the list is unchanged.

The procedure `remove_occurrences/3` can be constructed by using the `short_traverse` skeleton which will either traverse the entire list or stop when a condition is met (see Appendix C).

```
short_traverse(X,[]).
short_traverse(X,[X|Xs]) :-
            short_traverse(X,Xs).
short_traverse(X,[Y|Xs]) :-
            \+ X= Y.
```

Finally, we can get the definition of `remove_occurrences/3` by applying the *build* technique and by renaming the predicate. The *build* technique adds an extra argument to the defining predicate and adds an extra computation in the body to relate the object constructed from the body to the final object in the head of the clause.

```
remove_occurrences(X,[],[]).
remove_occurrences(X,[X|Xs],Ys) :-
            remove_occurrences(X,Xs,Ys).
remove_occurrences(X,[Y|Xs],[Y|Xs]) :-
            \+ X=Y.
```

Now recall that in order to calculate the `mode` we needed the length of the sublists, hence we extend skeleton remove_occurrences using the technique *count* to produce remove_len.

The predicate `remove_len/4` removes a given block of occurrences of an item, X, from the head of a list; returning the remainder of the list and the number of times X was found in the list. That is, the definition of `remove_len/4` is obtained by applying the technique *count* to the extension *remove_occurrences*

count(remove_occurrences) =

```
remove_len(X,[],[],0).
remove_len(X,[X|Xs],Ys,N) :-
            remove_len(X,Xs,Ys,N1),
            N is N1 + 1.
remove_len(X,[Y|Xs],[Y|Xs],0) :-
            \+ X=Y.
```

The above piece of code for `remove_len/4` will now be inserted into a skeleton in such a way that we process the list in chunks (i.e. in terms of the sublists discussed earlier). This gives

```
basic_skeleton([]).
basic_skeleton([X|Xs]) :-
            remove_len(X,[X|Xs],Ys,N),
            basic_skeleton(Ys).
```

Given the above `basic_skeleton` for traversing the list, we now extend it in three different ways and then as a last step we will combine these to get the final program.

The *mode* is obtained by keeping track of which sublist had the maximum length. This can be done by applying the technique, *calculate*, to the *basic_skeleton*. (Recall that the *calculate* technique adds an extra argument in the skeleton and an extra arithmetic subgoal to the body of each recursive clause to relate the calculation from the body to the final result in the head of the clause, see also Appendix D.)

mode = calculate(basic_skeleton)

```
mode([],0,Mode).
mode([X|Xs],Multiplicity,Mode) :-
            remove_len(X,[X|Xs],Ys,N),
            mode(Ys,Nultiplicity1,Mode1),
            max(Multiplicity1,Mode1,N,X,Multiplicity,Mode).

max(X,A,Y,B,X,A) :- X >= Y.
max(X,A,Y,B,Y,B) :- X < Y.
```

For example, given the query: mode([1,1,1,1,2,2,3,4],Multiplicity,Mode), Mode is instantiated to 1.

The sum of the elements of the list is obtained by adding the values of elements of the sublist `Ys` (the remainder of the list) plus the number obtained by multiplying `N*X`, where `N` is the number of times that `X` occurs in the list and `X` is the value of the element `X`. These results are calculated by the extension *summ*. The extension *summ* is obtained by applying the technique *calculate* to the *basic_skeleton*.

The definition of the extension *summ* is:

summ = calculate(basic_skeleton)

```
summ([],0).
summ([X|Xs],Sum) :-
            remove_len(X,[X|Xs],Ys,N),
            summ(Ys,Sum1),
            Sum is Sum1 + N*X.
```

The length of the list is obtained by calculating the length of the sublist `Ys` (the remainder of the list) plus the number of times that `X` was removed from the list. This is done by the extension *len* which is obtained by applying the technique *calculate*, to the *basic_skeleton*.

len = calculate(basic_skeleton)

```
len([],0).
len([X|Xs],Length) :-
            remove_len(X,[X|Xs],Ys,N),
            len(Ys,Length1),
            Length is Length1 + N.
```

These three extensions: *mode*, *summ* and *len* are independent of each other but share the same common *basic_skeleton*. This means that it is indeed possible to combine them and get the final program `mode_summ_len/5` which calculates the *mode*, the sum of the elements of the list and the length of the list:

```
mode_summ_len([],0,_Mode,0,0).
mode_summ_len([X|Xs],Multiplicity,Mode,Sum,Length) :-
            remove_len(X,[X|Xs],Ys,N),
            mode_summ_len(Ys,N1,Mode1,Sum1,Length1),
            max(N1,Mode1,N,X,Multiplicity,Mode),
            Sum is Sum1 + N*X,
            Length is Length1 + N.
```

This approach is suitable for our methodology for combining programs based on the *program history*. In Kirschenbaum et al.'s approach there is a clear separation between control flow and techniques and these appear explicitly as editing operations. Then, information about how the program was constructed can be recorded in our program history using an editor based on the notion of skeletons and techniques.

An editor based on this approach requires the selection of the control flow and a set of techniques for developing the program. This is not an easy task for Prolog beginners. In Bundy's approach we do not have this problem because the editor only provides two schemata (recursive and non-recursive schema). However, Bundy's approach requires a higher level of sophistication in controlling how these are used.

3.2.2 Robertson's Techniques Editor

Besides the editor by Bundy, the Edinburgh group has also been working in programming techniques trying to show how techniques can be useful for Prolog programming environments and in teaching Prolog programming skills. In this and the following section we only present the work related to the techniques editors. However in [Bowles *et al.* 94] there is a complete description of the work has been done to apply Prolog programming techniques to various stages of program development such as: in automatic analysis of the programs, in a Prolog tracer and in program transformation.

Robertson implemented a simple Prolog techniques editor with the purpose of helping novice programmers in initial programming tasks. The editor is based on the notion of skeletons and techniques described in [Kirschenbaum *et al.* 89]. The difference between the operation of Kirschenbaum's idea and Robertson's techniques editor is that in the

first system programs are constructed by applying techniques independently to a skeleton, obtaining several program components and finally merging these components together to obtain the final program. The second system works by maintaining a partial program, to which each technique/elaboration is applied to produce a linear sequence of development. Both approaches work under the supposition that the application of the technique does not affect the control flow of the skeleton.

This system suggests some possible instantiations of variables (in elaborations) for each clause. For some simple predicates it is also possible to use the system in 'auto' mode, in which the system completes the valid conditions of the predicates without user interaction. This characteristic is not offered by the other editors presented in this thesis.

This editor is a simple prototype which is based on Kirschenbaum et al.'s approach to techniques and skeletons. It can provide the information required in the *program history*. This prototype must be extended in order to allow the programmer to apply editing commands such as include new clauses in the program, add new subgoals in the program, etc. It could also be adapted to record the program history.

3.2.3 Bowles' Editor

The editor, named Ted, was implemented by Bowles and is also targeted at novices. Ted helps novices to learn Prolog by providing convenient patterns with which to construct programs, and allows the process of combining these patterns and learning in what circumstances they are appropriate [Bowles 93]. This editor uses a different approach to techniques to that of Kirschenbaum et al. In Bowles' approach techniques, for example, are local to clauses rather than applying across whole predicates. Ted provides a syntax editor consisting of a point-and-click interface supplemented with a set of edit operations which allow a technique to be applied to a clause. Both these editors (Robertson's and Bowles' editors) allow techniques to be included into programs in a simple way. In Robertson's editor, techniques are considered as software components which can be incorporated into a general control flow defined by skeletons, and therefore apply across whole predicate

definitions, whilst in Ted techniques are viewed at a clause level.

This editor, as with Robertson's editor, could to be extended in order to record the information of the *program history*, although the clause-level manipulations used by Ted would make this a little more difficult than with Robertson's editor.

3.3 Conclusions

Through this chapter we present approaches based on program schemata and editors based on the notion of skeletons and techniques. The first group does not provide the kind of information that the composition system requires such as flow of control and data flow. The second group of editors (based on the notion of skeleton and techniques) makes clear separation between the flow of control and the data flow. So, provided that users can use the editing operations of these systems, no extra work needs to be performed to obtain the *program history*.

The Edinburgh group have implemented two different editors in this style. One of them (implemented by Robertson) is a version which embodies the idea of skeletons and techniques defined by Kirschenbaum et al. The other editor, also implemented at Edinburgh University, called Ted uses a slightly different definition of techniques than the definition given by Kirschenbaum et al. Either of these editors could both provide the information that our composition system requires although the clause-level manipulation used by Ted would require more re-adaptation than with Robertson's editor.

4

The Composition Problem

This chapter presents related work on the composition problem. Firstly, we present several approaches to the composition problem in procedural languages. Secondly, we show transformation systems for functional languages based on the standard transformation operations such as unfold/fold operations and, finally, the last part of this chapter gives a description of the composition problem in logic programming languages.

Figure 4.1 shows the solutions to the composition problem for different paradigms of programming languages.

Much work has been done on combining a pair of programs with exactly the same flow of control in different paradigms of programming languages such as procedural, functional and logic programming languages. However none of them is sophisticated enough to allow major decisions in the combination process to be taken automatically by the system.

The features that we would like to provide in our combining system are as follows:

- Reduce user interaction, ie. the main decisions should rely on the system and not the user. For example, the selection of the order in which the transformation operations need to be applied should be decided by the system.
- Offer the system to people without any specialist knowledge in program transformation.

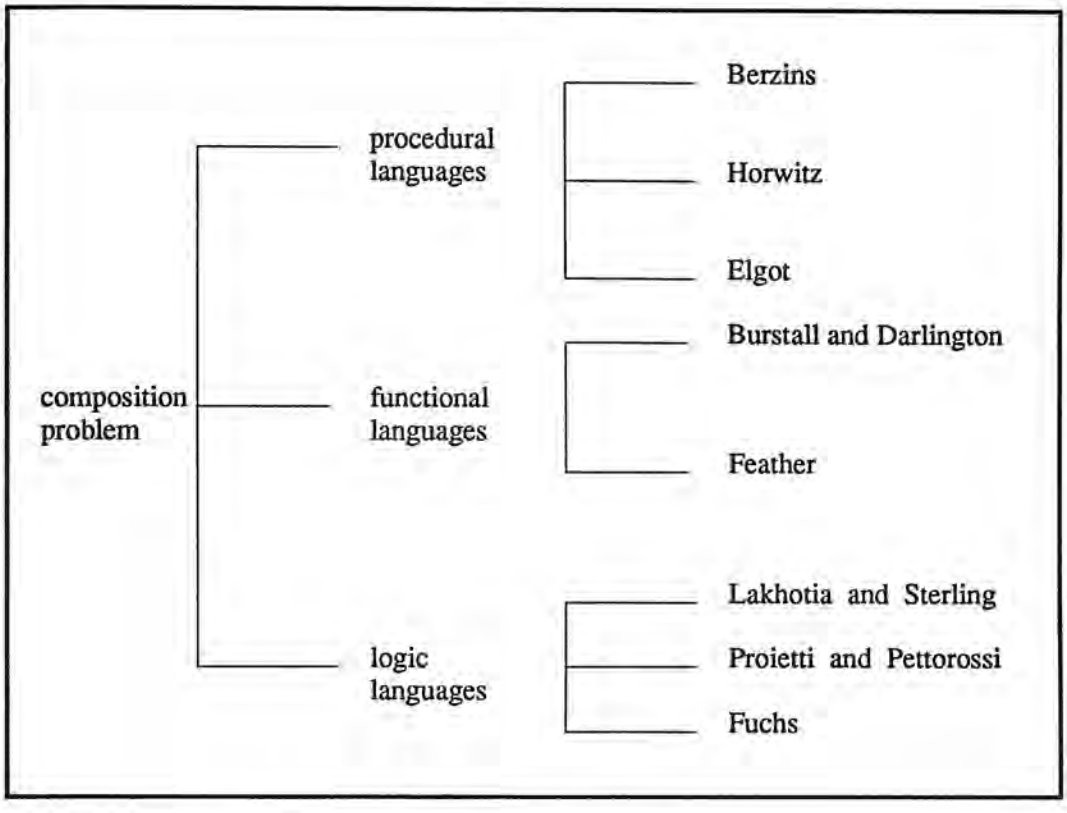

Figure 4.1: Composition Problem Solutions

- Allow the transformation of realistically sized Prolog programs.
- Allow the combination of programs with either the same or different flow of control.

4.1 Composition in Procedural Languages

The development of a large system of software involves many people doing extensions and modifications to the system that need to be co-ordinated to ensure correctness. Therefore an automatic method for merging two versions of a program that guarantees correctness of the result becomes important in software development. There have been several attempts at producing tools which help in the integration of programs. In environments like Unix we have tools such as *source control code systems(SCCS)* which support features of merges

of independent updates to a program. An SCCS system automatically merges a number of updates to a source text, treated as an uninterpreted text string, and the conflicts are reported where there is overlap of modified substrings.

Berzins characterises the semantics of the merged program in terms of the semantics of the original programs, and gives several rules for constructing a semantically correct merged program [Berzins 86]. He examines the problem of combining independent updates to a program in the context of applicative programs. An *applicative program* consists of a set of recursive function definitions and an expression to be evaluated. Function definitions have the form *name(parameters)=expression.* The expression to be evaluated can be a constant, a variable, a function application or a conditional expression. Berzins limits his approach to program extensions and does not include program modification. A program extension extends the domain of the partial function without altering any of the initially defined values, while a modification redefines values that were defined initially. In Berzins' approach a program that results from merging two programs A and B preserves the behaviour of both. A and B cannot be merged if they conflict at any points where both are defined.

The result of merging two programs is a set containing a definition for each function appearing in either of the programs where functions are matched by name. If a function appears in one of the programs but not in the other, it will appear in the merged program unchanged. If the same function appears in both programs, but with a different number of arguments, a syntax error should be reported. Otherwise, the formal parameters of one function are renamed if necessary to make the formal parameters of both definitions coincide, and the expressions in the bodies of the two definitions are merged in order to produce the body of the function in the merged program. However the paper does not describe in detail how the bodies are merged.

We avoided Berzins approach, due to the fact that it is restricted to combining programs with the same number of arguments. Consequently it does not provide the flexibility that our composition system requires, which is essential in order to allow the user to construct

more complex programs.

Horwitz et al.'s approach is more similar to our work because they merge programs by integrating extensions (variants) of an initial template [Horwitz *et al.* 88]. Their approach is restricted to a limited class of programs. The types of programs which can be merged are those which contain expressions containing only variables and constants, and in which the only statements used in the programs are assignment statements, conditional statements and while-loops. They present an algorithm *integrate* that takes as input three programs A, B and a program *Base*, where A and B are the two variants of *Base*. The output for the algorithm *integrate* is the new program M which integrates A and B with respect to Base, unless it is detected that the changes made to Base to produce A and B interfere.

For program integration, program dependence graphs (PDG's) are used because it is claimed that this representation captures only relevant orderings of statements within control structures and because program dependence graphs are a suitable representation for program slicing [Horwitz *et al.* 88]. A slice of a PDG with respect to variable x is a graph containing all vertices on which x has a transitive flow or control dependence (i.e. all the vertices that can reach x via flow or control edges) [Horwitz *et al.* 88]. Further details are defined in Appendix I.

To illustrate, we will consider the program `main` defined below, which sums integers from 1 to 10 and leaves the result in the variable `sum`. The program dependence graph corresponding to this example (see Figure 4.2) was designed with the following convention: the boldface arrows represent control dependence edges; dashed arrows represent def-order dependence edges; solid arrows represent loop-independent flow dependence edges; solid arrows with a perpendicular mark represent loop-carried flow dependence edges.

The program which computes the `main` is shown as follows:

```
sum := 0;
x:= 1;
while x < 11 do
            sum := sum + x;
            x := x + 1
od
```

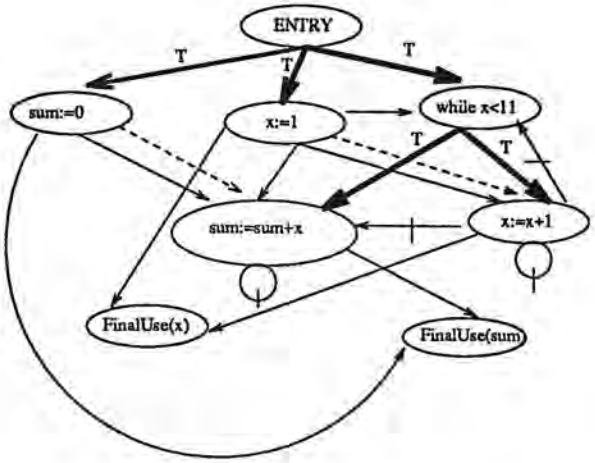

Figure 4.2: An example program

In Figure 4.2 the program `main` has two variables `x` and `sum`, so for each variable we have an *initial state node* and a *final use node.* There are several edges represented. All the edges labelled `true` are those edges which are not subordinate to any control predicate. There is a *control dependence edge* from a distinguished vertex from which the control flow starts called the `ENTRY` to each of the nodes which are not subordinate to any predicate. For instance, we have edges from `ENTRY` to `sum:=0`, from `ENTRY` to `x:=1`, from `ENTRY` to `while x<11`, from `while x<11` to `sum:=sum+x` and finally from `while x<11` to `x:=x+1`.

This graph also has def-order dependence edges from vertex `sum:=0` to `sum:=sum+x` because both vertices are defining the same variable and for the same reason we have a def-order dependence edge from `x:=1` to `x:=x+1`.

There are loop-independent flow dependence edges from vertex `sum:=0` to `sum:=sum+x`; from `sum:=0` to `finalUse(sum)`; from `x:=1` to `finalUse(x)`; from `x:=1` to `sum:=sum+x`, from `x=1` to `sum:=sum+x` and from `sum:=sum+x` to `finalUse(sum)`. There is an edge from the definition of the variable to every place in which the same variable is used.

Finally in the graph we have loop-carried flow dependence edges from `x:=x+1` to `while x<11` and in the vertex `sum:=sum+x` and vertex `x:=x+1`. These are both enclosed in a loop.

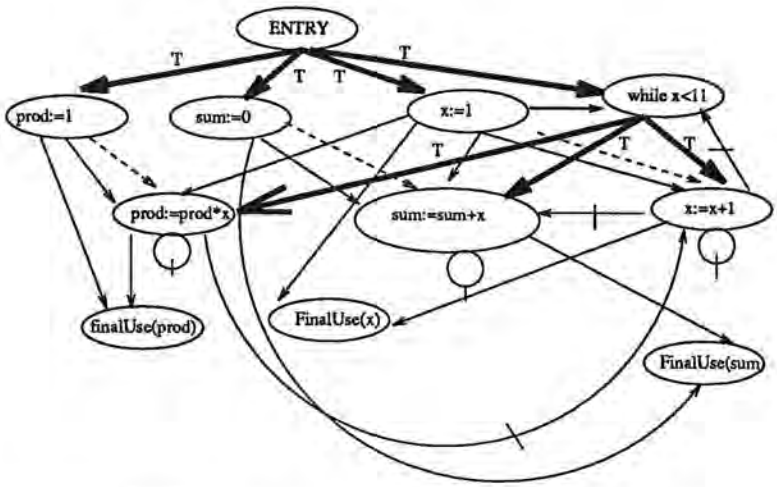

Figure 4.3: The program dependence graph for variant A

We now consider two variants of the `main` program. In variant A, two statements have been added to the original program to compute the product of numbers from 1 to 10. The set $D_{A,Base}$ contains three extra vertices (ie. the assignment vertices labelled `prod := 1` and `prod := prod*x` and `finalUse(prod)`). The program dependence graph corresponding to this variant A is in Figure 4.3, which is similar to the previous graph the difference being that this graph includes the new instructions in the program *Variant A*.

Variant A

```
prod := 1;
sum := 0;
x := 1;
while x < 11 do
            prod := prod * x;
            sum := sum + x;
            x := x + 1
od
```

In variant B one statement has been added to compute the *mean* of the 10 integers. The program dependence for $D_{B,Base}$ contains two more vertices than the program base: the assignment vertex labelled `mean := sum/10` and the `finalUse(mean)`. The program dependence graph corresponding to this variant B is in Figure 4.4.

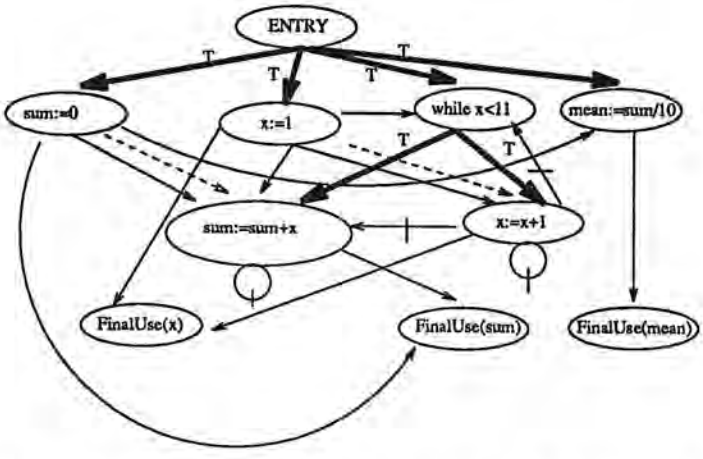

Figure 4.4: The program dependence graph for variant B

Variant B

```
sum := 0;
x := 1;
while x < 11 do
            sum := sum + x;
            x := x + 1
od
mean := sum/10
```

The two programs variant A and variant B represent extensions of the original program that computes the sum of 10 integers. We now show how to merge the program dependence graphs of two variants (extensions) so that the differences between the behaviour of program *Base* and its variants are preserved.

Two programs enhanced from a program *Base* are merged by merging their program dependence graphs. The merge process is done using the changed behaviour in program A that is characterised by the slice $G_A/D_{A,Base}$ (which captures the computation threads of A that differ from those in Base) and slice $G_B/D_{B,Base}$ (see Figure 4.5 and 4.6). Therefore the merged graph G_M shown in Figure 4.8 should be formed from the elements of G_A that contain the changed behaviour of A, the elements of G_B (that characterise the changed behaviour of B) and the elements from G_{Base} that characterise the common unchanged behaviour (*Base* program).

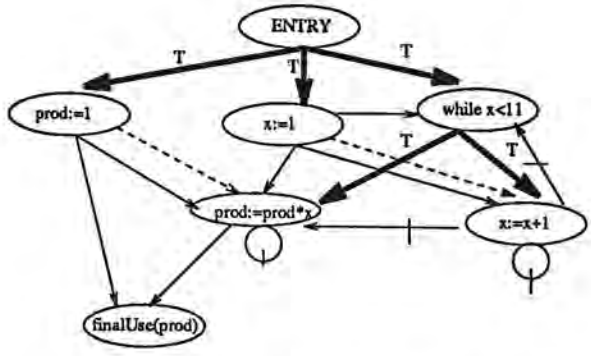

Figure 4.5: The slice GA/DA,Base

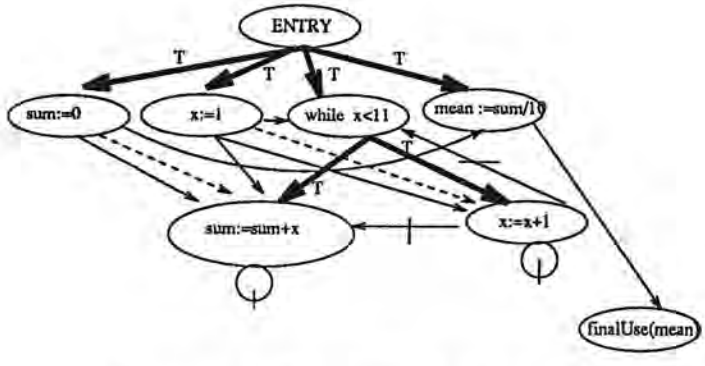

Figure 4.6: The slice GB/DB,Base

$$G_M = (G_A/D_{A,Base}) \cup (G_B/D_{B,Base}) \cup (G_{Base}/(D_{A,Base} \cap D_{B,Base}))$$

where the slice $G_A/D_{A,Base}$ is the changed behaviour in A, the slice $G_B/D_{B,Base}$ characterises the changed behaviour in B and the slice $G_{Base}/(D_{A,Base} \cap D_{B,Base})$ represents the behaviour that is preserved in both A and B. This program slice is shown in Figure 4.7. The union of slices from Figure 4.5, Figure 4.6 and Figure 4.7 gives the program dependence graph G_M.

Program slicing is used in the algorithm *integrate* to determine the changes in the behaviour of each variant with respect to the *Base* program. Programs A and B interfere if a merged program dependence graph, G_M (that can be created using the algorithm *integrate*) cannot show the changed behaviour of the two variants A and B. This may happen if it is possible

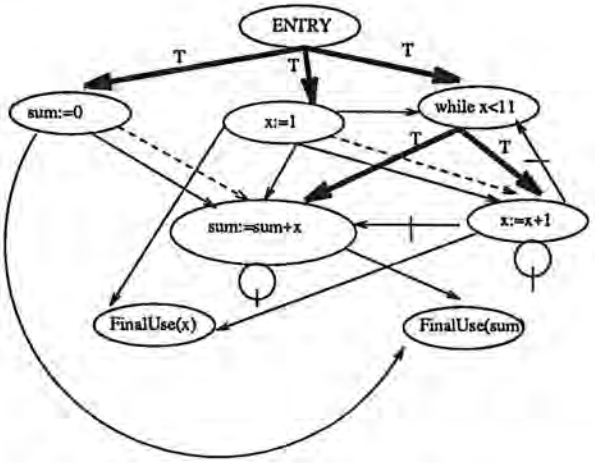

Figure 4.7: The slice which represents the behaviour that is preserved in both A and B

that G_M will not preserve the differences in behaviour of A or B with respect to *Base*, or because the union of two feasible PDG's is not a feasible PDG.

The last stage of the integration method is to produce the merged program from the merged program dependence graph. Given a program dependence graph G_M (see Figure 4.8) that was created by merging non-interfering variants A and B, the procedure defined with this purpose (called reconstitute program) must decide if G_M is feasible, and will then produce the corresponding program. The operation reconstitute-program creates a program corresponding to the program dependence graph G_M by ordering all vertices, otherwise it discovers that G_M is infeasible. G_M is infeasible if the vertices in a subtree rooted at v cannot be ordered.

Merged program

```
prod := 1;
sum := 0;
x := 1;
while x < 11 do
            prod := prod * x;
            sum := sum + x;
            x := x + 1
od
mean := sum/10
```

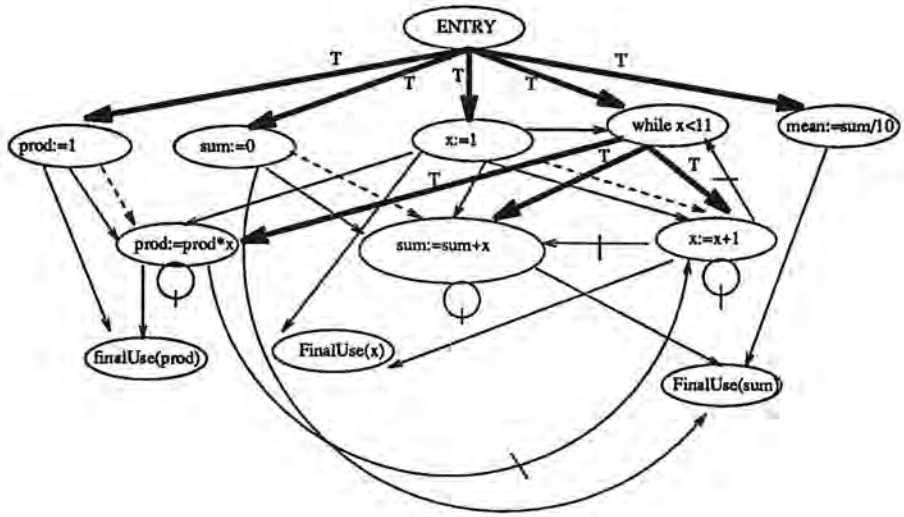

Figure 4.8: The graph G_M

Horwitz et al.'s proposal combines programs efficiently if they have been derived from an initial generic template. However this approach is very restrictive. Only programs which can be merged are those which contain expressions containing only variables and constants, and in which the only statements used in the programs are assignment statements, conditional statements and while-loops. It needs to be extended to handle other programming language constructs, such procedure and function calls, break statements and I/O statements.

Another approach to the composition problem in procedural languages is the use of the theory of flowcharts. This approach was developed by Elgot [Elgot 70]. He defined a graphical approach which consists of the combination of atomic flowcharts such as *if-then-else*, *statements*, etc. He describes the composition of two schemata using the standard definition of composition of two functions.

The following example taken from [Elgot 70] shows the composition of two program schemata using flowcharts (see figure 4.9). In the figure program P was constructed using the schema *if-then-else* in which b is a boolean expression and A is T or F and $\overline{A}$ is the other.

Let $\beta : X \longrightarrow X \times \{T, F\}$ be a function satisfying the following definition:

$$\beta(x) = (x, b(x))$$

where T is true and F is false. This function represents the if-then-else flowchart. This function β will be used in the combination of program P and program Q.

In the same figure (figure 4.9), the program Q is constructed by using statement schemata **r** and **s**. These schemata can be represented as functions such as $r, s : X \longrightarrow X$ where X is any set. We use the function $\oplus$, called *OR-exclusive*,

$(r \oplus s) : X \times \{T, F\} \longrightarrow X$

and defined by

$(r \oplus s)(x, T) = r(x)$ *or* $(r \oplus s)(x, F) = s(x)$

where $x \in X$. That is, it selects the first or second function according as to whether the second argument is true or false.

The composition of program P and program Q can be represented as a function γ defined as follows:

$\gamma : X \xrightarrow{\beta} X \times \{T, F\} \xrightarrow{(r \oplus s)} X$

$\gamma(x) = (r \oplus s) \circ \beta(x) \equiv (r \oplus s)(\beta(x))$

This function is the composition of two functions: *OR exclusive* denoted by $(r \oplus s)$ and function β where the symbol $\circ$ denotes the combination operator. Then the combined program is formed from the composition of the functions β (representing if-then-else) and *OR exclusive* (statements r and s). The graphical representation of this combined program using flowcharts is shown in figure 4.9.

In short, Elgot presents a flowchart approach in which simpler programs are constructed by interconnecting atomic parts such as *if-then-else statements* by associating each program

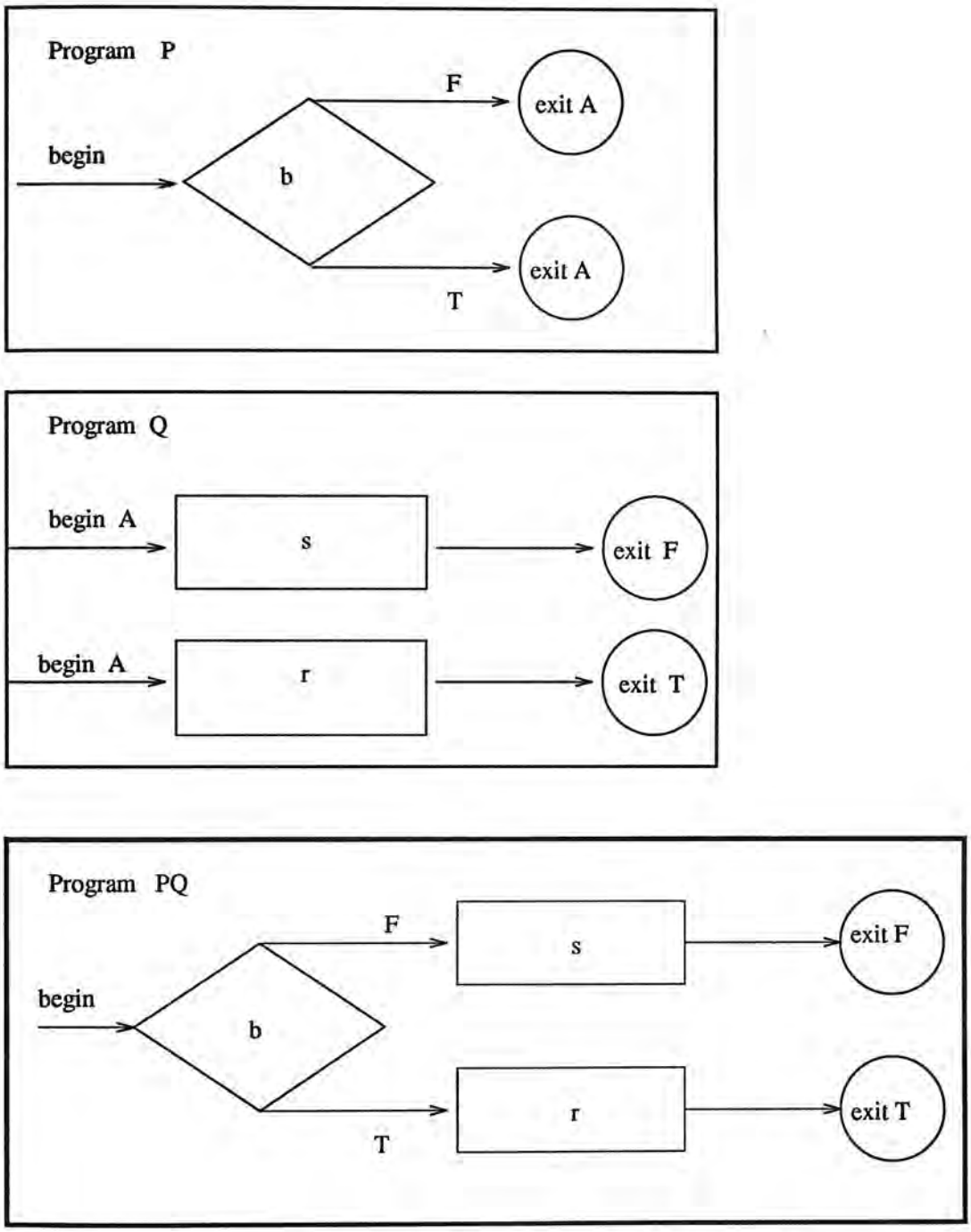

Figure 4.9: Composition of Program Schemas in Procedural Language

as a function. He claimed that more complex programs can be obtained by composing the functions in the usual way.

In our opinion this approach is restricted to combining very simple program flowcharts and so does not go very far. In particular, it is unlikely that users would want to combine flowcharts such as the one presented above. Besides being very restricted, it is also impractical because the function which performs the flowchart combination needs to be defined by the user, and this becomes much more complex after each performance of the combination process.

4.2 Transformation Systems for Functional Languages

Burstall, Darlington, Sato, Lakhotia and Sterling and other researchers in the field have found that program transformation is useful as a methodology for program development. A wide range of programming systems make use of program transformation as part of their operation. Program transformation has been used to improve the efficiency of a program automatically. A second use of transformations is in program synthesis: the generation of programs from a specification of the problem. Finally, a third use is in program adaptation: adaptation of a program written in one language to a related language with different primitives [Partsch & Steinbruggen 83, Rich & Walters 86].

Burstall and Darlington developed a system for the transformation of programs written in a functional language based on recursive equations. They define transformation rules: definition, instantiation, unfolding, folding, abstraction and laws of primitives (associativity, commutativity). These rules preserve partial correctness, that is that the new equation may not terminate in some cases when the original equations did [Burstall & Darlington 77].

The method used by Burstall and Darlington [Burstall & Darlington 77] consists of the repeated application of manipulations to recursion equations to produce modified recursion equations. The idea is to start with a very simple and hopefully correct program, then to

transform it into a more efficient one by altering the recursion. The system works by using unfolding/folding rules, and proposes folding operations that the user accepts or rejects, asking for an alternative. The user must supply equation definitions, and also provide, translation rules according to the data structures of the programs. For example, the associativity rule can be applied to relations such as addition over numbers, concatenation over lists and union over sets. Another requirement of the user is to give explicit reduction rules. Examples of programs that have been tested on Burstall and Darlington's system are: algorithms for computing Fibonacci numbers and cartesian products.

For example, consider the program `f` which traverses a binary tree and computes the sum of its tips (which are the values assigned to the leaves), and `g` which computes their product. The task is to build a program which computes both of them at once. We assume a tree is either a `tip` or a `tree` consisting of two trees (`tree` and `tip` are the constructor functions).

[1]	$f(tip(x)) \Leftarrow x$	*given*
[2]	$f(tree(x,y)) \Leftarrow f(x) + f(y)$	*given*
[3]	$g(tip(x)) \Leftarrow x$	*given*
[4]	$g(tree(x,y)) \Leftarrow g(x) * g(y)$	*given*
[5]	$h(x) \Leftarrow < f(x), g(x) >$	*given*
[6]	$h(tip(x)) \Leftarrow < f(tip(x)), g(tip(x)) >$	*instantiation*
	$\Leftarrow < x, x >$	*unfolding* 1,3
[7]	$h(tree(x,y)) \Leftarrow < f(tree(x,y)), g(tree(x,y)) >$	*instantiation*
	$\Leftarrow < f(x) + f(y), g(x) * g(y) >$	*unfolding* 2,4
	$\Leftarrow < u + v, w * t >$	
	where $< u, w, v, t > = < f(x), g(x), f(y), g(y) >$	*abstraction*
	$\Leftarrow < u + v, w * t >$	
	where $<< u, v >, < w, t >> = < h(x), h(y) >$	*folding with* 5

Equations 1 and 2 define program `f` and, similarly, the equations 3 and 4 define program `g`. Definition number 5 is the join specification. Equation number 6 is obtained by instantiation of the join specification. The right hand side in number 6 is arrived at by applying the unfold operation to 5 using the definitions 1 and 3. Equation number 7 is obtained by instantiation of the join specification in 5. The equivalence of the right hand side of equation number 7 is the result of the application of the unfold operation using equations 2 and 4 and, finally, the last equivalence of the right hand side of equation 7 is obtained

by applying the fold operation using 5.

The main characteristic of Burstall and Darlington's system is that it requires a considerable amount of user guidance in the transformation of the program as the size of programs increase. The order in which the transformation operations are performed is vital in the transformation of the program. A wrong selection order might produce a non-efficient combined program. In conclusion we can say that the user needs to make numerous decisions, so he needs to have a good knowledge of the transformation process in order to make the right choices.

The ZAP system developed by Feather in [Feather 78] is based on the Burstall and Darlington system. The system was designed with the purpose of supporting large-program transformation. The principle of the system is the fold/unfold method as in the Burstall and Darlington system and the input target language is the same as the previous Burstall and Darlington system, NPL. The difference between ZAP and Burstall and Darlington's system is that in this system the user can write meta-programs to be applied to NPL programs, and therefore to direct the transformation of these programs at this high level.

ZAP also can be seen as a system which only verifies the expected answer (i.e. it can be used as a checker). However the most interesting characteristic is that the user can define approximately the answer that he requires and the system will fill in all the details of the answer. The way in which this is achieved is to include in the pattern $$ symbols which the system will match to portions of expressions. In the example defined using ZAP's commands, we gave to the system the following pattern:

```
$$ (A,B,C,D,N,f(A,B,C,D,N))
```

indicating that we expect an answer containing a call to $f(A, B, C, D, N)$; expressions involving A, B, C, D and N formed with functions $+$, $*$ and *sub*; constructors such as *successor* (succ) and constants such as 0. See the dialogue with ZAP system shown later in this section.

The user must provide rewrite rules and auxiliary function definitions. In the case that a

user wants to make a transformation, the ZAP language allows the user to define context rules and definitions which are to be available. ZAP also allows the user to specify the goal of the transformation as a schematic pattern telling the general form of the recursive definition required.

To give an idea to the reader how the dialogue with ZAP is performed, consider the example that computes the scalar product of two vectors. The process of definition using the commands of the ZAP system is presented below, where comments are represented in square parentheses. The function *sub* allows access to components of vectors (i.e. $(X\ sub\ succ\ N) * (Y\ sub\ succ\ N)$ is the same as $x[N+1] * y[N+1]$), and *succ* is the constructor function *successor*.

START

DEF [give here NPL definitions for f and dot (functions used for computing the scalar product of two vectors]

$dot(X, Y, 0) \Leftarrow 0$

$dot(X, Y, succ\ N) \Leftarrow dot(X, Y, N) + X\ sub\ succ\ N * Y\ sub\ succ\ N$

$f(A, B, C, D, N) \Leftarrow dot(A, B, N) + dot(C, D, N)$

END

CONTEXT

UNFOLD f dot [declare that f and dot are to be used in

unfolding operation]

USING + * sub f [state which functions are permitted in the

transformed equations]

LEMMAS ASSOCIATIVE + [declare + to be associative]

LEMMAS IDENTITY + 0 [declare 0 to be identity for + operation]

TRANSFORM [After creating context redefine f]

GOAL f(A,B,C,D,0) [expressions following GOAL will be the

left-hand sides of the new equation f]

GOAL f(A,B,C,D,succ N) $\Leftarrow$ $$ (A,B,C,D,N,f(A,B,C,D,N))

END

DELETE f(A,B,C,D,N)

END [to end the transform block]

STOP [to exit from the system]

In the first stage we define the functions f and *dot* in NPL. The system will try to transform the left-hand sides provided in GOAL, then $f(A,B,C,D,0)$ expands to 0 (by unfolding and applying reductions) which is a constant. The equation $f(A,B,C,D,0) \Leftarrow 0$ has thus been found.

$f(A,B,C,D,succ\ N)$ unfolds to

$dot(A,B,N) + (A\ sub\ succ\ N) * (B\ sub\ succ\ N) + dot(C,D,N) + (C\ sub\ succ\ N) * (D\ sub\ succ\ N)$

Because the last definition contains the function *dot*, it is not acceptable as the answer (this is defined in the instruction USING), so the system asks for a pattern which in this case should be the following pattern:

$f(A,B,C,D,N) + (A\ sub\ succ\ N) * (B\ sub\ succ\ N) + (C\ sub\ succ\ N) * (Dsub\ succ\ N)$

A pattern is the right hand side of the equation. This is defined by the user and is defined according to what the user expects as an answer. This definition requires knowledge about the transformation process, and this knowledge needs to be provided for each specific example. After this pattern is presented to the system, the system will check that this unfolded expression is equal to the unfolded left-hand side. Because that is the case, the

new equation formed by this as right-hand is added.

The ZAP's dialogue requires a good understanding of how to transform the programs. Even when ZAP has less user interaction than the NPL transformation system developed by Burstall and Darlington it still requires knowledge about program transformation to be supplied by the user. For this system the user is required to give the following information:

- A list of equations which define the programs to be transformed, and any auxiliary definition required for the transformation process.
- A list of lemmas to be used as rewrite rules (written in equation notation) and definitions of the function properties such as associativity, commutativity, etc.
- A list of all the properly instantiated left hand sides of the equations on which the user wants the system to work.

4.3 Composition in Logic Programming Languages

In this section we describe several approaches to the combination problem in logic programming languages: Lakhotia and Sterling's approach based in standard transformation operations, Proietti and Pettorossi's algorithm based in Tamaki and Sato's definitions for unfold/fold [Tamaki & Sato 84] (for further details on Tamaki and Sato's work see Appendix E) and Fuchs and Fromherz's approach based in transforming program schemata.

1. Lakhotia and Sterling created several methods for combining Prolog programs. Their earlier work was to develop the join 1-1 method [Lakhotia & Sterling 87]. This algorithm composes programs by performing clausal join of corresponding clauses. Clausal join does not compose subgoals that have local variables in their arguments. The join 1-1 method only combines programs with the same flow of control.

   Also Lakhotia and Sterling describe in [Lakhotia & Sterling 87] another method called the procedural join which combines all the possible pairs of clauses. This

method does not impose any restrictions like the previous one. Again, this method only combines programs with the same flow of control.

Later Sterling and Kirschenbaum describe an algorithm which does not have the limitation of clausal join [Sterling & Kirschenbaum 91]. The problem is solved by using extension relationships which must be provided by the user. However, providing these extension relationships require a good understanding of a skeleton classification. In particular, the user needs to compare the flows of control between the skeleton and each of the programs to be combined, identify if one program is derived from a particular skeleton and then be able to match the inherited components (between skeleton and program). Furthermore, given two arbitary programs it may generally not be possible to determine if there is a relation of extension between them.

The basic operation in their algorithm is the composing of pairs of clauses. This operation is restricted to combine only corresponding clauses. The composition algorithm composes the heads of clauses and their bodies with respect to a join specification. As the previous join 1-1 method, this method only combines programs with the same flow of control which are extensions of the same skeleton.

In Chapter 6 we show how we have reimplemented the join 1-1 method and the procedural join. These methods are suitable for combining some classes of programs in our classification schema.

2. The Proietti and Pettorossi's algorithm is based on unfolding, folding and addition of new join specifications [Proietti & Pettorossi 92]. The main characteristic of the algorithm is that it eliminates unnecessary variables. This algorithm requires three actions which need to be defined by the user: the introduction of a set of join specifications for the folding step, selection of the calls in the body of the clauses for unfolding stages and choice of arithmetic laws to be applied.

   For example consider the programs `get_odd/2` and `count/2`. The program `get_odd/2` extracts odd numbers from a list, and `count/2` counts the number of elements in a list.

```
1:  get_odd([], []).
2:  get_odd([H|T], [H|O]) :-
        odd(H),
        get_odd(T, O).
3:  get_odd([H|T], O) :-
        even(H),
        get_odd(T, O).
```

```
4:  count([],0)
5:  count([H|T], Count) :-
        count(T,C1),
        Count is C1+1.
```

A new program `count_odd/2`, which extracts the odd numbers and computes the length of the list at the same time, is generated by using the join specification defined as follows:

```
6:  count_odd(L, C) :- get_odd(L, Os), count(Os, C).
```

Firstly, we apply the unfold operation to the definition of `get_odd/2` in `count_odd/2`:

```
7:  count_odd([], C) :-
            count([], C).
8:  count_odd([H|T], C) :-
            odd(H),
            get_odd(T, Os),
            count([H|Os], C).
9:  count_odd([H|T], C) :-
            even(H),
            get_odd(T, Os),
            count(Os, C).
```

After unfolding the call `count/2` in clause 7 by using clause 4 we obtain the following clause:

```
10:  count_odd([], 0).
```

By folding clause 9 using clause 6 we get the clause shown below. At this stage no new join specification is required:

```
11:  count_odd([H|T],C) :-
            even(H),
            count_odd(T,C).
```

A new join specification `new1/3` must be introduced by the system in order to continue transforming the program. The new join specification is shown as follows:

```
D: new1(H,T,C) :- odd(H), get_odd(T, Os), count([H|Os], C).
```

Clause 8 is folded by using the new join specification `new1/3` giving the following clause:

```
8f:  count_odd([H|T], C) :- new1(H,T,C).
```

Unfolding the call `count/2` in clause `D`, we obtain the following clauses:

```
D1:  new1(H,T,C) :- odd(H), get_odd(T, Os), count(Os, C1), C is C1+1.
```

The selection of which subgoal is unfolded in the new join specification (clause `D`) requires user interaction. In order to do this task the user needs to have some knowledge of program transformation. In clause `D` the user has two options: to unfold the call `get_odds/2` or `count/2`. The wrong selection of which subgoal need to be unfolded could cause the generation of inefficient combined program.

The fold operation is applied in clause D_1, obtaining the following clause:

```
D2:  new1(H,T,C) :- odd(H), count_odd(T,C1), C is C1+1.
```

At this stage the resulting program consists of clauses 10, 11, 8f and D_2.

A final simplification step is done by unfolding transient predicates, like new1/3. Unfolding new1/3 in clause 8f give us the following clause:

```
D3:  count_odd([H|T],C) :- odd(H), count_odd(T,C1), C is C1+1.
```

Finally, the resulting combined program is formed by clauses 10, 11 and D_3.

This method produces efficient combined programs but requires user interaction at important stages of the composition process, such as to choose which call in the new join specification D is unfolded. This algorithm is not making use of any information concerning the control flow of the program or techniques used in it.

3. Another approach to the composition problem is the work developed by Fuchs and Fromherz [Fuchs & Fromherz 91]. The combination of programs is done by transforming input program schemata into output schemata. Each transformation schema represents one transformation strategy, for instance a sequence of unfold and fold operations. This process is performed in three stages: *abstraction* of the programs to a program schemata, *selection* of a transformation schema with this schema as input and a suitable schema as output, and finally *specialization* of the output schema to the transformed program. User interaction is required during the *selection* stage, which requires that the user concentrates on the form of the input and output programs.

   - Abstraction. For each program P_i we find a program schema S_i which abstractly describes P_i. This abstraction produces a set of substitutions θ_i for schema variables, and the literals A_i are abstracted to the abstract literals G_i.
   - Selection. A transformation schema transforms the set of abstract terms $\{S_1/G_1, \ldots, S_n/G_n\}$ into an abstract term S/G. However, if there are several transformation schema which have $\{S_1/G_1, \ldots, S_n/G_n\}$ as input then we need to select the transformation schema which generates the desired output program schema S, together with an abstract literal G.
   - Specialization. We apply the substitution $\theta = \theta_1, \ldots \theta_n$ to S/G in order to get the transformed program $P = S\theta$ and the transformed literal $A = G\theta$.

   For example, consider the predicate `sum/2` which computes the sum of the elements of a list and the predicate `count/2` which computes the number of elements in a list. These programs are shown as follows:

```
sum([],0).
sum([H|T],Sum) :-
            sum(T,Sum1),
            Sum is Sum1 + H.
count([],0).
count([H|T],C) :-
            count(T,C1),
            C is C1 + 1.
```

These two programs can be abstracted to one schema of the Gegg-Harrison schemata, defined in Chapter 2. This particular programs match with the schema_A. This abstraction process is performed automatically by Fuch's system. Let schema S_1 be the schema for the program **sum/2** and S_2 the schema associated with **count/2**. These two schemata and the abstract literals $G1$ and G_2 are shown as follows:

G_1 : `Schema_A1(L,&g1)`

S_1 :

```
Schema_A1([],&11).
Schema_A1([H1|T1],&12) :-
            process11,
            Schema_A1(T1,&14),
            process12.
```

G_2 : `Schema_A2(L,&g2)`

S_2 :

```
Schema_A2([],&21).
Schema_A2([H2|T2],&22) :-
            process21,
            Schema_A2(T2,&24),
            process22.
```

Note that the literals A_1 and A_2 (which are the operands in the join specification) are abstracted to the literals G_1 and G_2. For this our working example, A_1 is defined as `sum(X,S)` and A_2 as `count(X,C)`. After performing the matching by using `schema_A`, G_1 is instantiated to `Schema_A1(L,&g1)` and G_2 to `Schema_A2(L,&g2)`.

As a second stage, called *selection*, the user chooses the output schema S for the combined program. This selection implies that the user knows in advance the form of the combined program. This stage might not be easy for users in general.

For our working example let us assume that the user choose the following output schema:

```
G: Schema_A1'_'Schema_A2(L,&g1,&g2).

S:
Schema_A1'_'Schema_A2([],&11,&21).
Schema_A1'_'Schema_A2([H1|T1],&11,&21):-
            process11,
            process21,
            Schema_A1,'_',Schema_A2(T1,&14,&24),
            process12,
            process22.
```

This output schema encodes a sequence of transformation that is automatically performed to the schemata devised in the first stage of this process. So, in order to obtain the combined program, the system needs to derive S/G from S1/G1 and S2/G2. This is performed by applying the following set of operations: one definition, two unfolding steps and a folding step which is encoded on the chosen ouput schema.

The system will produce the following program:

```
sum_count_a1([],0,0).
sum_count_a1([H|T],Sum,Count) :-
            sum_count_a1(T,Sum1,C1),
            Sum is Sum1 + H,
            Count is C1 * 1.
```

Fuchs's approach produces efficient combined Prolog programs, but requires user interaction in the *selection* stage at which the user must select for a given input schemata one output schema.

4.4 Conclusions

Through this chapter we present several approaches to the combination problem. Firstly we introduce the approaches taken for procedural languages. None of them provides the flexibility that we would like to have in our composition system. They are quite restricted in terms of the kind of programs that can be combined (same flow of control) and also restricted to programs which only contain assignment statements, conditional statements, and while-loops. Secondly, we present functional approaches to the combination problem which require a lot of direction from the user in terms of the sequence of the transformation

operations to be applied to a specific program and, finally, we show several approaches to the combination problem in logic programming languages. Again, none of them provide all the features that we would like to have in our composition system.

The transformation system developed by Burstall and Darlington requires knowledge about program transformation. User interaction is required to direct the transformation of the program. This guidance is at a very low level; the user needs to define which equations need to be unfolded or folded. The ZAP system tries solve this problem, but also requires user interaction in the definition of how to transform the program when the user is defining the problem in the language provided for this task (see dialogue shown in section 4.2). This definition is prepared in a more abstract way than in the previous system. However, it requires knowledge about program transformation.

Lakhotia and Sterling define two combination methods which are based on the standard transformation operations as defined by Tamaki and Sato [Tamaki & Sato 84] and one method based on the similarity notion between programs and the skeleton from which they are derived. These methods work for a restricted classes of program (same flow of control) but it also requires user intervention.

Proietti and Pettorossi propose a combination method for logic programs with the same flow of control, using basic fold/unfold transformations; this approach relies largely on the participation of the user and its efficiency depends on the decisions the user makes.

Fuchs' approach reduces user interaction in terms of deciding the sequence of transformations, but the user still needs to choose the output schema.

As a final conclusion we can say that we did not find a method general enough to handle the combination of every possible pair of programs in an efficient manner.

5

Classification of Programs and Automatic Selection

The problem of combining programs efficiently can be made more tractable by taking into consideration the characteristics of the programs to be combined. In this chapter we firstly describe a classification of Prolog programs according to their structural features. These classes are characterised by standard patterns of flow of control. Secondly, we describe how the program history is obtained from the techniques editor. Thirdly, we define our set of combining methods and finally we define the automatic selection of the combining methods.

5.1 Classification of Prolog Programs

Chapter 4 describes several approaches to the composition problem for procedural languages (e.g. Berzins and Horwitz) and also for logic programming languages (e.g. Proietti and Pettorossi, Lakhotia and Sterling and Fuchs and Fromherz). These approaches are restricted to a specific class of programs that can be combined. Our solution to this problem was, firstly, to divide the set of programs that we can write in Prolog into classes according to stipulated structural features present in the original skeleton (from which the program was constructed) and secondly to implement a combining method which works efficiently for each class of programs defined in our hierarchy (see Figure 5.1).

Each class is associated with a skeleton. The skeletons used in our classification were devised by Lakhotia and Sterling [Lakhotia & Sterling 87]. Our classification of programs makes use of the skeletons: traverse, short_traverse, search, meta-interpreter and counter, which are described in [Lakhotia & Sterling 87] which are also defined in Appendix C. This list of skeletons is not complete and might be extended in order to cover more classes of programs but with this current set of skeletons it is possible to write many of the Prolog programs commonly found in textbooks. Each of these skeletons defines a different class of programs as shown in Figure 5.1. The important point is that each program generated from a skeleton inherits the flow of control of that skeleton. Therefore, (as far as flow of control is concerned) it is sufficient to obtain information about the program from just the features of the skeleton.

Note that our classification given in Figure 5.1 is an adapted version of the Lakhotia and Sterling skeleton classification [Lakhotia & Sterling 87]. Lakhotia and Sterling proposed to classify skeleton in 3 categories: meta-interpreters, parsers and manipulation of recursive data structures. We start with this classification and we define mutations of these classes. After this we define features that each class of program holds and then we use these features for automatic selection of the combination method.

Our classification contains five classes of program with the same flow of control as is given by the skeleton and also has five classes with a different flow of control. These programs with altered flow of control are called *mutants*. Mutant programs are created by adding to the original skeleton subgoals which change the flow of control or by adding clauses to the definition of the skeleton. The addition of the new clauses in a program are required to preserve the structure of the initial flow of control of the program extending this with additional behaviours. The clauses which can be added are thus restricted by the following conditions.

The base case (in the skeleton) states how to traverse the data structure and identifies each skeleton as unique. It has the restriction that it cannot be changed to a different base case with different pattern of data structure. The reason for this restriction is that if we alter

the flow of control of the initial skeleton in its main structure, then the resulting mutant might not belong to the same class as the starting program. For example, by changing the base case of a meta-interpreter program we might not arrive at a well defined mutant of meta-interpreter.

It is not possible to alter existing recursive clauses. So the user can only add new recursive clauses which operate over the same data structure as the original program (although with different tests).

In Figure 5.1 the classification by skeleton is done on the left-hand subtree and a corresponding mutant category for each skeleton class is shown on the right-hand subtree. Currently in our combination system, a given program is classified as belonging to a class by using the *program history*. It might also be feasible to extract the components of the program (skeleton and techniques) for a restricted set of programs by using program analysis [Bental 92].

The classes defined in the traversal family are defined, in our knowledge base of skeletons, only for the case when the data structure is a list or a tree. However we could also deal with different data structures by defining new classes and including them in the traversal family.

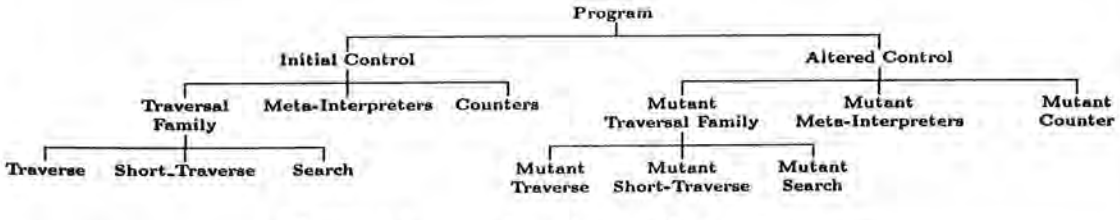

Figure 5.1: Classification of Prolog Programs

Our combining system provides us with a set of methods which can be used for combining programs in the same class or programs belonging to different classes. However there exist classes of programs which cannot be combined into a single recursive program. These are described in Section 6.5.

5.1.1 Classification of pairs of Programs

In order to assign a method for combining each pair of programs, we first classify the pairs of programs that will be combined into two groups: programs with the same flow of control and programs with a different flow of control. Programs with the same flow of control are those which are enhancements of the same skeleton. Programs with a different flow of control may have originated from the same skeleton but at some stage of the development extra clauses were added to at least one of the programs. Alternatively they could be programs built using different skeletons: for instance, one program constructed using the traverse skeleton and the other using the short_traverse skeleton. The classification of pairs of programs is shown in figure 5.2. This classification consists of pairs of programs belonging to Traverse-restricted, Traverse-general, Meta-counter, Non-restricted and Different-flow. We made a distinction between programs belonging to the Traverse-restricted and Traverse-general. This was done in order to determine precisely the programs in which the traversing of the data structure can be synchronised (Traversal-restricted). Further details are described in Section 6.6.2.

- Pairs of type *Traverse-restricted* are programs which derive from the same skeleton (except the meta-interpreters or counter skeleton), both have the same number of clauses; both programs (taken clause by clause) have the trivial condition (ie., logically equivalent to true) or in both clauses the `test` should be identical. Each *test* determines a possible case of the induction parameter. A test can be an arithmetic expression or a boolean expression. Also these programs traverse just one data structure of a given type (for instance lists or trees). Finally the pattern used to compose or decompose the data structure should be the same in each program.

- Pairs of type *Traverse-general* are programs which derive from the same skeleton (except the meta-interpreters or counter skeleton), both have the same number of clauses; where each program has a different test (different to the trivial condition) and each program can traverse a different data structure of the same type. The

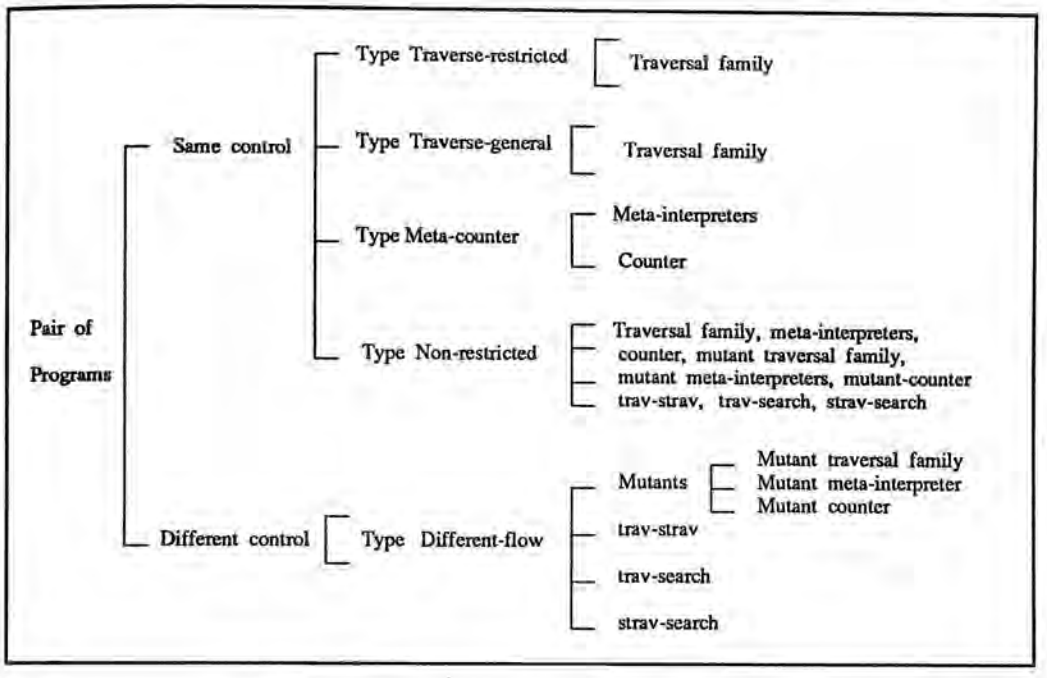

Figure 5.2: Classification of pairs of Prolog programs

pattern used to compose or decompose the data structure should be the same in each program.

Figure 5.3 shows schematically the characteristics which programs classified as type Traverse-restricted or Traverse-general hold.

- Pairs of type *Meta-counter* are programs which are either constructed using the meta-interpreter or counter skeleton. Both have the same number of clauses; both programs (taken clause by clause) have the trivial condition or in both clauses the `test` should be identical. Also these programs traverse just one data structure of a given type (for instance lists or trees). Finally the pattern used to compose or decompose the data structure should be the same in each program (see Figure 5.4).

- Pairs of type *Different-flow* are programs which have slightly modified their flow of control with respect to the skeleton from which they are derived. They have different number of clauses; both programs (taken clause by clause) have the trivial condition or in both clauses the `test` should be identical. Each program in the pair

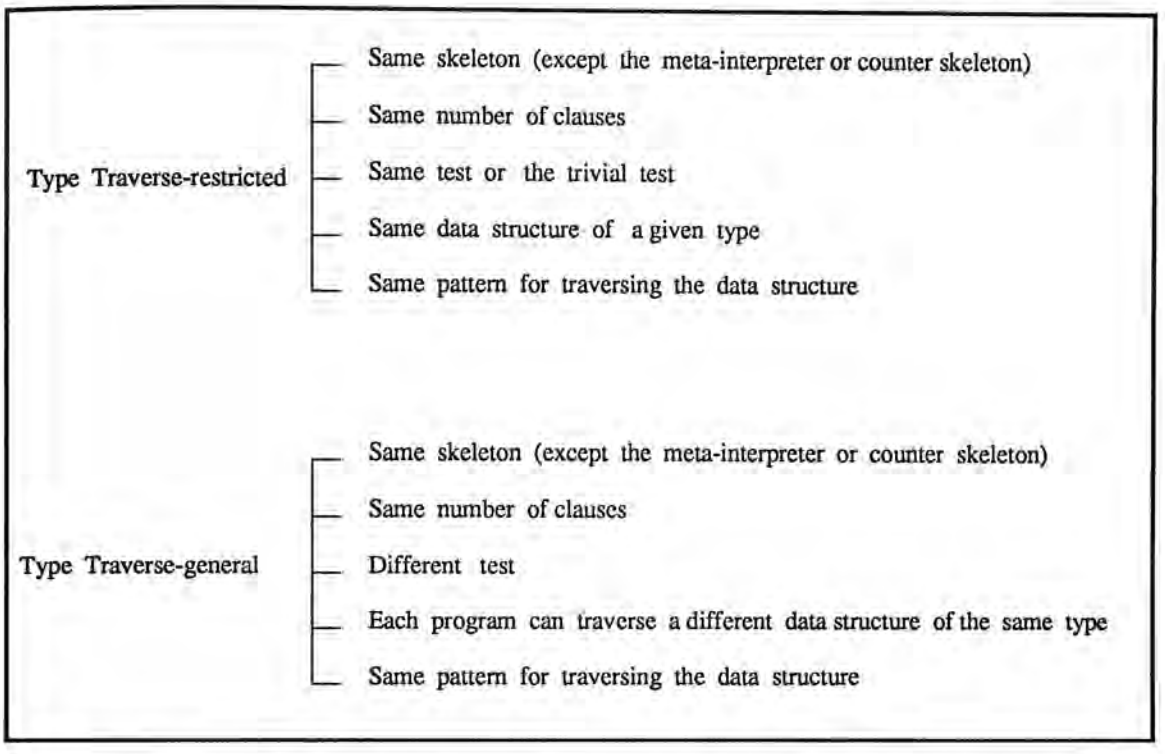

Figure 5.3: Properties of Type Traverse-restricted and Traverse-general

traverses just one data structure of a given type (for instance lists or trees). Finally the pattern used to compose or decompose the data structure should be the same in each program (see Figure 5.4). In this type are grouped mutants of the traversal family, meta-interpreter mutant, counter mutant, pair trav-strav, pair trav-search and pair strav-search.

- The *trav-strav* pairs of programs are those where one is constructed using the traverse skeleton and the other is built using the short_traverse skeleton. Both can have different number of clauses; each program in the pair has a different test (different to the trivial condition) and each program can traverse a different data structure of a same type. The pattern used to compose or decompose the data structure should be the same in each program.
- The *trav-search* pairs of programs are those where one is constructed using the traverse skeleton and the other is built using the search skeleton. This pair

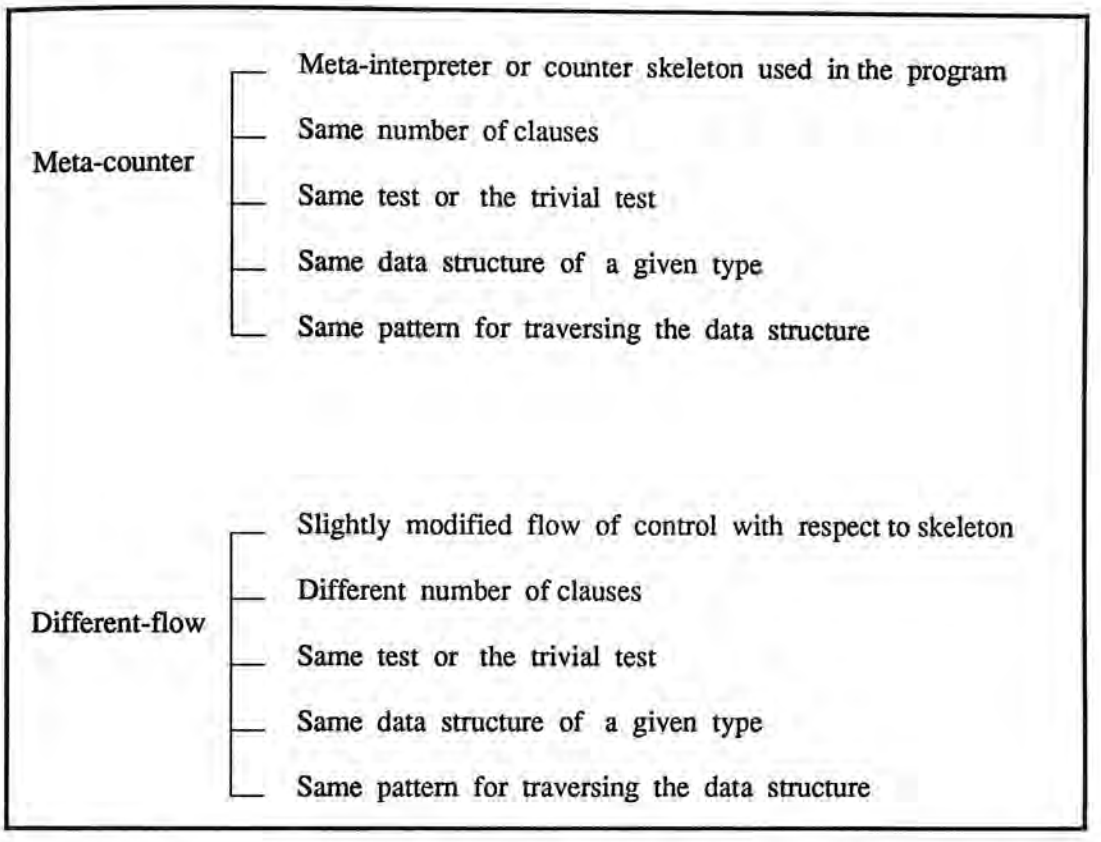

Figure 5.4: Properties of Type Meta-counter and Different-flow

holds the characterists of the pair trav-strav.

- The *strav-search* pairs of programs are those where one is constructed using the short_traverse skeleton and the other is built using the search skeleton. In a similar fashion this pair holds the characteristics given for pair trav-strav.

- Pairs of type *Non-restricted* are those programs which can be derived from the same skeleton or from different skeleton, both have the same or different number of clauses; both programs (taken clause by clause) can have zero or different tests. Also these programs traverse one or more data structures of a given type or different type. Finally the pattern used to compose or decompose the data structure can be the same or different in each program.

  Figure 5.5 shows schematically the set of properties of the type Non-restricted.

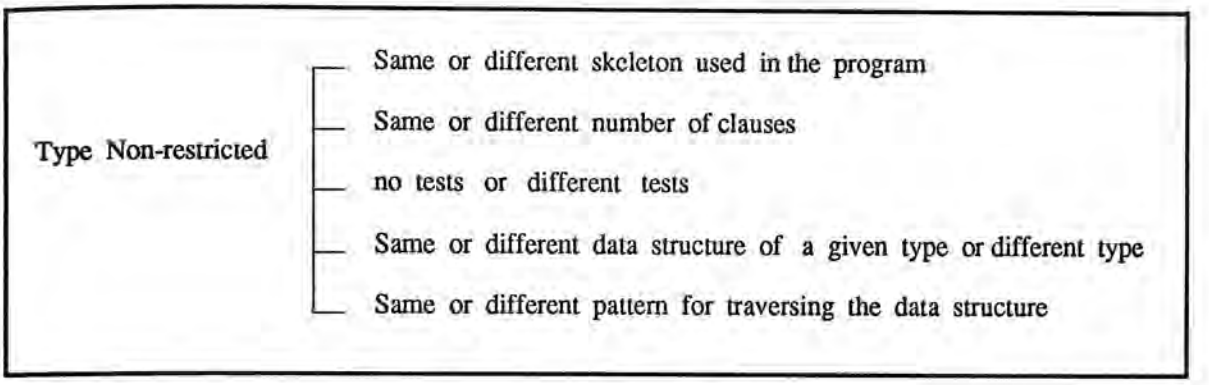

Figure 5.5: Properties of Type Non-restricted

5.2 Program History

The message of our composition system is that when meta-information is attached to code in a machine readable format then this meta-information can be very useful for automatic program writers; provided that the meta-information includes knowledge of standard techniques (developed by experts as an efficient controllable way to write programs).

A program history may be thought of as "meta-information" on the code. The composition system can read this meta-information and use it to guide the combination. This meta-information is used to infer the user's intentions in requesting the program combination, and so deal with cases of underspecification. So the use of program history is a powerful way to render program combination more effective. Our example on page 104 shows how the meta-information can be used to infer likely user intentions. The system can guess that users might wanted to synchronise the data structure by using the program history (i.e. if the pair of programs are classified as belonging to type Traverse-restricted then synchronisation of the data structure can be performed). This is one way to infer the user's intentions. Note that this might not be easily performed in approaches which only use standard transformation techniques for combining programs.

Also the machine readable high level meta-information contains the information necessary to classify the program by the classification defined in section 5.1, and hence select the

correct combination method. Our classification of programs relies on the skeleton used in the construction of the program. The information recorded in the program history can be obtained during the construction of the program by using the techniques editor.

In the following example we show how the program history could be obtained from the techniques editor. As mentioned on page 39, Robertson's editor provides menus allowing the user to select a skeleton and a set of techniques for the construction of his program. At this stage we can record key stages in the program development. Some of the information can be obtained straightforwardly. Examples of easily obtained information are: the name of program, the type of program (by looking in our classification of programs), the arity of the program (number of arguments), the name of the skeleton and the number of clauses.

However, it is more complicated to obtain other information such as how the subgoals in the body of the skeleton were transformed into the program (in order to obtain this information, we would compare each clause in our skeleton against each corresponding clause in the program). To be specific, suppose that the user wants to build the program `sum/2`. Let us assume that he choses the traverse skeleton from the menu offered in the techniques editor. This skeleton is:

```
traverse([]).
traverse([H|T]) :-
          traverse(T).
```

By renaming the predicate and applying the sum technique, which sums the value of an argument, we obtain the following program.

```
sum([],0).
sum([H|T],Sum) :-
          sum(T,S1),
          Sum is S1 + H.
```

At this stage we need to record the set of techniques that were used in the construction of the program.

Once we have the program, we also need to record how the subgoals in the skeleton were transformed. This is done by comparing each clause in the skeleton with its corresponding

clause in the program. For this particular example we have that the body of the first clause is equal to `true` and is (trivially) transformed into the subgoal `true`. In the second clause we have that `traverse(T)` has been enhanced to give `sum(T,S1)`. Also, we record that this program has the same flow of control as the skeleton used in its construction.

In a similar fashion we can obtain the features of the program `prod/2`.

```
prod([],1).
prod([H|T],Prod) :-
          prod(T,Prod1),
          Prod is Prod1 * H .
```

Furthermore the program history for the combined program (shown below) can be derived from the histories associated with each program that went into the combination and by using user interaction.

```
sum_prod([],0,1).
sum_prod([A|B],C,D) :-
           sum_prod(B,E,F),
           C is E + A,
           D is F * A.
```

For our example, knowing both the history for `sum/2` and for `prod/2` allows us to derive the program history for the combined program `sum_prod/3`. In particular, the combined history contains the type, which is determined in accordance with the type associated to `sum/2` and `prod/2` (see rules defined in section 5.2.1), the skeleton which is determined by analysing which skeletons are used in program `sum/2` and `prod/2`, and the techniques used in `sum_prod/3` are basically the list of techniques used in both programs. On a clause by clause basis we can also find how the subgoals in the skeleton were transformed for the program `sum_prod`. Finally, the name of the combined program is obtained from the join specification given by the user; and by user interaction we get the number of clauses, the arity and the number of tests.

The program history is defined using the relations *his_prog/9* defined as follows:

$$his\_prog(P, T, A, S, N, TsT, Tec, NM, M).$$

where P is the name of the program, T is the type of the program (one of the types in our classification), A is the arity, S is the name of the skeleton, N is the number of clauses for the program, TsT is the number of tests (each of these tests determine one recursive case), Tec indicates which techniques were used in the program, NM is a list containing the set of non-mutant clauses and M is a list containing the set of mutant clauses.

The NM argument is a list recording, for each clause, how the subgoals in the skeleton were transformed and also the set of tests for each clause. This means that the argument NM contains the structural differences between the program and the skeleton used in the the program.

In a similar fashion the list of mutant clauses M is recorded in the program history.

Knowledge of the relation between the subgoals in the program and the subgoals in the skeleton is obtained through the *key points* which are defined in the program history. A *key point* takes a subgoal (G_i) in a clause of the skeleton and finds the corresponding transformed subgoal in a program P, which is an extension of the same skeleton. The subgoal can be found in two ways: as an enhancement (more arguments, different name), or as the same subgoal (same name, same number of arguments).

For instance, consider the following set of relations recorded in the program history concerning to the programs `sum/2` and `prod/2` defined above.

```
his_prog(sum,type_traverse,2,traverse,2,0, sum_technique,
          [[1,true,true,no_test],[2,trav(H),sum(T,S1)]],nil).
his_prog(prod,type_traverse,2,traverse,2,0,no,
          [[1,(true,true),no_test],[2,trav(H),prod(T,P1)]],nil).
```

The meanings of each of the nine arguments in the history are as follows:

1. says the name of the programs (i.e. `sum/2` and `prod/2`),

2. states that both programs belong to the traverse type,

3. says that both programs have arity 2,

4. indicates that both were constructed using the traverse skeleton,

5. states that both have 2 clauses,

6. indicates that both programs were constructed without using any test,

7. indicates that the technique *sum* was applied to the program `sum/2` so the seventh argument is *sum_technique* and for the program `prod/2`, neither the sum nor count technique was used, so the argument is `no`,

8. states how the initial subgoals in the skeleton were transformed in each program,

9. shows the list of mutant clauses. This list has as an argument the empty list (nil) because none of these programs is a mutant of the traverse skeleton.

For each program in our composition system, its program history is recorded as the relation *his_prog/9*. Also, the new knowledge concerning the generated combined program is automatically stored by the system for future stages in the combination process. The new history is obtained from the characteristics of the initial pair of programs (which are recorded in their program histories) and also by user interaction. For instance, the type for the combined program `sum_prod/3` is derived from the fact that both programs (`sum/2` and `prod/2`) have type_traverse. Then the combined program `sum_/3` is type_traverse (see the set of *type criteria* rules defined in Section 5.2.1).

The program history for the program `sum_prod/3` is as follows:

```
his_prog(sum_prod,type_traverse,3,traverse,2,0,sum_technique,
          [[1,(true,true),no_test],[2,trav(H),sum_prod(T,Sum1,P1)]]), nil).
```

In summary we can say that addition of "meta-information" to the code in the form here of the *program history* can be used very effectively to guide the combination process. In

this way the output can be achieved with less need for user direction and is more likely to conform to the user intentions (as we demonstrate later). There is also the possibility that if the process still needs user guidance then the system can use the program history to ask questions of the user in the language of skeletons and techniques rather than low level questions about how to transform the program such as which subgoals can be unfolded, etc. However this type of dialogue demands research outside the scope of this thesis.

5.2.1 Type criteria

The assignment of the type for the combined program is performed using rules similar to the examples given below. This assignation for the combined program is guaranteed by the properties defined in our set of properties in Chapter 9. The notation which was used in the following set of rules is as follows: (A, A) where the first argument in the pair is the type of program P_1 and the second argument is the type of program P_2.

1. $(A, A) \longrightarrow A$
2. $(A, B) \longrightarrow B$ where neither A or B are of type *mutant*.
   - $(traverse, search) \longrightarrow search$
   - $(traverse, short\_traverse) \longrightarrow short\_traverse$
   - $(short\_traverse, search) \longrightarrow search$
3. $(A, mutant_A) \longrightarrow mutant_A$
4. $(mutant_A, A) \longrightarrow mutant_A$

5.3 The Set of the Combination Methods

Our approach to the composition problem was to create a method for combining each class described in Section 5.1. Another way in which the problem can be addressed is to create

a single method which embodies the characteristics of each of our methods, for example a single method using unfolding, meta-folding, arithmetic rules and knowledge about the program. However given our identification of key classes of programs, we felt more natural to build the system in a "classification and combine" style, although it might possible to integrate into a single, all encompassing algorithm at later stage. We address this as future research in Chapter 10.

In chapters 6 and 7 we compare the performance of different combination methods on each pair of programs which can be derived from the classes devised in Section 5.1. Some methods give better results on particular classes of programs and under certain constraints. Factors taken into consideration in deciding whether or not a combined program is efficient include the number of times that the program needs to scan the data structures, and the compactness of the combined program. For discussion of how we rank the methods for efficiency see Section 6.3.

The description of our set of methods is presented in the order of complexity of each method. The applicability of each of our methods relies on the following general restrictions:

- The pair of programs to be combined is written using pure Prolog (i.e. no side-effecting predicates or cuts).
- The initial pair of programs must have been created using a techniques editor which is capable of producing the necessary history information.
- Programs created using skeletons in which a new flow of control can be passed as an argument are not handled in our current implementation.

5.3.0.1 Methods for Combining Programs with the Same Flow of Control

The first group is divided again into two cases: the methods which use the join specification shown in *CASE A* and the methods which use the join specification presented in *CASE B.*

- CASE A. $T(\vec{I_P}, \vec{I_Q}, \vec{O_P}, \vec{O_Q}) \Leftarrow P(\vec{I_P}, \vec{O_P}), Q(\vec{I_Q}, \vec{O_Q})$.
- CASE B. $T(\vec{I_P}, \vec{I_Q}, \vec{O_T}) \Leftarrow P(\vec{I_P}, \vec{O_P}), Q(\vec{I_Q}, \vec{O_Q}), F(\vec{O_P}, \vec{O_Q}, \vec{O_T})$.

where $\vec{I_P}, \vec{I_Q}$ are vectors of distinct input variables.

where $\vec{O_P}, \vec{O_Q}, \vec{O_T}$ are vectors of distinct output variables.

where F is a predicate that produces as output the vector $\vec{O_T}$ using the values obtained in procedure P and in procedure Q.

Note that in our join specification or user specification (both defined in Chapter 2), the programs are called in a conjunctive form (only using and) for the following two reasons: 1). programs are mostly written with and, 2). and is commutative (if there is not dependence between variables, i.e. if a variable is instantiated later, it cannot be moved around and placed before). So, subgoals can be moved around in order to obtain an efficient program.

If programs are called in a disjunctive form (in the join specification) the system will be required to perform a lot of transformations using De Morgan's laws in order to apply the folding operation. (i.e. if in the body of the clause we have both and and or it is much harder to move terms together to allow subgoals be folded). The use of the program history is still valid for the case when the join specification is defined as follows:

$$T(\vec{I_P}, \vec{I_Q}, \vec{O_P}, \vec{O_Q}) \Leftarrow P(\vec{I_P}, \vec{O_P}) \; ; \; Q(\vec{I_Q}, \vec{O_Q}).$$

However, the efficiency of the combined program is reduced drastically by allowing or in the join specification because after De Morgan's laws are applied the bodies of some clauses are drastically increased by adding all the possibles combinations of terms which are conjunctions of disjunctions. Therefore through this thesis we only consider join specifications using and between its operands. For further details see example defined in Appendix G.

However, we are limited to conjunctive join specifications: disjunctive ones can always be converted to a number of conjunctive ones. e.g.

$$T(\vec{I_P}, \vec{I_Q}, \vec{O_P}, \vec{O_Q}) \Leftarrow P(\vec{I_P}, \vec{O_P}) \; ; \; Q(\vec{I_Q}, \vec{O_Q}) \equiv \begin{array}{l} T(\vec{I_P}, \vec{I_Q}, \vec{O_P}, \vec{O_Q}) \Leftarrow P(\vec{I_P}, \vec{O_P}) \\ T(\vec{I_P}, \vec{I_Q}, \vec{O_P}, \vec{O_Q}) \Leftarrow Q(\vec{I_Q}, \vec{O_Q}) \end{array}$$

CASE A

The set of methods for combining programs by using the join specification of *CASE A* are shown below.

1. the *synchronization* method,
2. the *join 1-1* method,
3. the *procedural-join* method and
4. the *meta-composition* method
5. the *DS* method

CASE B

In *case B* we have two methods, one of which is called *particular* and the other the *general* method. The *particular method* is an extension to Burstall's work, which was developed for functional languages. The *general method* is an extension to procedural join developed by Lakhotia and Sterling [Lakhotia & Sterling 87]. Note that, for purposes of clarity, we have separated the methods into two independent sets A and B. The set of methods in A operate only over a pair of programs, whilst the set of methods in B also exploit the predicate F (which appears after the operands of the join specification) in order to obtain further optimisations.

5.3.0.2 Method for Combining Programs with Different Flow of Control

The *mutant* method is used in the combination of programs belonging to the same class but with slightly different flows of control. This method is one of the contributions of this work, and can be applied because we have access to the program history. An example of

the use of this method appears on page 161. In this example `count/2` is created using the *traverse* skeleton formed by two clauses and program `get_odd/2` is a mutant of the same skeleton.

The *traverse-short_traverse* method (Trav-Strav) combines programs created by using the *traverse* and *short_traverse* skeletons. These programs are combined using procedural join because both skeletons have the following restrictions: the only difference between the skeletons *traverse* and *short_traverse* is the base case; the base case for the traverse skeleton ensures that the entire list will always be processed and in short_traverse the execution will either traverse the entire list or stop when a condition has been met.

The *traverse-search* method (Trav-Search) combines programs created using the *traverse* and *search* skeleton. These kinds of programs also can be combined using the procedural join method.

The *short_traverse-search* (Strav-Search) is the same case as the *traverse-search* above.

Figure 5.6 and Figure 5.7 summarise the set of methods for combining the two types of program (same and different flow of control). The importance of classifying programs into the classes defined in Section 5.1 and restricting the transformations over each method are of particular importance because we can combine programs efficiently by applying the most efficient method for each class.

Note that Join 1-1 and Procedural join were developed by Lakhotia and Sterling and we have re-implemented them using the notion of program history. Our re-implemented versions of those methods still make use the unfold and fold transformation, but now, by using the program history, it is possible to check that the arguments intended to provide the flow of control are compatible in the combination and to perform the synchronisation of the data structures. All the other methods in this thesis were developed by us in order to widen the range of programs that can be combined.

Some methods in our composition system enforce synchronised traversal of a shared data structure. This is a way to kill unwanted solutions due to unwanted backtracking (without

Figure 5.6: Methods for Combining Prolog Programs with the Same Flow of Control

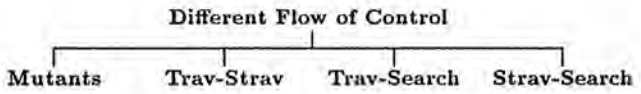

Figure 5.7: Methods for Combining Programs with Different Flow of Control

having to resort to cuts). In fact to exclude the unwanted solutions can be difficult (if not impossible) to achieve by defining a join specification as a Prolog program and applying the unfold/fold operations to it. A full description of each method is given in the next two chapters.

5.4 Automatic Selection of the Combining Method

This section describes work that we have done on the automatic selection of the combining method. We have designed and implemented an algorithm called the *selection procedure*, which decides which method can be used in the combination of a pair of programs. In our approach, the knowledge about the program history plays a major role in this selection. The automatic selection is of particular importance to the user level, since the user does not need to worry about deciding which is the best method by performing program analysis on the pair of programs.

The selection stage requires information concerning the skeletons and techniques which were used in the construction of both of the programs. This information is taken from

the program history. Therefore the main decisions rely on the program history and do not require user interaction. The questions required to be answered by the user are only the name and arity of the top level predicates which will be combined, and the definition of the join specification. In the definition of the join specification the user needs to be aware of the dependency between input/output vectors of variables from the programs to be combined (see Chapter 9). The only exception to this rule is our *mutant* method which is semi-automatic. This method allows the combination of several families of mutant programs. For this method the offered combined clause can be accepted/rejected by the user. The join specification gives an indication of a user's intentions. However the user must take the final decision to ensure that the proposed mutant combined clause matches with his intentions. Nevertheless the proposed combined clause is correct according to the join specification. If the combined clause suggested by the system is rejected then the responsibility lies with the user to provide an alternative.

The features taken into consideration in selecting the combination method are:

1. *the skeleton employed.* The skeleton used in the program must be any of the skeletons defined in our knowledge base of skeletons.

2. the number of clauses in each program. If the programs do not have the same number of clauses, then the selected combination method must combine adequately the extra clauses.

3. the tests. If corresponding clauses of the programs differ in one or more tests, then the combination method must consider exceptions for each of these tests.

4. the *data structure* used in the program. More efficient combination can be achieved by getting rid of spurious (repeated) variables — this is only possible if it is known what data structure those variables stand for.

5. the *pattern* used to recurse up or down the data structure. The removal of spurious variables with the same data structure is only possible if they have the same pattern of recursion.

6. the *techniques* applied to build the program, for instance if the technique *sum* or *count* were used in the program. By using this parameter a more efficient combined program can be obtained.

The features 1 to 5 give information concerning to the flow of control which is used in the program and the feature 6 gives data information (provided by techniques). In Chapter 6 and Chapter 7 we will describe a set of methods which takes advantage of the techniques knowledge for getting a more optimised combined program.

The system classifies a given pair of programs by whether it belongs to type Traverse-restricted, type Traverse-general, type Meta-counter, Non-restricted or Different-flow (if the pair belongs to some category defined for programs with different flow of control). Once the system identifies the type of the pair of programs by using information recorded in the program history, then the system finds information regarding the techniques used in each program (also contained in the program history) and finally selects the combining method. This selection is performed using the set of rules defined below. These rules use the features (1-6) defined above.

The set of rules used in the selection are given below. Note that these are not the actual rules used in our system, which need to access the formal program history. These are English paraphrases of the actual definitions.

RULE 1:
if
both programs are not required to be derived from the same skeleton,
both are not required to have the same number of clauses,
both programs are allowed to have zero or different tests,
the pattern used to construct/deconstruct the data structure is not required to be the same in each program and
there is a dependence between variables (the input for program Q is the output of program P)
then
the method is: synchronization.

RULE 2:

if

both programs are derived from the same skeleton (traverse, short_traverse or search skeleton),
both have the same number of clauses,
both programs are allowed to have zero or the same tests,
both traverse the same data structure and
the pattern used to construct/deconstruct the data structure must be the same in each program

then

the method is: join 1-1.

RULE 3:

if

both programs are derived from the same skeleton (traverse, short_traverse or search skeleton),
both have the same number of clauses,
both programs are allowed to have different tests,
both traverse data structures of a given type and
the pattern used to construct/deconstruct the data structure must be the same in each program

then

the method is: procedural join.

RULE 4:

if

both programs are derived from the same skeleton (traverse, short_traverse or search skeleton),
both have the same number of clauses,
both are allowed to have zero or different tests,
both traverse data structures of a given type,
the pattern used to construct/deconstruct the data structure must be the same in each program and
both programs were constructed using the *count* or *sum* technique

then

the method is: particular.

RULE 5:

if

both programs are derived from the same skeleton (traverse, short_traverse or search skeleton),
both have the same number of clauses,
both are allowed to have different tests,
both traverse data structures of a given type,
the pattern used to construct/deconstruct the data structure must be the same in each program and
Only one programs was constructed using the *count* or *sum* technique

then

the method is: general.

RULE 6:

if

both programs are derived from the meta-interpreter or count skeleton,
both have the same number of clauses,
both programs are allowed to have zero or the same tests,
both traverse data structures of a given type,
the pattern used to construct/deconstruct the data structure must be the same in each program

then

the method is: meta-composition.

RULE 7:

if

both programs are derived from the same skeleton (traverse, short_traverse or search skeleton),
both have the same number of clauses,
both are allowed to have different tests,
one of the programs works over a different data structure than the other program and
the pattern used to construct/deconstruct each data structure must be the same in each program

then

the method is: DS.

RULE 8:

if

both programs are mutants of the same class or
one of them is mutant (same class)
each program is allowed to have different number of clauses,
both programs must have the same test for the corresponding clauses,
both traverse data structures of a given type and
the pattern used to construct/deconstruct the data structure must be the same in each program

then

the method is: mutant method

RULE 9:

if

one program is constructed by using the traverse skeleton and
the other is built by using the short_traverse skeleton
both have the same number of clauses,
both programs are allowed to have different tests,
both traverse data structures of a given type and
the pattern used to construct/deconstruct the data structure must be the same in each program

then

the method is: procedural join

RULE 10:

if

one program is constructed by using the traverse skeleton and
the other is built by using the search skeleton
both have the same number of clauses,
both programs are allowed to have different tests,
both traverse data structures of a given type and
the pattern used to construct/deconstruct the data structure must be the same in each program

then

the method is: procedural join

RULE 11:

if

one program is constructed by using the short_traverse skeleton and
the other is built by using the search skeleton
both have the same number of clauses,
both programs are allowed to have different tests,
both traverse data structures of a given type and
the pattern used to construct/deconstruct the data structure must be the same in each program

then

the method is: procedural join

A decision tree containing our selection rules is shown in Figure 5.8, Figure 5.9 and Figure 5.10. Note that we have divided it into in 3 figures for reasons of clarity.

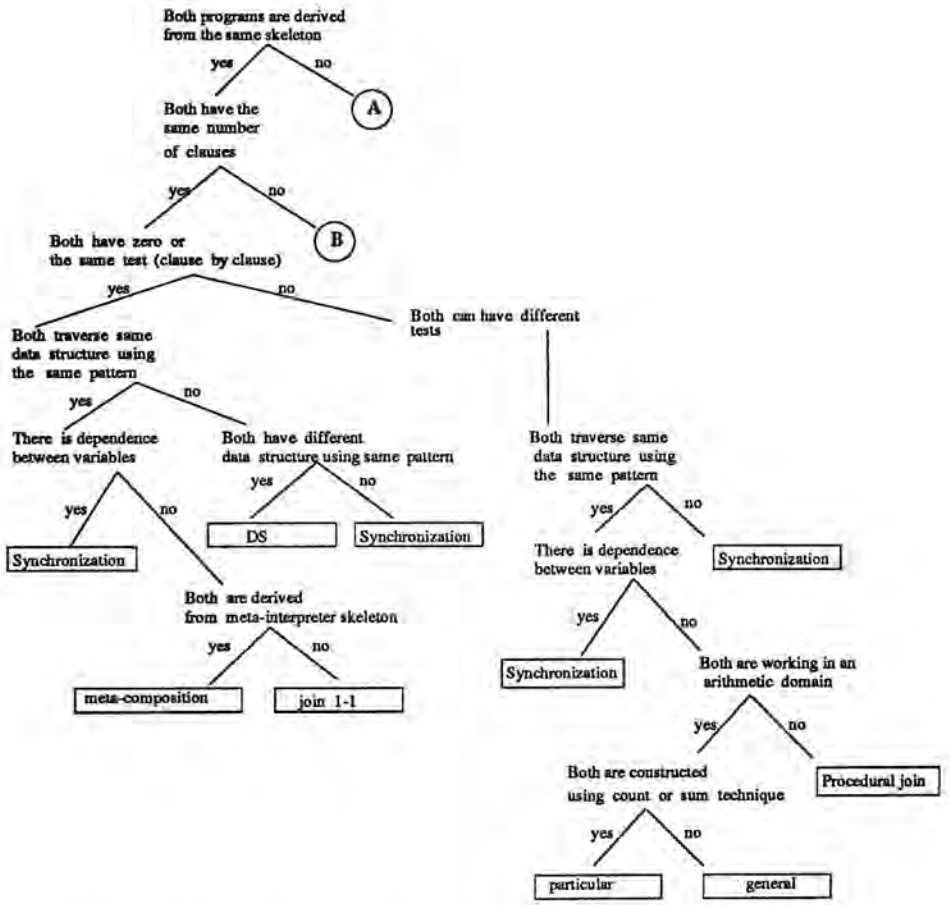

Figure 5.8: A Decision Tree based on Features of Programs

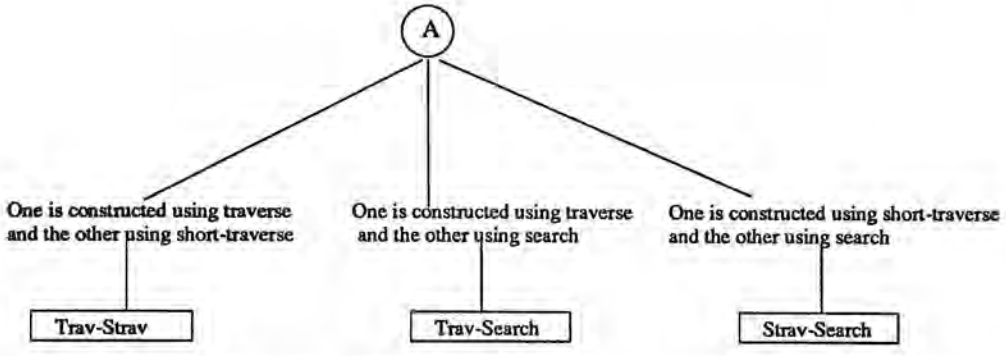

Figure 5.9: Decision Tree based on Features of Programs (Part A)

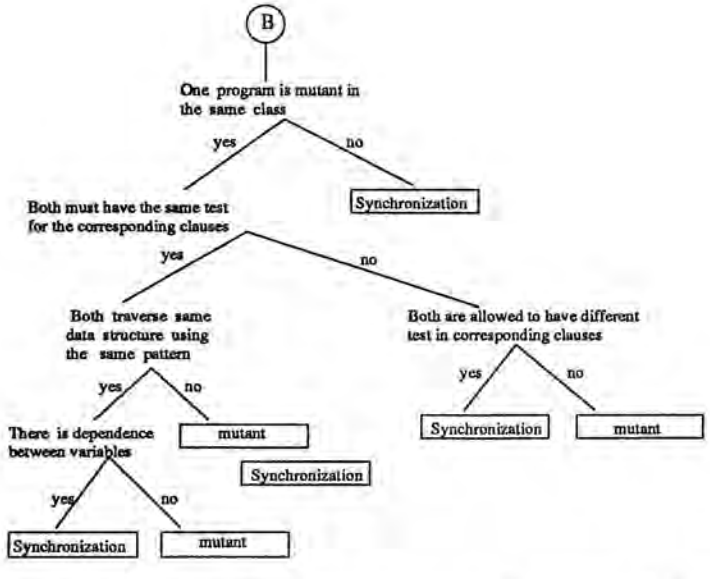

Figure 5.10: Decision Tree based on Features of Programs (Part B)

5.5 Conclusions

The contributions of this chapter are twofold. Firstly, that the classification in terms of control flows and techniques was useful in that it allowed us to develop methods for each class separately. These methods were far more powerful, and required less user direction, than was possible for methods designed to work with all programs. Secondly, that knowl-

edge about the program can be used to automatically select the composition method. Such automatic selection is of particular importance at the user level.

6

Methods for the Same Flow of Control

This chapter presents the set of methods for combining programs with the same flow of control implemented in our composition system. We also discuss the performance of each of the combining methods by analysing their behaviour for each pair of programs which can be derived from our hierarchy of program classes.

6.1 General Conditions

In order to apply the methods, various conditions must hold. However join 1-1, procedural join, meta-composition, particular and general method assume that the following conditions are always satisfied.

- both have the same number of clauses,
- the pattern used to construct/deconstruct the data structure should be the same in each program
- both traverse the same type of data structure.

Besides these we might have (in some cases) extra conditions (denoted by ϕ_i)

6.2 Computational Cost for the Combined Program

If the programs being combined have the following properties:

- The programs share a common abstraction of flow of control determined by the skeleton used in their construction (S_1).
- For the two specializations of S, called i_1 and i_2 (extensions), each specialization computes different values over the same data structure.

Then we can get greater efficiency by transforming them into a single fully recursive program. The reason being that if we run the initial pair we will twice incur the cost of traversal. That is, the final cost of running the pair of programs separately is twice the cost of traversal plus the costs of each computed value in each program, as follows:

$$2ct(S_1) + \sum_{i=1}^{n} ft(i_1) + \sum_{j=1}^{m} ft(i_2)$$

where,

$ct(S_1)$ is the cost of execution of the flow of control (the cost associated with recursing up or down the data structure)

$ft(i_1)$ is the cost associated with the computing a value in the specialization i_1.

$ft(i_2)$ is the cost of computing a value in the specialization i_2.

On the other hand the cost associated with the combined program is defined below.

$$ct(S_1) + \sum_{i=1}^{n} ft(i_1) + \sum_{j=1}^{m} ft(i_2)$$

In the combined program the final cost is that of recursing just once down the data structure, along with the cost of each computed value in each program.

6.3 Efficiency Estimation

The possible answers after applying a method are defined below. The answers include an estimate of the efficiency of the combined program. However, it is generally difficult to give objectives measures of efficiency which are always applicable to classes of programs. Hence, the measures used in each table are relative comparisons based on what we believe to be typical cases. The real measure of the expected efficiency of the combined program would depend on many other factors, such as the distribution of the data used in the query, and details of the compiler/interpreter.

In making our estimate of the efficiency the main factor we take into account is the amount of duplication of effort made in the program. For example, the program should minimise the number of times a data structure is traversed, or tests are applied.

Another factor is the compactness of the combined program: we prefer to generate more compact code, because a large number of clauses can adversely affect efficiency (due to excess memory requirements), also if there are a large number of clauses per predicate then the time to select the appropriate one can be increased. It is not uncommon for a lower efficiency ranking to arise because the combined program requires the auxiliary use of the original programs, thus increasing the overall number of clauses in the combined program.

These are not always the best guides to efficiency but at least provide useful landmarks. Thus, the metrics which appear in our tables are defined as follows:

"blank" Whenever there is no entry in the table then we mean that the pair of programs cannot be combined reliably

α "Correct and efficient." This means that the combined program will not only be correct but will also be fully recursive, hence the program will not needlessly duplicate the traversal of the data structure.

β "Correct but inefficient." The combined program is correct but not fully recursive, because it still contains the original programs. The inefficiency arises because the

presence of the original programs means the combined program is much larger than for the α case. Also, there can be some duplication of effort when traversing data structures.

γ "Correct but very inefficient." Means that we totally rely on the original programs. Each program does its own traversal of the data structure thus always duplicating effort.

Note that in practise we would expect the difference between the α and β cases to be much smaller than between the β and γ cases.

6.4 Notation

The tables presented in each combining method show the performance of the method (subject to certain extra applicability conditions). The entries in each table (showing the performance of each method) are defined in terms of the type of program P_1 (row) and the type of the program P_2 (column). Also the columns and rows are labelled using the notation defined in Figure 6.1:

Skeleton	
t	traverse
st	short-traverse
s	search
ctr	counter
$meta$	meta-interpreters
$mut - t$	mutant of the traverse
$mut - st$	mutant of the short-traverse
$mut - s$	mutant of the search
$mut - ctr$	mutant of the counter
$mut - meta$	mutant of the meta-interpreter

Figure 6.1: Table Notation

6.5 The Synchronization Method

The *synchronization* method allows the combination of all the pairs of programs using the most primitive form of combination, which is simply to add them as subgoals in the definition of the combined predicate. If the combined predicate is named T then we have:

$$T(\vec{I_P}, \vec{I_Q}, \vec{O_P}, \vec{O_Q}) \ :- \ P(\vec{I_P}, \vec{O_P}), Q(\vec{I_Q}, \vec{O_Q}).$$

Note that in this case our join specification corresponds directly to a Prolog program.

This method does not impose any restriction on the class to which each of the programs must belong. The combined program obtained using this method is not a full recursive program in the sense that it simply "sticks the programs end-to-end" rather than merging them within a recursive definition. This method is very general but is useful only for the combination of programs which cannot be combined using any of the other methods. However there are special cases of programs which we are combining by using this method. For example, programs P and Q in which the input for program Q is the complete output of program P and hence P must finish execution before Q begins. The combined predicate T is shown as follows:

$$T(\vec{I_P}, \vec{O_P}, \vec{O_Q}) \ :- \ P(\vec{I_P}, \vec{O_P}), Q(\vec{O_P}, \vec{O_Q}).$$

where the input vectors of variables are $\vec{I_P}$ and $\vec{I_Q}$ and, similarly; $\vec{O_P}$ and $\vec{O_Q}$ are the output vectors of variables. We classify this as a "very inefficient" form of combination because each of the programs P and Q must be executed separately, and so we are very likely to duplicate the effort involved in scanning the data structures.

For instance consider the case when our initial programs are the meta-interpreters `solve/2` and `interpret/1` which are constructed by using the skeleton `solve/1` defined in [Sterling & Shapiro 86]. The meta-interpreter `solve/2` proves that `Goal` is true with

respect to the program being interpreted and the meta-interpreter `interpret/1` explains the proof for the same `Goal` defined on Page 129. Note that this example corresponds to the (meta,meta) entry in the table 6.1.

A new program `how/2` which solves a goal (query) and interprets at the same time is required by using the join specification defined as follows:

```
how(Goal,Proof) ⇐ solve(Goal,Proof), interpret(Proof).
```

The code for `solve/2` is shown below and the code for the meta-interpreter `interpret/1` is defined on page 129.

```
solve(true,true).
solve((A,B),(ProofA,ProofB)) :-
            solve(A,ProofA), solve(B,ProofB).
solve(A,(A :- Proof)) :-
            clause(A,B), solve(B,Proof).
```

In this example we have the case that there is a dependence between vectors of variables in program `solve/2` and `interpret/1`. The program `interpret/1` needs to have the complete proof tree for a given goal in order to explain how it was proved. Therefore, this pair of programs cannot be combined (in our composition system) into a single program `how/2`. They are combined by using the synchronization method which does not produce a full recursive program.

The table 6.1 shows the performance of the synchronization method (see section 6.3 for an explanation of the term γ).

The synchronization method does not impose any of the restrictions defined in Section 6.1. This means that both programs do not need to have the same number of clauses, the pattern used to construct/deconstruct the data structure does not need to be the same in each program, and also both programs do not need to traverse the same type of data structure. The set of programs which can be combined are those which our system classify as type Non-restricted defined in Chapter 5.

Synchronization (M_1)										
P_1/P_2	t	st	s	ctr	meta	mut-t	mut-st	mut-s	mut-ctr	mut-meta
t	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ
st	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ
s	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ
tr	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ
meta	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ
mut-t	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ
mut-st	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ
mut-s	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ
mut-tr	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ
mut-meta	γ	γ	γ	γ	γ	γ	γ	γ	γ	γ

Table 6.1: Table for the Synchronization Method

As we discussed above, this method is very general but gives an inefficient combined program. Consequently, none of the entries in the table are blank, however they all get only a γ, meaning that the combined program is correct but very inefficient. This problem of inefficiency forces us to look for better methods which can generate a more efficient combined program for some of the entries in the table.

6.6 The Join 1-1 Method

Join 1-1 was developed by Lakhotia et al. [Lakhotia & Sterling 87]. We have re-implemented it for combining Prolog programs belonging to type Traverse-restricted (defined in Chapter 5). The join 1-1 of program P and Q is the set of clauses obtained by combining all pairs of corresponding clauses of the two programs with respect to a join specification.

Since the skeletons of P and Q are identical (clause by clause) join 1-1 allows the combination the i^{th} clause from program P with the i^{th} clause from program Q. This restriction in the join operation has the advantage that the programs generated by join 1-1 are free of redundant clauses, because this method only allows joined clauses which operate on the same instance of the data structure in the induction parameter.

The *join 1-1* method is performed by doing unfolding and folding operations in corresponding clauses as was defined by Tamaki and Sato [Tamaki & Sato 84] but in a restricted way directed by the program histories. This method enforces the same traversal of the data structure for each clause (i.e. traverse the same element on the data structure), so only a subset of answers obtainable by using join 1-1 might be characterised. However, this narrowing of results is often precisely what users require when merging two programs over a shared flow of control (as we demonstrate in Section 6.6.1).

The algorithm for the join 1-1 is defined below. The notation used is as follows: $P_{i,head}$ is the head of the clause P_i, $P_{i,body}$ is the body of clause P_i, and mgu means the most general unifier.

Join 1-1 Algorithm

1. Define the join specification T defined as:

$$T \Leftarrow P, Q$$

   Note that this join specification is not a Prolog program.
2. Take a pair of clauses P_i and Q_i.
3. Create a template written as Prolog clause by using the information provided in the join specification:

$$T_i \text{ :- } P, Q$$

4. Unfold the calls P and Q in T_i with respect to P_i and Q_i:

   - if P and $P_{i,head}$ unify with a substitution θ_P.
   - and Q and $Q_{i,head}$ unify with a substitution θ_Q.

   Then replace P in T_i with $P_{i,Body}$, Q with $Q_{i,body}$ and apply the mgu $\theta_P\theta_Q$ to the clause produced. Then

   T_i :- $P_{i,body}, Q_{i,body}\theta_P\theta_Q$

   In the case that $P_{i,body}$ is a conjunction of n subgoals $P_{i,goal1}, ..., P_{i,goaln}$ with $n > 0$ and $Q_{i,body}$ is a conjunction of m subgoals $Q_{i,goal1}, ..., Q_{i,goalm}$ with $m > 0$ then the result of unfolding is the clause:

   T_i :- $(P_{i,goal1}, ..., P_{i,goaln}), (Q_{i,goal1}, ..., Q_{i,goalm})\theta_P\theta_Q$

   In other words the unfolding operation takes the body of the clause T and expands it using the body definitions of the clauses with heads which unify with P and Q.
5. Apply the *fold* operation if there exists a pair of subgoals $P_{i,goalp}$ and $Q_{i,goalr}$ in the body of the clause T_i such that $(P_{i,goalp}, Q_{i,goalr})$ and (P, Q) unify with some mgu θ

   Then replace the two subgoals $P_{i,goalp}$ and $Q_{i,goalr}$ with T_i.
6. Repeat this process for all clauses in procedure P and procedure Q.

6.6.1 Example 1

This example shows what we can do by using the knowledge about the program rather than using only the standard transformation operations. This method preserves the spirit of the program histories (and hence their functionalities) rather than the simplest logical alternative (which we would get with simple application of fold and unfold). We believe this is more likely to match user intentions (although we would expect the user to confirm this). Note that this example corresponds to the (s,s) entry in the table 6.2.

Let us now proceed to the example. Suppose we have two predicates:

`pos(List,Element,Pos)`:
finds `Element` in the `List` with position `Pos` (counting from the head of the list).

```
pos([X|_],X,1).
pos([_|T],X,N) :-
        pos(T,X,NP),
        N is NP+1.
```

`path(List,Element,Path)`:
finds `Element` in the `List` with path `Path` (meaning sequence of elements up to and including the `Element`).

```
path([X|_],X,[X]).
path([H|T],X,[H|R]) :-
        path(T,X,R).
```

Now suppose that the user requests that we combine these two programs. Most transformation systems will take the join specification to be the Prolog program

```
p_path1(L,X,P,LP) :-
        pos(L,X,P),
        path(L,X,LP).
```

and then convert this to the logically equivalent program

```
p_path1([X|_],X,1,[X]).
p_path1([X|T],X,1,[X|R]):-
        path(T,X,R).
p_path1([X|T],X,N,[X]) :-
        pos(T,X,NP),
        N is NP+1.
p_path1([H|T],X,N,[H|R]) :-
        p_path1(T,X,NP,R),
        N is NP+1.
```

In this case, if we give the query `p_path1([a,b,a,b],b,P,Path)` then Prolog will give four answers

`P = 2`	`P = 4`	`P = 2`	`P = 4`
`Path = [a,b]`	`Path = [a,b,a,b]`	`Path = [a,b,a,b]`	`Path = [a,b]`

In our opinion, this would usually not correspond to the users intentions. Instead, it seems more likely that the user would want to combine the functionalities of the programs to get

> `p_path2(List,Element,Pos,Path)`:
> finds `Element` in the `List` with position `Pos`, and path `Path`.

in which case the last two solutions would be unwanted, because they correspond to finding different copies of the same element in the list and returning the path for one but the position for the other. Systems that rely only on the join specification written in the Prolog form have no way to know that the user wanted exactly to find the position and path for the same `Element`. That is, we want to synchronise the list search performed in `pos/3` and `path/3`. There is no way to express this requirement directly in Prolog using only the given predicates.

However, our system does not rely on the Prolog form of the join specification, but effectively has an extended join specification which we can write as

```
p_path2(L,X,P,LP) ⇐
                pos(L,X,P),
                path(L,X,LP).
```

where the underlined arguments provide flows of control which we want to synchronise. Since we assume knowledge of the history of development of the program we can check which arguments were intended to provide the flow of control and assess whether they will be compatible in combination. In this case it will observe (from the program history) that both programs have the same flow of control, called "search"[1], and only differ in that one has a count technique added, and the other an accumulator technique. We assume that the user would also like the flow of control of the output to be "search" and would simply like to have both the count and accumulator techniques added. This gives the output program

[1]So-named because its function is to search for a particular element

```
p_path2([X|_],X,1,[X]).
p_path2([H|T],X,N,[H|R]) :-
            p_path2(T,X,NP,R),
            N is NP+1.
```

which will indeed only return the first two (desired) solutions and not the last two undesired solutions.

Let us look at this in a bit more detail. The flow of control for both `pos/3` and `path/3` is a form of list search, which can be described as:

```
search([H|_],H) :-
            t1(H).
search([H|T],X):-
            t2(H),
            search(T,X).
```

where `t1/1` and `t2/1` are tests. The corresponding simplified versions of the program histories for the programs `pos/3` and `path/3` are

```
his_prog(pos,3,search, [[1,(true,true),no_test],[2,(search(T,X),pos(T,X,NP)),no_test]],
        count).
his_prog(path,3,search, [[1,(true,true),no_test],[2,(search(T,X),path(T,X,R)),no_test]],
        accumulator).
```

As reminder to the reader we will describe each of the arguments in this simplified version of the program history. Note that we have deleted 4 arguments in these simplied versions of the program history. However a full description of the components recorded in the program history are described in Chapter 5. The first argument is the name of the program; the second is the arity; the third is the name of the initial control flow used in the construction of the program; the fourth argument is a list recording, for each clause, how the subgoals in the initial control flow were transformed; and the fifth is the technique used to extend the initial flow of control.

The program histories thus say that both programs were built starting from "search", that the test literals were trivially true, but that the techniques used were different. In this case, the composition system then selects the join 1-1 method that makes the combination

by combining only corresponding clauses, rather than combining all pairs of clauses, thus achieving the desired synchronisation in the unpacking of the lists.

6.6.2 Example 2

As contrast to the previous example, we now consider a case where combination using join 1-1 gives the same set of final results as combination performing all the possible combinations of clauses (i.e. performing the full cartesian product). The notation used is as follows: $P_{i,head}$ is the head of the clause P_i, $P_{i,body}$ is the body of clause P_i, mgu means the most general unifier and substitutions are represented as $\{t_i/X_i\}$ where each X_i is a variable and t_i is a term.

Let us consider the predicate `sum/2` which computes the sum of the elements of a list and the predicate `count/2` which computes the number of elements of a list. This example corresponds to the (t,t) entry in the table 6.2.

```
P1 : sum([],0).
P2 : sum([H|T],Sum) :- sum(T,Sum1),
                       Sum is Sum1 + H.
Q1 : count([],0).
Q2 : count([H|T],C) :- count(T,C1),
                       C is C1 + 1.
```

The application of join 1-1 on the two procedures `sum/2` and `count/2` with respect to the join specification shown below will be described as follows:

```
sum_count(List,Sum,Count) ⇐ sum(List,Sum), count(List,Count).
```

The process consists of taking clause 1 from program `sum/2` and clause 1 from program `count/2` and to create the template $T_1 :- P, Q$. By unfolding P and Q using P_1 and Q_1, P_1 unifies with P using the substitution θ_P = {[]/List, 0/Sum} and Q_1 unifies with Q with the substitution θ_Q = {[]/List, 0/Count} by replacing $Q_{1,body}$ = `true` and $Q_{1,body}$ = `true` in T_1, we have the following clause:

```
sum_count([],0,0) :- true, true.
```

By making reductions on the previous clause we get the first combined clause which is shown as follows:

```
Combined clause 1:      sum_count([],0,0).
```

Secondly, we take clause 2 from program `sum/2` and clause 2 from program `count/2` and create a template of the join specification i.e. creating $T_2 :- P, Q$. After unfolding P and Q in T_2 with respect to P_2 and Q_2 we get the clause that appears below. P_2 unifies with P with the substitution $\theta_P = \{$`[H|T]/List, Sum/Sum`$\}$ and Q_2 unifies with Q by using the substitution $\theta_Q = \{$`[H|T]/List, Count/Count`$\}$ therefore by replacing

$P_{2,body}$ = `sum(T,Sum1),Sum is Sum1 + H` and

$Q_{2,body}$= `count(T,C1), Count is C1 + 1` in

T_2 we get the following clause:

```
Combined clause 2:

sum_count([H|T],Sum,Count) :-
            sum(T,Sum1),
            Sum is Sum1 + H,
            count(T,C1),
            Count is C1 + 1.
```

By folding combined clause 2 with respect to the join specification `sum_count/3`, then adding the combined clause 1, we obtain the final program shown below. This composed program is much more efficient because it traverses the list only once to compute both results instead of traversing the list twice to get the same results.

```
sum_count([],0,0).
sum_count([H|T],Sum,Count) :-
            sum_count(T,Sum1,C1),
            Sum is Sum1 + H,
            Count is C1 + 1.
```

6.6.3 Restrictions

This method is a limited method for combining programs. It works efficiently for pairs of programs which do not have tests in recursive clauses or when the test is the same in both predicates. If these conditions are not met we can generate an incomplete combined program such as the example shown below, in which we have applied the join 1-1 method in the combination of the programs `sum_odds/2` and `sum_fives/2` defined on page 113. Recall that a method which works for this is the procedural join method, described on page 111). For this particular example the program `sum_odds/2` has a test `odd(N)` and the program `sum_fives/2` has the test `five(N)`. We have two conditions, so we need to consider the set of possible combinations which are shown as follows:

- case 1: `odd(X) and five(X)`,
- case 2: `odd(X) and not(five(X))`,
- case 3: `not(odd(X)) and five(X)`,
- case 4: `not(odd(X)) and not(five(X))`

The incorrect program `sum_of/3` generated using the join 1-1 method is shown below. This incorrect program is obtained because the combination is only performed by taking corresponding clauses and in our example these corresponding clauses test different conditions.

```
sum_of([],0,0).
sum_of([X|R],SO,SF) :-
            odd(X),
            five(X),
            sum_of(R,S1,S2),
            SO is S1 + X,
            SF is S2 + X.
sum_of([X|R],SO,SF) :-
            \+ odd(X),
            \+ five(X),
            sum_of(R,SO,SF).
```

Join 1-1 (M_2)					
P_1/P_2	t	st	s	ctr	meta
t	ϕ_1, α				
st		ϕ_1, α			
s			ϕ_1, α		
ctr				ϕ_1, β	
meta					ϕ_1, β

Table 6.2: Table for the Join 1-1 Method

The problem described above is solved using the procedural join method described in Section 6.7. This method considers all the possible combinations of the clauses (not only corresponding clauses).

The table 6.2 shows the performance of the join 1-1 method (see section 6.3 for an explanation of the terms α and β). Note that this table is smaller than the one for the synchronization method. The reason is we ignore all the mutant entries because this method is not suitable for mutants. The same convention is used in the tables for the procedural join, meta-composition, DS, particular and general method.

The condition ϕ_1 states that both programs belong to type Traverse-restricted. This means that both predicates (taken clause by clause) have zero tests or if both predicates have a test then the test is the same. In this method also we require that the programs have their clauses in the same order. The combined program does not contain redundant clauses.

If we look at the pattern of the entries in the table 6.2 we can see that this method only combines programs which are extensions of the same skeleton. In particular two programs constructed using either the counter or meta-interpreter skeleton are combined correctly but less efficiently than the other pairs which can be combined using this method (see Section 6.8).

6.7 The Procedural Join Method

The *procedural-join* (developed by Lakhotia and Sterling [Lakhotia & Sterling 87]) is a development of the previous method. For completeness, in our composition system, we re-implemented the procedural join method. The *procedural join* method is performed by a sequence of unfolding and folding operations taking into consideration the form of induction parameter for the combination of clauses from each program [Lakhotia & Sterling 87]. This method is used in the combination of programs belonging to type Traverse-general (defined in Chapter 5). The clauses in each program do not need to be in the same order as the order given by the skeleton.

Procedural join composes a new program from two given programs by firstly combining the first clause of program P with each of the clauses in program Q (taken one by one); secondly taking clause two of program P with each of the clauses in program Q; and so on. This method does not enforce the synchronization in the traversing of the data structure for each clause. Therefore all the answers can be obtainable in the combined program.

The notation used in the algorithm for procedural join is as follows: P_i is the i^{th} clause from program P and similarly Q_j is the j^{th} clause of program Q. The algorithm for procedural join is defined as follows:

Procedural Join Algorithm

1. Define the join specification $T \Leftarrow P, Q$.
2. Take a pair of clauses P_i and Q_j.
3. Create a template T_k.

$$T_k :- P, Q$$

4. Unfold P and Q in T_k with respect to P_i and Q_j.
5. Apply the *fold* operation if there exists a pair of subgoals in the body of the clause T_k which can be folded.
6. Repeat this process for all clauses in procedure P and procedure Q.

Procedural join is analogous to the *join operation* in relational algebra where the θ-join of two relations $\mathcal{R}$ and $\mathcal{S}$ are those tuples in the cartesian product of $\mathcal{R}$ and S [Ullman 86].

In some examples after applying procedural join without restriction in the definition of the data structure on which they operate, we get programs with clauses containing subgoals that fail at all times. These clauses do not contribute to the meaning of the generated program and thus are redundant and they need to be eliminated from the program by the user.

6.7.1 Assumptions for the Application of the Algorithm

1. The extensions have a similar structure because they are derived from the same skeleton.
2. The extensions operate on the same type of data structure (eg. lists, trees, etc).
3. The order of the clauses from each program do not need to be same (ie. the corresponding clauses do not be in the same order for applying this method).
4. The program was constructed only using techniques which do not change the flow of control. The kind of techniques used are these which adds computation around the flow of control, provided by the skeleton, preserving the structure of the skeleton.

5. The values that the programs compute do not depend upon each other, they depend only on the input data.

6.7.2 Example

The example program `sum_odds/2` computes the sum of the odd numbers in a list and the program `sum_fives/2` computes the sum of all the number fives that appear in a list. The definitions of these two programs are given below. Note that this example corresponds to the (t,t) entry in the table 6.3.

```
sum_odds([],0).
sum_odds([X|R],S0) :-
            odd(X),
            sum_odds(R,S1),
            S0 is S1 + X.
sum_odds([X|R],S0) :-
            \+ odd(X),
            sum_odds(R,S0).
```

```
sum_fives([],0).
sum_fives([X|R],SF) :-
            five(X),
            sum_fives(R,S2),
            SF is S2 + X.
sum_fives([X|R],SF) :-
            \+ five(X),
            sum_fives(R,SF).
```

The join specification used for combining the two programs `sum_odds/2` and `sum_fives/2` is shown below:

```
sum_of(L,SumOdds,SumFives) ⇐ sum_odds(L,SumOdds), sum_fives(L,SumFives).
```

The algorithm can be described as follows:

Step 1: taking clause 1 from program `sum_odds/2` and clause 1 from program `sum_fives/2` we can create the template $T_1 :- P, Q$. Unfolding P and Q with respect to P_1 and Q_1. P_1 unifies with P by using the substitution
$\theta_P = \{$`[]/L, 0/SumOdds`$\}$ and Q_1 unifies with Q by using the substitution
$\theta_Q = \{$`[]/L, 0/SumFives`$\}$. Replacing $P_{1,body}$ which in this case unifies with `true` and $Q_{1,body}$ which unifies with `true` in T_1, we obtain the following clause:

```
sum_of([],0,0) :- true, true.
```

which is equivalent to the following clause:

```
sum_of([],0,0).
```

Step 2: taking clause 1 in program `sum_odds/2` and clause 2 in program `sum_fives/2` we find that `L` unifies with [] for clause 1 and `L` unifies with `[X|R]` for clause 2 so it is not possible to create a template of the join specification and the process continues taking another pair of clauses, this time clause 1 from program `sum_odds/2` and clause 3 from program `sum_fives/2` in a similar way to that described for clause $P1$ and $Q2$.

Step 3: taking clause 2 from program `sum_odds/2` and clause 1 from program `sum_fives/2` we find that these clauses have a different form of structural induction parameter. Therefore we have the same case as in step 2.

Step 4: taking clause 2 from program `sum_odds/2` and clause 2 from program `sum_fives/2` we can create an instance of the join specification (i.e. $T_2 \ :- \ P, Q$). Unfolding P and Q in T_2 with respect to P_2 and Q_2, P_2 unifies with P by using the substitution $\theta_P = \{$`[X|R]/L, SO/SumOdds`$\}$ and Q_2 unifies with Q by using the substitution $\theta_Q = \{$`[X|R]/L, SF/SumFives`$\}$.

Therefore by replacing:

$P_{2,body}$= `odd(X), sum_odds(R,S1), SO is S1 + X` and

$Q_{2,body}$= `five(X), sum_fives(R,S2), SF is S2+X` in

T_2= `sum_of([X|R],Sum,Prod)` we have the following clause:

```
sum_of([X|R],SO,SF) :-
                odd(X),
                sum_odds(R,S1),
                SO is S1 + X,
                five(X),
                sum_fives(R,S2),
                SF is S2 + X.
```

The body of the clause defined above (after applying the unfolding operation) can be folded to obtain the following clause:

```
sum_of([X|R],SO,SF) :-
        odd(X),
        five(X),
        sum_of(R,S1,S2),
        SO is S1 + X,
        SF is S2 + X.
```

Step 5: taking clause 2 from program `sum_odds/2` and clause 3 from program `sum_fives/2` we can create an instance of the join specification $T_3 \ :- \ P, Q$. Unfolding P and Q in T_3 with respect to P_2 and Q_3, give us the case that P_2 unifies with P under the substitution $\theta_P = \{$`[X|R]/L, SO/SumOdds`$\}$ and Q_3 unifies with Q under the substitution $\theta_Q = \{$[X|R]/L, SF/SumFives$\}$.

Therefore by replacing:

$P_{2,body}$= `odd(X), sum_odds(R,S1), SO is S1 + X` and

$Q_{3,body} =$ `\+ five(X), sum_fives(R,SF)` in T_2 we obtain the following clause:

```
sum_of([X|R],SO,SF) :-
        odd(X),
        sum_odds(R,S1),
        SO is S1 + X,
        \+ five(X),
        sum_fives(R,SF).
```

The body of the clause defined above (after applying the unfolding operation) can be folded to give two instances: `sum_odds(R,S1)` with `sum_fives(R,SF)` obtaining `sum_of(R,S1,SF)`. The combined clause is shown below.

```
sum_of([X|R],SO,SF) :-
        odd(X),
        \+ five(X),
        sum_of(R,S1,SF),
        SO is S1 + X.
```

Step 6: taking the case of clause 3 from program `sum_odds/2` and clause 1 from program `sum_fives/2` we find that these clauses have a different form of structural induction parameter and therefore cannot be combined.

Step 7: taking clause 3 from program `sum_odds/2` and clause 2 from program `sum_fives/2` we can create an instance of the join specification $T_4 \ :- \ P, Q$. Unfolding P and Q in T_4 with

respect to P_3 and Q_2, P_3 unifies with P with the substitution $\theta_P = \{$`[X|R]/L, SO/SumOdds`$\}$ and Q_2 unifies with `Q` by using the substitution $\theta_Q = \{$`[X|R]/L,SF/SumFives`$\}$.

Therefore by replacing:

$P_{3,body}$ = `\+ odd(X), sum_odds(R,SO)` and

$Q_{2,body}$= `five(X), sum_fives(R,S2), SF is S2 + X` in T_4 we get the following clause:

```
sum_of([X|R],SO,SF) :-
                \+ odd(X),
                sum_odds(R,SO),
                five(X),
                sum_fives(R,S2),
                SF is S2 + X.
```

The body of the clause defined above can be folded and the combined clause is shown below.

```
sum_of([X|R],SO,SF) :-
                \+ odd(X),
                five(X),
                sum_of(R,SO,S2),
                SF is S2 + X.
```

Step 8: taking clause 3 from program `sum_odds/2` and clause 3 from program `sum_fives/2` we can create an instance of the join specification T_5 :- P, Q. Unfolding P and Q in T_5 with respect to P_3 and Q_3, P_3 unifies with P with the substitution $\theta_P = \{$`[X|R]/L, SO/SumOdds`$\}$ and Q_3 unifies with Q by using substitution $\theta_Q = \{$`[X|R]/L,SF/SumFives`$\}$.

Therefore by replacing:

$P_{3,body}$ = `\+ odd(X), sum_odds(R,SO)` and

$Q_{3,body}$ = `\+ five(X), sum_fives(R,SF)` in

T_5 we have the following clause:

```
sum_of([X|R],SO,SF) :-
                \+ odd(X),
                sum_odds(R,SO),
                \+ five(X),
                sum_fives(R,SF).
```

The body of the clause defined above can be folded and the combined clause is shown below.

```
sum_of([X|R],SO,SF) :-
            \+ odd(X),
            \+ five(X),
            sum_of(R,SO,SF).
```

The resulting combined program is as follows:

```
T1 : sum_of([],0,0).
T2 : sum_of([X|R],SO,SF) :-
            odd(X),
            five(X),
            sum_of(R,S1,S2),
            SO is S1 + X,
            SF is S2 + X.
T3 : sum_of([X|R],SO,SF) :-
            odd(X),
            \+ five(X),
            sum_of(R,S1,SF),
            SO is S1 + X.


T4 : sum_of([X|R],SO,SF) :-
            \+ odd(X),
            five(X),
            sum_of(R,SO,S2),
            SF is S2 + X.
T5 : sum_of([X|R],SO,SF) :-
            \+ odd(X),
            \+ five(X),
            sum_of(R,SO,SF).
```

This method as we show on page 104 might generate unwanted solutions. Also the procedural join method does not handle the problem of when calls to be fold do not have syntactically identical input variables as the definition of the join specification. So, we implement the meta-composition method which is described in next section. This method handles the problem of local variables, it does not produce redundant clauses. The disadvantage is that it can only be used with programs which are enhancements of the same flow of control. However, this meta-composition method can be used for combining efficiently programs belonging to the meta-interpreters class or programs belonging to the *counter*

Procedural join (M_3)					
P_1/P_2	t	st	s	ctr	meta
t	ϕ_2, α	ϕ_2, β	ϕ_2, β		
st	ϕ_2, β	ϕ_2, α	ϕ_2, β		
s	ϕ_2, β	ϕ_2, β	ϕ_2, α		
ctr				ϕ_2, β	
meta					ϕ_2, β

Table 6.3: Table for the Procedural Join Method

class. It can also be used for programs which are constructed using the data abstraction technique.

The table 6.3 summarises the performance of the procedural join method (see section 6.3 for an explanation of the terms α and β).

The condition ϕ_2 means that both programs (taken clause by clause) have one or more different tests, but no order of the clauses in both programs is required.

The combined program could have unwanted solutions (see example in section 6.6.1). This method covers six more entries in the table 6.3: the pairs (st,t), (t,st), (s,t), (t,s), (st,s) and (s,st) which were not covered by the previous method (join 1-1). However, we still get a inefficient combined program for the pairs (ctr,ctr) and (meta,meta). This problem inspires the meta-composition method which deals with this special class of programs.

The off-diagonal entries have different flow of control and hence are studied in more detail in Chapter 7, e.g. for an example of the (s,t) entry see page 178.

6.8 The Meta-Composition Method

This method generalises Lakhotia and Sterling's algorithm named procedural join [Lakhotia & Sterling 87] by relaxing the constraint that the two recursive calls to be folded must have syntactically identical input variables. This is necessary, for example, when the value being recursed is computed by a call to a user-defined predicate (see the first and

second example in this section). This method can be used for combining pairs of different classes of programs such as meta-interpreters, programs constructed using the *counter* skeleton or all the classes already covered by the join 1-1. Also this method can be used for any class of program cited above which is constructed by using data abstraction. Data abstraction generalises the specific data structures or built-in predicated replacing them with more abstract predicates (see the third example in this section).

Note that this method will enforce the same traversal of the data structure for each clause as join 1-1, and so will give a subset of answers as described in Section 6.6. Commonly this enforcing of synchronised traversal of a shared data structure is what is desired and the naive (fold-unfold) join is not really what the programmer wants. This is a way to kill spurious solutions due to unwanted backtracking (without having to resort to cuts).

The main characteristic of programs which belong to the meta-interpreters class and programs constructed by using the *counter* skeleton is that they use local variables to recurse over the data structure. Local variables are variables which occur in the body of a clause but not in its head. These two classes cannot be combined efficiently by using join 1-1 even when both programs are extensions of the same skeleton. The reason for this is that the join 1-1 method cannot deal with local variables which are not controlled in the join specification. So even when candidate subgoals to be folded appear, they might not be folded because they do not match to an instance of the join specification.

The meta-composition method allows optimisation over local variables with certain restrictions. In general, it is not safe to make the assumption that new local variables introduced in the body of a clause can always be unified. However there are circumstances where we can unify new local variables confidently. For example, in the following piece of code the local variables `X` and `Y` are introduced in the same environment (same clause) hence they can be unified.

```
predicate(A,X)
        :
predicate(A,Y).
```

Also, it is safe to unify variables which are used for traversing the same data structure. In this thesis, variables which are used for traversing the same data structure are said to be variables having the same functionality.

The main assumption for the meta-composition method is that it is restricted to combining corresponding clauses from each program. Hence, we can be sure that the candidate subgoals to be folded are used for traversing the same instance of the data structure, and so the variables can be safely unified.

The algorithm for meta-composition based on key points is shown below. In this algorithm we use the concept of key points defined in Chapter 5. A key point gives the structural differences between a program and the skeleton from which is derived the program.

Meta-composition Algorithm

1. Define the join specification $T \Leftarrow P, Q$.
2. Take a pair of clauses P_i and Q_i.
3. Create a template T_i.

$$T_i \ :- \ P, Q$$

4. Unfold P and Q in T_i with respect to P_i and Q_i.
5. Apply the *meta-fold* operation if there exists a pair of subgoals in the body of the clause T_i which can be folded. This is performed by using key points which are defined in the program history.
6. Repeat this process for each pair of clauses from program P and program Q.

In the *meta-composition* method we are only using a section of the *program history* information. The valuable information required from the *program history* is how the subgoals that appear in the skeleton were transformed into the program (i. e. key points defined in Chapter 5). Also by using the program history, it can be determined if a second join specification can be used for further optimisation. This fact is deduced from the program history. For example, if we have recorded that the program is created using the meta-

interpreter called modulant skeleton of meta-interpreter (defined in Appendix C) then the system infers that the clause number 2 in the program calls in its body an auxiliary predicate which can be used for further optimisation using a second join specification.

Our implementation of the meta-composition algorithm is based on unfolding and meta-folding transformations. This meta-folding operation is different to the folding operation used in procedural join and join 1-1. The *meta-folding* operation verifies, before folding, whether the subgoals (from program P and from program Q) that are candidates for joining are enhancements of the same subgoal in the skeleton. These candidate subgoals are determined by the use of *key points* at each stage of the composition process.

The meta-composition method is a suitable method for combining meta-interpreters but it is not restricted to that class of programs. This method can be used for combining programs which are extensions of the same skeleton with the same number of clauses. Therefore we can use the meta-composition method for combining all programs which can be combined using join 1-1. The converse is not true, programs that belong to meta-interpreters or the counter class cannot be efficiently combined using join 1-1. This is because the join 1-1 method performs a sequence of transformations consisting of unfolding, folding and then the merge operation, whilst the meta-composition method performs unfolding, meta-folding and the merge operation.

In short, the meta-composition method uses knowledge about the development of the program, providing information about the subgoals that appear in the skeleton and how they were transformed in each program. This information controls the folding operation and, by using this information, reduction in the numbers of local variables can be performed as shown in the following example.

6.8.1 Example 1

In the following example we combine the meta-interpreter `new_fuzzy/3` which returns the associated certainty factor and explanation for a query and the meta-interpreter

fuzzydepth/3 which attempts to compute the certainty factor for a query but is limited by the depth to which it can recurse. These two meta-interpreters are shown below. Note that this example corresponds to the (meta,meta) entry in the table 6.4.

```
P1: new_fuzzy(A,B,fact(A,B)) :-
      fact(A,B).

P2: new_fuzzy(not(D),B,not(D,B,F)) :-
      new_fuzzy(D,E,F),
      B is 1 - E.

P3: new_fuzzy(A,B,rule(A,B,G)):-
      rule(A,D,E),
      new_fuzzy(D,F,G),
      B is E * F.
```

```
Q1: fuzzydepth(A,B,C) :-
      fact(A,C).

Q2: fuzzydepth(not(E),B,C):-
      fuzzydepth(E,B,F),
      C is 1 - F.

Q3: fuzzydepth(A,B,C) :-
      B > 0,
      D is B - 1,
      rule(A,E,F),
      fuzzydepth(E,D,G),
      C is F * G.
```

```
P4: new_fuzzy((D & E),B,conj(G,I,B)):-
      new_fuzzy(D,F,G),
      new_fuzzy(E,H,I),
      min(F,H,B).

P5: new_fuzzy((D or E),B,disj(G,I,B)):-
      new_fuzzy(D,F,G),
      new_fuzzy(E,H,I),
      max(F,H,B).
```

```
Q4: fuzzydepth((D & E),B,C) :-
      fuzzydepth(D,B,F),
      fuzzydepth(E,B,G),
      min(F,G,C).

Q5: fuzzydepth((D or E),B,C) :-
      fuzzydepth(D,B,F),
      fuzzydepth(E,B,G),
      max(F,G,C).
```

Additionally, we need the following definitions for min/3 and max/3 defined below.

```
max(A,B,A) :- A > B.
max(A,B,B) :- A =< B.

min(A,B,A) :- A < B.
min(A,B,B) :- B =< A.
```

The two meta-interpreters new_fuzzy/3 and fuzzydepth/3 defined above will be combined using the join specification shown below.

```
new_fuzzydepth(Goal,Cf,Exp,Depth) ⇐
        new_fuzzy(Goal,Cf,Exp),
        fuzzydepth(Goal,Depth,Cf).
```

where the underlined arguments provide flows of control which we want to synchronise. Since we assume knowledge of the history of development of the program we can check which arguments were intended to provide the flow of control and assess whether they will be compatible in combination.

The corresponding simplified versions of the program histories for the programs `new_fuzzy/3` and `fuzzydepth/3` are

```
his_prog(new_fuzzy,3,meta-interpreter,[[1,(true,fact(A,B)),no_test],
      [2,(clause(A,B),true)),(solve(B),new_fuzzy(D,E,F)),no_test],
      [3,(clause(A,B),rule(A,D,E)),(solve(B),new_fuzzy(D,F,G)),no_test],
      [4,(solve(A),new_fuzzy(D,F,G)),(solve(B),new_fuzzy(E,H,I)),no_test],
      [5,(solve(A),new_fuzzy(D,F,G)),(solve(B),new_fuzzy(E,H,I)),no_test]],compute_cf).

his_prog(fuzzydepth,3,meta-interpreter,[[1,(true,fact(A,C)),no_test],
      [2,(clause(A,B),true),(solve(B),fuzzydepth(E,B,F)),no_test],
      [3,(clause(A,B),rule(A,E,F)),(solve(B),fuzzydepth(E,D,G)),B >0]
      [4,(solve(A),fuzzydepth(D,B,F)),(solve(B),fuzzydepth(E,B,G)),no_test],
      [5,(solve(A), fuzzydepth(D,B,F)),(solve(B),fuzzydepth(E,B,G)),no_test]],
      compute_cf).
```

Note that for reasons of clarity we have deleted 4 arguments in the previous simplied versions of the program histories. For each history, the first argument is the name of the program; the second is the arity; the third is the name of the initial control flow used in the construction of the program; the fourth argument is a list recording, for each clause, how the subgoals in the initial control flow were transformed; and the fifth is the technique used to extend the initial flow of control.

The program histories thus say that both programs were built using the "meta-interpreter" skeleton, that both programs have the same number of clauses and both traverse the same data structure. By applying the meta-composition method that makes the combination by combining only corresponding clauses, rather than combining all pairs of clauses, we achieve the desired synchronisation in the unpacking of the data structure.

In this case it will observe (from the program history) that both programs have the same flow of control, called "meta-interpreter"[2], and they were constructed using the same techniques. We assume that the user would also like the flow of control of the output to be "meta-interpreter" and would simply like to have the compute_cf technique (which computes certainty factor) added. The definition of this technique is in Appendix D.

Let us look at this in a bit more detail. The flow of control for both **new_fuzzy/3** and **fuzzydepth/3** is a variation of meta-interpreter skeleton, which can be described as:

```
solve(true).
solve(A) :-
        clause(A,B),
        solve(B).
solve((A,B)) :-
        solve(A),
        solve(B).
```

The stages of the composition process can be described as follows:

Step 1: taking the first clause of program new_fuzzy/3 and the first clause of program **fuzzydepth/3** we obtain the clause shown below.

```
new_fuzzydepth(A,B,fact(A,B),_Depth) :-
        new_fuzzy(A,B,fact(A,B)),
        fuzzydepth(A,Depth,B).
```

Then by unfolding the call new_fuzzy/3 and fuzzydepth/3 in the previous clause we get the following clause:

T_1 :
```
new_fuzzydepth(A,B,fact(A,B),_Depth) :-
        fact(A,B),
        fact(A,B).
```

We obtain the combined clause T_1 by applying the merge operation which removes the syntactically identical subgoals.

T_1 :
```
new_fuzzydepth(A,B,fact(A,B),_Depth) :-
        fact(A,B).
```

[2]So-named because its function is to allow us to create meta-interpreters

Step 2: using the second clause of program `new_fuzzy/3` and the second clause of program `fuzzydepth/3` and by unfolding their definition in T_2 we thus get the clause shown below.

```
T2 : new_fuzzydepth(not(E),B,not(E,B,G),Depth) :-
        new_fuzzy(E,F,G),
        B is 1 - F,
        fuzzydepth(E,Depth,F1),
        B is 1 - F1.
```

In this clause the subgoals `new_fuzzy(E,F,G)` and `fuzzydepth(E,Depth,F1)` cannot be replaced by a single subgoal `new_fuzzydepth(E,F,G,Depth)` in the normal way by using the join specification. The variable `F` needs to be unified to `F1`. This can be done in two ways: one is by program analysis, the other is by using the histories of the programs provided by the techniques editor. We took the second approach because we already have the program history. This approach involves the use of extra information which is kept in the history of the program development. This history of the program provides knowledge about how the initial skeleton is transformed in the development of the program. By using this information it is possible to fold the subgoals which are derived from the appropriate subgoal in the initial skeleton.

The information contained in the eighth argument (fourth here) of the program history informs us that the subgoals `new_fuzzy(E,F,G)` and `fuzzydepth(E,Depth,F1)` are enhancements of the same subgoal in the skeleton `solve(B)`. This argument is a list recording, for each clause, how the subgoals in the initial control flow were transformed. This information enables us to infer, without having to perform any program analysis, which variables are used for computing exactly the same value (ie. they have the same functionality, for instance to decompose the data structure) and hence can be bound together. The binding of these local variables allows the folding of the subgoals `new_fuzzy(E,F,G)` and `fuzzydepth(E,Depth,F1)`. The optimised combined clause shown as follows:

```
T2 : new_fuzzydepth(not(E),B,not(E,B,G),Depth) :-
        new_fuzzydepth(E,F,G,Depth),
        B is 1 - F.
```

Proietti and Pettorossi proposed an algorithm based on unfolding, folding and addition of new join specifications [Proietti & Pettorossi 92]. The main characteristic of the algorithm

is that it eliminates unnecessary variables which occur in the body of the clause and they do not occur in the head of the clause (ie. they are local variables). This algorithm requires three actions which need to be defined for the user: the introduction of a set of join specifications for the folding step, selection of the calls in the body of the clauses for unfolding stages and choice of arithmetic laws to be applied.

However Proietti and Pettorossi's algorithm cannot be applied to the meta-interpreter class. The reason is that no further subgoal can be unfolded (i.e. no more redundant variables can be eliminated). So no more optimisation can be performed using this approach based on the standard unfolding and folding operations. However our solution based in *program history* handles the problem of removing unnecessary duplicated variables by detecting whether they are used for deconstructing or building the data structure (by means of the program history) and also reduces user interaction.

Step 3: we take the third clause of each program and by unfolding their definition in T_3 we get the clause shown below.

```
T3 : new_fuzzydepth(A,B,rule(A,B,H),Depth) :-
                rule(A,E,F),
                new_fuzzy(E,G,H),
                B is F * G,
                Depth > 0,
                D1 is Depth - 1,
                rule(A,E1,F1),
                fuzzydepth(E1,D1,G1),
                B is F1 * G1.
```

By performing the binding of local variables E1 to E and G1 to G we obtain the following clause:

```
T3 : new_fuzzydepth(A,B,rule(A,B,H),Depth) :-
                rule(A,E,F),
                Depth > 0,
                D1 is Depth - 1,
                new_fuzzydepth(E,G,H,D1),
                B is F * G.
```

Step 4: we take the fourth clause from each program and by unfolding the following clause is obtained.

```
T4 : new_fuzzydepth((D & E),B,conj(H,J,B),Depth) :-
        new_fuzzy(D,G,H),
        new_fuzzy(E,I,J),
        min(G,I,B),
        fuzzydepth(D,Depth,F1),
        fuzzydepth(E,Depth,G1),
        min(F1,G1,B).
```

In this clause we need to infer that the variable G can be unified to F1 and that the variable I can be unified to G1. This cannot easily be obtained by performing program analysis of the subgoals which involve the variables G,F1,I and G1. For example we can take the subgoals min(G,I,B) and min(F1,G1,B) and hypothetical substitutions such as G=.2, I=.3 and F1=.2 and G1=.5. The variable I cannot be unified with G1 even when the variable G can be unified with F1. However the program history informs us that the subgoals new_fuzzy(D,G,H) and fuzzydeph(D,Depth,F1) are enhancements of the same subgoal in the skeleton and also the variable G and F1 have the same functionality as is defined in the join specification (to compute the certainty factor) and can be unified.

In a similar fashion new_fuzzy(E,I,J) and fuzzydepth(E,Depth,G1) are enhancements of the same subgoal in the skeleton solve. This information and the information about the use of each variable (functionality) allows us to determine that the variables G and F1 can be unified and that the variable I and G1 can be unified. Therefore the previous clause can be optimised by means of the meta-folding operation which determines which variables can be safely unified. By applying the meta-folding operation we obtain the combined clause T_4.

```
T4 : new_fuzzydepth((D & E),B,conj(H,J,B),Depth) :-
        new_fuzzydepth(D,G,H,Depth),
        new_fuzzydepth(E,I,J,Depth),
        min(G,I,B).
```

Similarly the clause T_5 can be obtained. This clause is shown as follows:

```
T5 : new_fuzzydepth((D or E),B,disj(H,J,B),Depth) :-
        new_fuzzydepth(D,G,H,Depth),
        new_fuzzydepth(E,I,J,Depth),
        max(G,I,B).
```

The complete program new_fuzzydepth/4 is shown as follows:

```
T1 : new_fuzzydepth(A,B,fact(A,B),_Depth) :-
         fact(A,B).

T2 : new_fuzzydepth(not(E),B,not(E,B,G),Depth) :-
         new_fuzzydepth(E,F,G,Depth),
         B is 1 - F.

T3 : new_fuzzydepth(A,B,rule(A,B,H),Depth) :-
         rule(A,E,F),
         Depth > 0,
         D1 is Depth - 1,
         new_fuzzydepth(E,G,H,D1),
         B is F * G.

T4 : new_fuzzydepth((D & E),B,conj(H,J,B),Depth) :-
         new_fuzzydepth(D,G,H,Depth),
         new_fuzzydepth(E,I,J,Depth),
         min(G,I,B).

T5 : new_fuzzydepth((D or E),B,disj(H,J,B),Depth) :-
         new_fuzzydepth(D,G,H,Depth),
         new_fuzzydepth(E,I,J,Depth),
         max(G,I,B).
```

In general, the combined program will return just a subset of the values that would be obtained from the join specification if we were to regard it simply as a Prolog program. However, in our opinion, the combined program is almost certainly what the programmer actually desired. Hence we prefer to regard the join specification as a high-level description that implicitly contains more constraints than the naive Prolog equivalent.

For example, Appendix F contains the behaviour of the "naively combined meta-interpreters" obtained by regarding the specification as just Prolog. In this case we need to take the full cartesian product. Not only is the result different from what we believe the programmer would have intended, but it is also very inefficient because of the many extra clauses.

Thus, after comparing our combined program generated using our meta-composition method with a program generated by transformations of the specification considered purely as a Prolog program, we find that our combined program is both closer to the user's presumed intentions and also more efficient.

In fact, using only the original two programs, it would be difficult (if not impossible) to write a join specification just in Prolog that would exclude the unwanted extra solution obtained from the naive join specification.

Potentially this gives a big advantage over the standard program transformation methods which can only be given the input Prolog specification, and so would have difficulty recovering the suggestion (available from the skeletons) that quite possibly only the subset of solutions are needed in which both programs traverse the Goal in the same fashion.

The naive Prolog specification failed because it did not enforce the same traversal in both clauses, and hence the depth limit `Depth` was not forced to correspond to the explanation `Exp` provided.

In the following example we show the case where the value being recursed over is itself computed by a call to a user-defined predicate. This is another case in which the meta-composition method can safely unify local variables.

6.8.2 Example 2

In this example our initial programs are the meta-interpreters `interpret/1` (taken from [Sterling & Shapiro 86]) and `count/2`, which are constructed by using the skeleton `solve/1` (depicted in Appendix C). The program `interpret/1` which explains a proof tree created for a goal (query) and `count/2` counts the number of rules needed for proving this goal. Note this example also corresponds to the (meta,meta) entry in the table 6.4.

```
interpret((Proof1,Proof2)):-
        interpret(Proof1),
        interpret(Proof2).
interpret(Proof):-
        fact(Proof,Fact),
        write([Fact,' is a fact in the database']).
```

```
interpret(Proof):-
      rule(Proof,Head,Body,Proof1),
      write([Head,' is proved using rule']),
      display_rule(rule(Head,Body)),
      interpret(Proof1).

count((A,B),C):-
      count(A,CA),
      count(B,CB)
      C is CA + CB.
count(A,1):-
      fact(A,_R).
count(A,C):-
     rule(A,_Head,_Body,B),
     count(B,CB),
     C is CB + 1.
```

Additionally, we need the following definitions for `fact/2` and `rule/4`:

```
fact((Fact :- true),Fact).

rule((Goal :- Proof),Goal,Body,Proof) :-
     Proof \== true, extract_body(Proof,Body).
```

We consider the meta-interpreter `interpret/1` as being comprised of clauses P_1, P_2 and P_3 and `count/2` by clauses Q_1, Q_2 and Q_3. In both programs the clauses are in the same order.

A new program `int_count/2` (comprised for clauses T_1, T_2 and T_3), which explains a proof tree and simultaneously counts the number of rules in the proof, is generated by using the following join specification:

```
int_count(Proof,C) ⇐
interpret(Proof), count(Proof,C).
```

The process consists of taking clause 1 from program `interpret/1` and clause 1 from program `count/2`, and creating an instance of the join specification (*i.e.*, $T_1 \ :- \ P, Q$). After unfolding `interpret(Proof)` and `count(Proof,C)` with respect to P_1 and Q_1, we obtain the following clause:

```
T1 :  int_count((Proof1,Proof2),C) :-
          interpret(Proof1), interpret(Proof2),
          count(Proof1,CP1), count(Proof2,CP2),
          C is CP1+CP2.
```

Applying the folding operation we get the following clause:

```
T1 : int_count((Proof1,Proof2),C) :-
        int_count(Proof1,CP1), int_count(Proof2,CP2),
        C is CP1+CP2.
```

Taking clause 2 from each program and applying the unfolding operation we obtain the clause shown below:

```
T2 : int_count(Proof,1):-
        fact(Proof,Fact1),
        nl, write([Fact1,' is a fact in the data base']),
        fact(Proof,Fact2).
```

The histories of the programs provided by the techniques editor inform us that subgoals `fact(Proof,Fact1)` and `fact(Proof,Fact2)` are enhancements of the same subgoal in the skeleton `solve` which is a variant of the skeleton `solve` defined in Appendix C. This information enables us to infer, without having to perform any program analysis, which variables have the same functionality and hence can be bound together. In this case, we are able to determine that variable `Fact2` can be unified to `Fact1` and we get a more optimised clause T_2:

```
T2 : int_count(Proof,1):-
        fact(Proof,Fact1),
        nl, write([Fact1,' is a fact in the data base']).
```

Finally, taking clause 3 from each program and applying the unfolding operation, we get to clause T_3:

```
T3 : int_count(Proof,C) :-
        rule(Proof,Head1,Body1,Proof1),
        nl, write([Head1,' is proved using the rule']),
        display_rule(rule(Head1,Body1)),
        interpret(Proof1),
        rule(Proof,Head2,Body2,Proof2),
        count(Proof2,CB),
        C is CB+1.
```

This clause can be optimised by means of the meta-folding operation. This operation determines that the variables `Proof1` and `Proof2` can be unified by knowing that these variables

have the same functionality, i.e., they are used recursively to deconstruct the data structure required by each program. The binding {Proof1/Proof2} of local variables allows the folding of the subgoals rule(Proof,Head1,Body1,Proof1) and count(Proof2,CB) and the removal of the second rule/4 subgoal. The optimised combined clause T_3 is as follows:

T_3 :
```
int_count(Proof,C):-
        rule(Proof,Head1,Body1,Proof1),
        nl, write([Head1,' is proved using the rule']),
        display_rule(rule(Head1,Body1)),
        int_count(Proof1,CB),
        C is CB+1.
```

It could be argued that another way to find out that Head1, Body1 and Proof1 can be unified to Head2, Body2 and Proof2 respectively, is by doing an analysis of each program to be combined and performing mode and type analysis in rule/4 and display_rule/1. In more sophisticated meta-interpreters this analysis can be complex and computationally expensive, whereas in our approach, all the required information is already available.

The resulting combined program using the extra knowledge is shown below. This program has better computational behaviour (traverses the proof a single time) than the combined program generated using standard unfold/fold transformation operations:

```
int_count((Proof1,Proof2),C) :-
        int_count(Proof1,CA), int_count(Proof2,CB),
        C is CA+CB.
int_count(Proof,1):-
        fact(Proof,Fact1),
        nl, write([Fact1,' is a fact in the data base']).
int_count(Proof,C):-
        rule(Proof,Head1,Body1,Proof1),
        nl, write([Head1,' is proved using the rule']),
        display_rule(rule(Head1,Body1)),
        int_count(Proof1,CB),
        C is CB+1.
```

Let consider us the query:

solve(place_in_oven(dish1,middle),P), int_count(P,C) using the following rules.

```
place_in_oven(Dish,top) :-
            pastry(Dish), size(Dish,small).
place_in_oven(Dish,middle) :-
            pastry(Dish), size(Dish,big).
place_in_oven(Dish,moddle) :-
            main_maeal(Dish).
place_in_oven(Dish,low) :-
            slow_cooker(Dish).

pastry(Dish) :- type(Dish,cake).
pastry(Dish) :- type(Dish,bread).

main_meal(Dish) :- type(Dish,meat).

show_cooker(Dish) :- type(Dish,milk_pudding).

type(dish1,bread).
size(dish1,small).
size(dish1,big).
pastry(dish1).
```

The answer to the query `solve(place_in_oven(dish1,middle),P)` gives:

```
P=place_in_oven(dish1,middle):-(pastry(dish1):-(type(dish1,bread):-true)) ,
        (size(dish1,big):- true)
```

The answer to the query `int_count(P,C)` with `P` instantiated to the proof obtained by program `solve/2` is as follows:

```
[place_in_oven(dish1,middle), is proved using the rule]
   IF pastry(dish1),size(dish1,big)
   [THEN , place_in_oven(dish1,middle)]
[pastry(dish1), is proved using the rule]
   IF type(dish1,bread)
    [THEN , pastry(dish1)]

[type(dish1,bread), is a fact in the data base]
[size(dish1,big), is a fact in the data base]
```

Also the variable `C` is unified to 4.

6.8.3 Example 3

The following example uses the technique of *data abstraction.* This example shows how the meta-composition method can be used when all the variables in the head of the clause are free variables.

Consider the program collect_per/2 which builds a list of people whose salary is greater than 10000. In this program, we have used data abstraction to remove references to specific data structures or built-in predicates and have replaced these with more abstract predicates. Note that this example also corresponds to the (meta,meta) entry in the table 6.4.

```
collect_per(L_per,Data_St):-
            base_case(L_per),
            assign_base_case(Data_St).
collect_per(L_per,Data_St) :-
            head_tail(L_per,Person,RestGroup),
            test_1(Person,Salary),
            test_2(Salary,10000),
            build(Person,CollectRest,Data_St),
            collect_per(RestGroup,CollectRest).
collect_per(L_per,Collect) :-
            head_tail(L_per,Person,RestGroup),
            test_1(Person,Salary),
            test_3(Salary,10000),
            collect_per(RestGroup,Collect).
collect_per(L_per,Collect) :-
            head_tail(L_per,Group,RestGroups),
            collect_per(Group,Collect1),
            collect_per(RestGroups,Collect2),
            append(Collect1,Collect2,Collect).
```

The program count/2 counts how many employees earn a salary greater than 10000.

```
count([],0).
count(LP,C) :-
            head_tail(LP,Per,Rest_Group),
            test_1(Per,S),
            test_2(S,10000),
            count(Rest_Group,C),
            C is C1 + 1.
count(LP,C) :-
            head_tail(LP,Per,Rest_Group),
            test_1(Per,S),
            test_3(S,10000),
            count(Rest_Group,C).
count(LP,C) :-
            head_tail(LP,Group,Rest_Groups),
            count(Group,C1),
            count(Rest_Groups,C2),
            C is C1 + C2.
```

Additionally, we need the following definitions for test_1/2, test_2, test_3, head_tail/3, assign_base_case/1 and build/3. This is the place in which we make

use of the data abstraction technique with the aim of allowing future revisions to the data structure can be performed easily.

```
test_1(person(_,_,S),S).
test_2(S,N) :- S>N.
test_3(S,N) :- S<N.

head_tail([H|T],H,T).

base_case([]).

assign_base_case([]).

build(Elem,Tail,Data_Str) :-
            Data_Str=[Elem|Tail].
```

The corresponding simplified versions of the program histories for the programs `collect_per/3` and `count/2` are

```
his_prog(collect_per,2,meta-interpreter,
  [[1,(true,[base_case(L_per),assign_base_case(Data_St)]),no_test],
  [2,(clause(A,B),head_tail(L_per,Person,RestGroup)),
  (solve(B),collect_per(RestGroup,CollectRest)), [test_1(Person,Salary),
  test_2(Salary,10000)]],
  [3,(clause(A,B),head_tail(L_per,Person,RestGroup)),
  (solve(B),collect_per(RestGroup,Collect)), [test_1(Person,Salary),
  test_3(Salary 10000)]],
  [4,(solve(A),collect_per(Group,Collect1)),
  (solve(B),collect_per(RestGroups,Collect2))]], build_list_technique).

his_prog(count,2,meta-interpreter, [[1,(true,true),no_test],
  [2,(clause(A,B),head_tail(LP,Per,Rest_Group)),
  (solve(B),count(Rest_Group,C)), [test_1(Per,S), test_2(S,10000)]],
  [3,(clause(A,B),head_tail(LP,Group,Rest_Group)),
  (solve(B),count(Rest_Group,C)), [test_1(Person,S), test_3(S,10000)]],
  [4,(solve(A),count(Group,C1)),(solve(B),count(Rest_Groups,C2))]],count_technique).
```

Note that the same convention for the program histories, used in the previous examples is used here as well (4 arguments have been deleted). For each history, the first argument is the name of the program; the second is the arity; the third is the name of the initial control flow used in the construction of the program; the fourth argument is a list recording, for each clause, how the subgoals in the initial control flow were transformed; and the fifth is the technique used to extend the initial flow of control.

The program histories thus say that both programs were built using the "meta-interpreter" skeleton, that both programs have the same number of clauses and both traverse the same data structure.

The join specification used for producing a new program, `collect_count/3`, is shown as follows:

```
collect_count(List1,List2,Count) ⇐
            collect_per(List1,List2),
            count(List1,Count)
```

Taking clause 1 from each program, creating an instance of the join specification and applying the unfolding operation we obtain the clause shown below:

T_1 : `collect_count([],[],0).`

Taking clause 2 from each program, creating an instance of the join specification and applying the unfolding operation we obtain:

T_2 :
```
collect_count(L_per,Data_St,C) :-
        head_tail(L_per,Person,RestGroup),
        test_1(Person,Salary),
        test_2(Salary,10000),
        build(Person,CollectRest,Data_St),
        collect_per(RestGroup,CollectRest),
        head_tail(L_per,Per,Rest_group),
        test_1(Per,S),
        test_2(S,10000),
        count(Rest_group,C1),
        C is C1 + 1.
```

After applying the meta-folding operation it is possible to unify the variables `RestGroup` to `Rest_group` obtaining the folded clause shown below (see fourth argument clause number 2 in the program history for each of the programs). The program history informs us that the variables `RestGroup` and `Rest_group` (local variables) are used for deconstructing the same instance of the data structure of corresponding clauses. So they can be safely unified. This also can be performed by inspecting the definition of `head_tail/3` which is used for the same input `L_per` in the subgoal 1 and subgoal 6. Therefore the variable `Person` can be unified to the variable `Per` and in a similar fashion the variable `RestGroup` to `Rest_group`.

```
T2 : collect_count(L_per,Data_St,C) :-
          head_tail(L_per,Person,RestGroup),
          test_1(Person,Salary),
          test_2(Salary,10000),
          build(Person,CollectRest,Data_St),
          collect_count(RestGroup,CollectRest,C1),
          test_1(Person,S),
          test_2(S,10000),
          C is C1 + 1.
```

By removing the syntactically identical subgoals (ie. `test_1/2` and `test_2/2`).

```
T2 : collect_count(L_per,Data_St,C) :-
          head_tail(L_per,Person,RestGroup),
          test_1(Person,Salary),
          test_2(Salary,10000),
          build(Person,CollectRest,Data_St),
          collect_count(RestGroup,CollectRest,C1),
          C is C1 + 1.
```

Similarly T_3 can be obtained:

```
T3 : collect_count(L_per,Collect,C) :-
          head_tail(L_per,Person,RestGroup),
          test_1(Person,Salary),
          test_3(Salary,10000),
          collect_per(RestGroup,Collect),
          head_tail(L_per,Per,Rest_Group),
          test_1(Per,_),
          test_3(S,10000),
          count(Rest_Group,C).
```

By applying the meta-folding operation we obtain the following clause:

```
T3 : collect_count(L_per,Collect,C) :-
          head_tail(L_per,Person,RestGroup),
          test_1(Person,Salary),
          test_3(Salary,10000),
          collect_count(RestGroup,Collect,C).
```

In a similar fashion T_4 can be obtained. By creating an instance of a join specification and by applying the unfolding operation we obtain the following clause:

```
T4 : collect_count(L_per,Collect,C) :-
          head_tail(L_per,Groups,RestGroups),
          collect_per(Groups,Collect1),
          collect_per(RestGroups,Collect2),
          append(Collect1,Collect2,Collect),
          head_tail(L_per,Gpos,Rest_Groups),
          count(Gpos,C1),
          count(Rest_Groups,C2),
          C is C1 + C2.
```

Applying the meta-folding operation produces the following clause: unifying **Groups** to **Gpos** and **RestGroups** to **Rest_Groups**.

```
T4 : collect_count(L_per,Collect,C) :-
          head_tail(L_per,Group,RestGroups),
          collect_count(Group,Collect1,C1),
          collect_count(RestGroups,Collect2,C2),
          append(Collect1,Collect2,Collect),
          C is C1 + C2.
```

A query for the program `int_count/3` is given as follows:

```
collect_count([person(maria,_,100000),person(wamb,_,200),
          person(elena,_,200000)],Collect,C)
```

The answer to the query gives:

`Collect=[person(maria,_,100000),person(elena,_,200000)]` and `C=2`.

This section has shown three examples which gave a different perspective on how this method can be used in the combination of programs.

The restrictions for the meta-composition method are as follows:

- programs must be derived from a skeleton that is either "meta-interpreter" or "count". Note that we do not include the set of skeletons defined in the traversal family in this restriction because they can be combined using join 1-1,
- the method only combines programs with the same number of clauses,
- both programs must traverse the same data structure,

Meta-composition (M_4)					
P_1/P_2	t	st	s	ctr	meta
t	ϕ_1, α				
st		ϕ_1, α			
s			ϕ_1, α		
ctr				α	
meta					α

Table 6.4: Table for the Meta-composition Method

- both programs must use the same pattern to construct/deconstruct the data structure determining flow of control,
- the meta-folding operation will only attempt to unify local variables which are in the same environment, which are used for traversing the same instance of the data structure.

The table 6.4 shows the performance of this method for each pair of programs (see section 6.3 for an explanation of the term α).

The condition ϕ_1 means that both programs belong to type Traverse-restricted (defined in Chapter 5). The resulting combined program is an efficient program for the counters and meta-interpreter classes. However the meta-composition can be used for combining other entries in the table under restriction ϕ_1. Note that the entries (t,t), (st,st) and (s,s) were already covered using the join 1-1 method. So the meta-composition method is more complex than strictly necessary for these.

6.9 The DS Method

The class of programs that we are considering to be combined using the DS method are those programs which operate over different data structures in some of their recursive cases. For example, consider the predicate `sum/2` which computes the sum of the elements

of a list or the sum of the elements of a binary tree. The second program `len/2` computes the length of a list. These programs are defined as follows:

```
P1 : sum([],0).
P2 : sum([H|T],Sum) :-
            sum(T,Sum1),
            Sum is Sum1 + H.
P3 : sum(tip(X),X).
P4 : sum(tree(R,L),Sum :-
            sum(L,Lsum),
            sum(R,Rsum),
            Sum is Lsum + Rsum.

Q1 : len([],0).
Q2 : len[H|T],Len) :-
            len(T,L1),
            Len is L1 + 1.
```

Note that this example corresponds to the (t,t) entry in the table 6.5.

The programs `sum/2` and `len/2` are combined by using a join specification defined as follows:

```
sum_len(List,Sum,Len) ⇐
            sum(List,Sum),
            len(List,Len).
```

Both programs operate over the same type of data structure, a list, and both programs are extensions of the same skeleton *traverse*, defined in Appendix C. The combined program `sum_len/3` can be obtained by applying the procedural join method defined earlier. The method works by combining the parts of each program which operate over the same data structure. In the example, these are the parts which traverse a list. Note that this method requires that the unification of the data structure be explicit on the head of each clause in order to produce groups. The reason for this is that analysis of the clauses would be difficult, in general because of data abstraction. The algorithm is described as follows:

DS Algorithm

1. Find the clauses which work over the same type of data structure and produce groups.
2. Combine clauses with the same data structure (i.e. in the same group).
3. Produce a new program which is formed by the combined clauses in each group.

The composed program obtained by applying the *DS* method is shown as follows:

```
sum_len([],0,0).
sum_len([H|T],Sum,Len) :-
            sum_len(T,S1,L1),
            Sum is S1 + H,
            Len is L1 + 1.

sum(tip(X),X).
sum(tree(R,L),Sum) :-
            sum(L,Lsum),
            sum(R,Rsum),
            Sum is Lsum + Rsum.

new_program(List,tree(X,Y),S,L,S_node) :-
            sum_len(List,S,L),
            sum(tree(X,Y),S_node).
```

The previous method can be seen as a variant of the procedural join method (defined in Section 6.7). The first stage is the selection of clauses which can be combined and then they are combined using the same sequence of transformations used on the procedural join method.

The table 6.5 shows the performance of the DS method (see section 6.3 for an explanation of the terms α and β).

The condition ϕ_3 means one of the program works over a different data structure than the other but does not put a restriction on the number of tests in the bodies of clauses.

An important subclass of programs which perform arithmetic operations in real/integer domains are those constructed using the *sum* or *count* technique. These appear frequently in Prolog text books ([O'Keefe 90, Bratko 86, Sterling & Shapiro 86]). The combination of these can be optimised using the particular method defined below.

DS (M_5)					
P_1/P_2	t	st	s	ctr	meta
t	ϕ_3, α	ϕ_3, β	ϕ_3, β		
st	ϕ_3, β	ϕ_3, α	ϕ_3, β		
s	ϕ_3, β	ϕ_3, β	ϕ_3, α		
ctr					
meta					

Table 6.5: Table for the DS Method

6.10 The Particular Method

The *particular* method is a method which combines programs defined in the same algebraic domain. This method copes with a special but common class of programs which involve arithmetic operations after the combined programs. The join specification is defined as: $T \Leftarrow P, Q, F_1 \ldots F_n$ where F_1 to F_n are subgoals which perform computations in the same algebraic domain as P and Q. A more detailed join specification is shown as follows:

$$T(\vec{IP}, \vec{IQ}, \vec{OT}) \Leftarrow P(\vec{IP}, \vec{OP}), Q(\vec{IQ}, \vec{OQ}), F(\vec{OP}, \vec{OQ}, \vec{OT})$$

The restriction for applying this method is that both of the programs P and Q should be constructed by using the *count* or *sum* technique. This imposes a restriction on the predicate F. This subgoal F needs to perform computations of the same type as the new subgoals added by means of the technique *count* or *sum*.

This method combines programs belonging to the type Traverse-general defined in Chapter 5. Besides this restriction we also require that both programs need to be constructed using technique sum or count or other *calculate* techniques. (Actually, the current implementation is restricted to just the count and sum techniques, but this could be easily extended by adding more rewrite rules whenever needed.)

This method works by performing the following sequence of transformation operations:

unfolding; application of laws such as commutativity or associativity which are valid in each algebraic domain; and finally folding (see Chapter 2). This method was implemented by creating a catalogue of transformation schemata tagged with conditions under which it is possible for the transformation to be performed. Each schema is represented in the system as a rewrite rule. We define these rewrite rules containing three components: the first component of the rule is the join specification, the second is the input schema, the third component is the transformed schema. The sequence of goals after the symbol ":-" are the conditions under which the schema can be transformed. For example, the first schema shown in Rule 1 is suitable when both of the programs to be combined perform computations which involve the addition operation.

In Rule 1 defined below we have that `Head_Df` is the head of the clause, `Op1` is the first operand in the join specification, `Op2` is the second operand in the join specification and the subgoal `S is TT +RR` is the extra subgoal which uses the partial results obtained from the execution of the predicate `Op1` and `Op2`. The subgoals `(G is F+C1)`, `(I is H+C2)` and `(E is G+I)` can be replaced by the expression `(E is S + (C1 + C2))`.

Rule 1:

```
rule((Head_Df :- Op1,Op2,S is TT+RR),
          (Op1,G is F+C1, Op2,I is H+C2, E is G+I),
          (Head_Df :- Op1,Op2, S is (F+H), E is S + (C1 + C2)))
          :- functor(Op1,NF1,AF1),
          functor(Op2,NF2,AF2), (NF1=NF2 ; NF1 \= NF2).
```

The second schema represented as Rule 2 is applicable when the third operand, in the join specification, involves the multiply operation. In a similar fashion `Head_Df` is the head of the clause (as before). `Op1` and `Op2` are the operands in the join specification.
`S is TT * RR` is the extra subgoal which uses the partial results obtained after the execution of the predicate `Op1` and `Op2`. The subgoals `(G is F*C1)`, `(I is H*C2)` and `(E is G*I)` can be replaced by the expression `(E is S + (C1 * C2))`.

RULE 2:

```
rule((Head_Df :- (Op1,Op2,S is TT*RR),
                 (Op1,G is F*C1, Op2,I is H*C2, E is G*I),
                 (Head_Df :- Op1,Op2, S is (F*H), E is S * (C1 * C2)))
                 :- functor(Op1,NF1,AF1),
                 functor(Op2,NF2,AF2), (NF1=NF2 ; NF1 \= NF2).
```

6.10.1 Algorithm

The algorithm for the *particular* uses transformation schemata for getting further optimisation in the combined program. The particular method exploits the set of subgoals which appear after the operands of the join specification $(F_1 \ldots F_n)$ in order to get a more optimised program.

Particular Algorithm

1. Create an instance T_k of the join specification $(T \Leftarrow P, Q, F)$.
2. Unfold P and Q in T_k with respect to P_i and Q_j with the result of unfolding being the clause:

   T_k :- $(P_{i,goal1}, ..., P_{i,goaln}), (Q_{j,goal1}, ..., Q_{j,goalm})\theta_R\theta_B$, $F_1, \ldots, F_n$.
3. The clause from step 2 can be rewritten as shown below. This is done in order to separate the set of subgoals which cannot be folded from the set of subgoals that can be folded. This reordering of subgoals is performed taking into consideration the dependence between variables (i.e. if a variable is instantiated later, it cannot be moved around and placed before). Note that in T_k the set of subgoals which do not need to be folded (added by i^{th} clause from program P) appears before $P_{i,goali}$. In a similar fashion the set of subgoals which do not need to be folded, added by the i^{th} clause from program Q, appears before $Q_{j,goali}$.

   $$T_k \text{ :- } (P_{i,goal1}, \ldots, P_{i,goali+1}, \ldots, P_{i,goaln}), P_{i,goali},$$
   $$(Q_{j,goal1}, \ldots Q_{j,goali+1}, \ldots, Q_{j,goalm}), Q_{j,goali}, F_1, \ldots, F_n.$$

4. Transform the previous schema into the schema shown below (T_k) which is the result of applying laws (commutativity, associativity, etc) over the sequence of subgoals which are not going to be folded and $F_1, \ldots, F_n$.

   T_k :- $P_{i,goali}Q_{j,goali}\Psi_1$
5. Apply the *folding* operation if there exists a pair of subgoals $P_{i,goali}$, $Q_{j,goali}$ and a set of subgoals $F_1, \ldots, F_n$ in the body of the clause T_k such that $(P_{i,goali}, Q_{j,goali}, F_1, \ldots, F_n)$ and (P,Q,F) unify with some mgu θ

   Then replace the subgoals $P_{i,goali}$, $Q_{j,goali}$ and $F_1, \ldots, F_n$ with T_k.
6. Repeat this process for all the clauses in program P and program Q.

The restriction of the algorithm described above is that both of the programs need to be created by using the *count* or *sum* techniques. However this could be generalised to other types of arithmetic operations such as union and intersection which are commutative and associative.

In the next section we have an example of a combined program which computes the length of two lists at the same time. The programs can also be combined using the *general* method but an optimised version is obtained using the *particular* method.

6.10.2 Example

For example, consider the problem of the construction of the combined program `len_two/3` which computes the length of two lists and produces the total length using the following definition of the predicate `len/2`.

```
len([],0).
len([_H|T], Len) :-
                  len(T,Lenax),
                  Len is Lenax + 1.
```

Note that this example corresponds to the (t,t) entry in the table 6.6.

The information that program `len/2` was constructed using the count technique can be inferred from the seventh argument of the program history. This argument keeps a record of which techniques were used during the construction of the program.

The join specification used for producing the programs `len_two/3` is as follows:

```
len_two(L1,L2, Len) ⇐
                  len(L1,Len1),
                  len(L2,Len2),
                  Len is Len1 + Len2.
```

This join specification uses more than a single argument for flow of control but both belonging to the same type of data structure.

The process of combination using the `particular` method can be described as follows:

Step 1: taking clause 1 from program `len/2` and clause 1 from program `len/2` we can create an instance of the join specification $T_1 :- P, Q, F_1$. By unfolding P and Q with respect to P_1 and Q_1. we obtain:

```
len_two([],[],Len) :- true, true, Len is 0+0.
```

Figure 6.2 shows a summary of the combining process described above, i.e. the resulting combined clause number 1 for program P (denoted as [P,1]) with first clause of program

Q (denoted as [Q,1]). In this diagram, the sequence of triangles gives the succession of transformations to the body of the combined clause. Below the first triangle we have the unfolded clauses [P,1] and [Q,1]. Below the second triangle we have the unified values for the variables and finally we get a value for the variable Len after applying arithmetic and logical reductions. These reductions are encoded in the transformation schema.

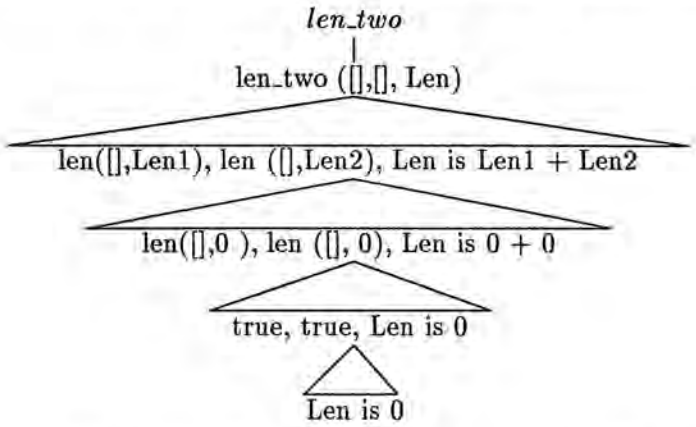

Figure 6.2: Clause 1 in the new program len_two/3

Step 2: we take clause 2 from program `len/2` and clause 1 from program `len/2` and create an instance of the join specification $T_2 :- P, Q, F_1$. Unfolding P and Q with respect to P_2 and Q_1 we get the clause that is shown below.

```
len_two([],[_H2|T2],Len) :-
                    true,
                    len(T2,Lenx),
                    Len2 is Lenx +1,
                    Len is 0+Len2.
```

A transformation schema is used at this stage in order to simplify the body of the previous clause. The transformation schema takes the expressions `Len is 0+Len2` and `Len2 is Lenx+1` and produces an equivalent expression. Returning to the unfolded clause and replacing the two goals

{ `Len2 is Lenx+1, Len is 0+Len2` } with the final expression { `Len is Lenx+1` } we get the clause:

```
len_two([],[_H2|T2],Len) :-
                    true,
                    len(T2,Lenx),
                    Len is Lenx+1.
```

This transformation process is performed using a transformation schema such as are defined on page 143. This transformation schema are rewrite rules which are used in order to achieve more efficiency in the combined program.

Performing boolean reductions (using rewrite rules) on the previous clause we obtain the following clause:

```
len_two([],[_H2|T2],Len) :-
                    len(T2,Lenx),
                    Len is Lenx+1.
```

The Figure 6.3 in similar fashion shows a summary of the previous stages. In this figure we have combined clause number one of program P ([P,1]) with the second clause of program Q ([Q,2]). Below the first triangle we have the instance of the join specification for clauses [P,1], [Q,2] and F_1 denoted as T_2. At the next stage we have the unfolded clauses [P,1] and [Q,2]. Below the third triangle we have the transformed clause T_2 after applying arithmetic rules and finally below the fourth triangle we have the definition of T_2 after applying boolean rules. Because it is not possible to apply any further optimisation, this definition of T_2 will be the second clause for the combined program.

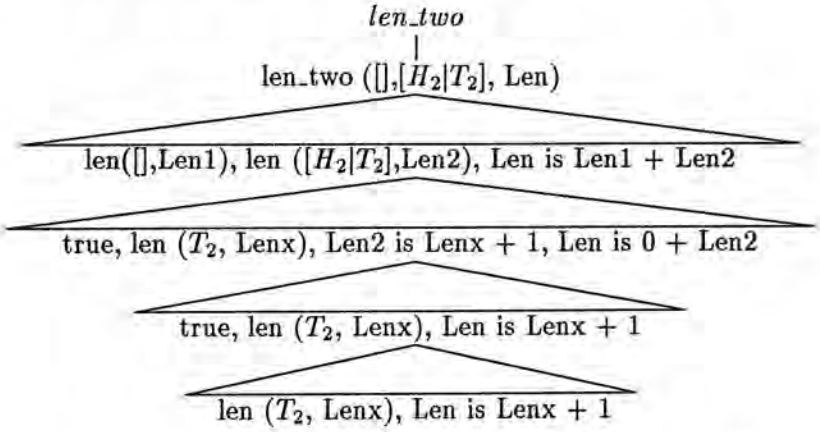

Figure 6.3: Clause 2 in the new program len_two/3

Step 3: in the same way we have the third combined clause which is:

```
len_two([_H1|T1],[],Len) :-
                len(T1,Lenx),
                Len is Lenx+1.
```

Figure 6.4 shows the combined clause [P,2] with [Q,1] according to the instance of the join specification T_3 shown below the first triangle. The combination process is the same as was described for the clause T_2.

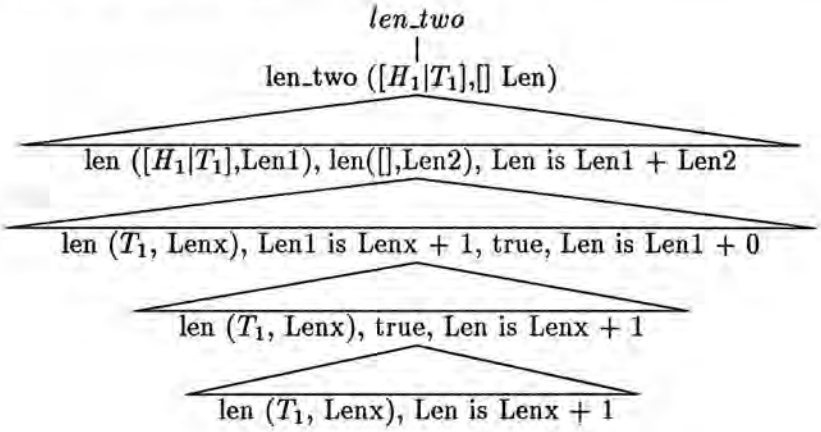

Figure 6.4: Clause 3 in the new program len_two/3

Step 4: we take clause 2 from program P and clause 2 from program Q and create an instance of the join specification $T_4 :- P, Q, F_1$. Unfolding P and Q with respect to P_2 and Q_2 we obtain the clause that is shown below.

```
T4 :len_two([_H1|T1],[_H2|T2],Len) :-
                len(T1,Lenx1),
                Len1 is Lenx1+1,
                len(T2,Lenx2),
                Len2 is Lenx2+1,
                Len is Len1+Len2.
```

Clause T_4 matches with the transformation schema used when the technique *count* was applied (on both programs). This transformation schema is shown below.

$Op1(T1, F),\ G\ is\ F + C1,\ Op2(T2, H),\ I\ is\ H + C2,\ E\ is\ G + I$ then

the program can be transformed to

$Op1(T1, F),\ Op2(T2, H),\ S\ is\ F + H,\ E\ is\ S + (C1 + C2)$

were $T1, G, F, C1, T2, I, H, C2, E$ and S are local variables; $Op1$ and $Op2$ are the operands in the join specification. The local variable S is a new variable which replaces all the local variables used in the predicates $Op1$ and $Op2$ (i.e. F and H) from the expression E *is* $G+I$.

The approach used is straightforward, we apply a sequence of rewrite rules to arithmetic expressions. Then these are converted in a form in which the fold operation can be applied. The sequence of rewrites for clause T_4 are defined below. Note that the rewrites used currently in our system are limited but might be extended (although we do not have proof of generality).

The transformation process starts by grouping subgoals which are computing arithmetic expressions. This reordering preserves the dependence between variables. Then given the following subgoals:

(`Len is Len1+Len2`) and (`Len1 is Lenx1+1`) and (`Len2 is Lenx2+1`).

we obtain that `Len is Lenx1+1 + Lenx2+1`. After using associativity and commutativity laws we get the expression: `Len is (Lenx1+Lenx2)+(1+1)`.

This expression can be rewritten as follows:

```
Len is (Lenx1+Lenx2)+2
```

Introducing a new variable `Lenax`, and equating it to (`Lenx1+Lenx2`) we obtain the following expression:

```
Len is Lenax+2
```

The axiom that our transformation schema uses is defined as follows: $\forall n_1, n_2 \ \exists n_3$ such as $n_3 = n_1 + n_2$ where n_1, n_2 and n_3 are integers/reals.

Therefore the clause T_4 can be rewritten as:

```
len_two([_H1|T1],[_H2|T2],Len) :-
            len(T1,Lenx1),
            len(T2,Lenx2),
            Lenax is Lenx1+Lenx2,
            Len is Lenax+2.
```

Note that all the previous transformation stages described above for transforming T_4 are encoded in the rewrite rule. This transformation process can be safely applied to local variables which are performing the same kind of computations on integer or real numbers. Note that this only can be done because in both programs either the technique *count* or *sum* was applied. For example, if one of the programs was constructed using the *sum* technique and the other program using the multiply technique (which adds subgoals involving the multiply operation) then we cannot make any further reduction of variables, using our current set of rewrite rules, and no more optimisation can be done.

Applying the folding operation to the previous clause we obtain the combined clause.

```
len_two([_H1|T1],[_H2|T2],Len) :-
                len_two(T1,T2,Lenax),
                Len is Lenax+2.
```

Figure 6.5 shows a summary of the stages described earlier for combining the second clause from each program. The instance of the join specification T_4 using [P,2], [Q,2] and F_1 appears under the first triangle. Again, below the second triangle we have the unfolded clause [P,2] with [Q,2]. The equation described under the third triangle is the resulted T_4 after the transformation schema was applied. Finally, below that the fourth triangle we have the resulting clause after the folding process is carried out by the *particular method.*

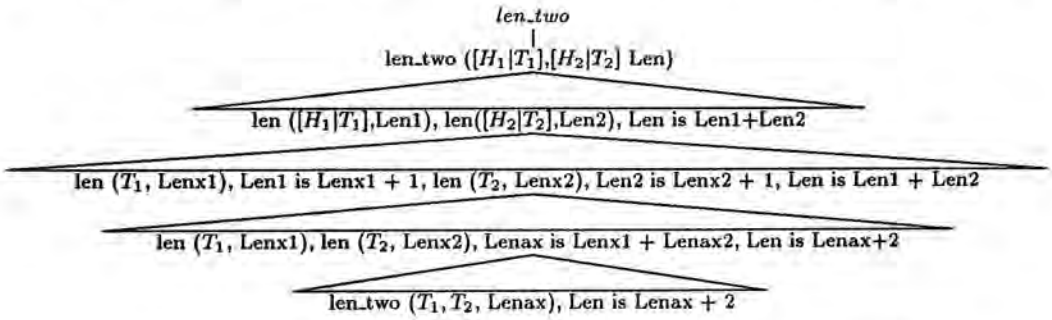

Figure 6.5: Clause 4 in the new program len_two/3

The listing of the whole program is shown as follows:

```
T1 : len_two([],[],0).
T2 : len_two([],[_H2|T2],Len) :-
          len(T2,Lenx),
          Len is Lenx+1.
T3 : len_two([_H1|T1],[],Len) :-
          len(T1,Lenx),
          Len is Lenx+1.
T4 : len_two([_H1|T1],[_H2|T2],Len) :-
          len_two(T1,T2,Lenax),
          Len is Lenax+2.
```

The main characteristic of the program `len_two/3` generated by using the 'particular' method is that this program cannot be generated by any of Sterling et. al.'s methods. The reason is that we are taking advantage of the fact that we have information about which techniques were used in the construction of the programs, so we can apply arithmetic laws which are valid for specific domains (integer/real numbers), obtaining a more optimised combined program than `len_two/3` generated by using the 'general' method (see page 155).

Let consider us the query `len_two([1,2,3],[4,5,6,7],X)`. The answer to the query gives `X=7`.

The restrictions for the particular method are as follows:

1. programs are derived from the traverse, short_traverse or search skeleton,
2. both have the same number of clauses,
3. both programs have zero or different tests,
4. both traverse the same type of data structure,
5. both programs were constructed either using the *count* or *sum* technique. Therefore the transformation process based on arithmetic properties (described above) can be safely applied to local variables because these variables are performing computations on integer or real numbers.

The restrictions 1-4 given above are inherited from the type Traversal-general to which the pairs of programs should belong in order to be combined with this method. The

Particular (M_6)					
P_1/P_2	t	st	s	ctr	meta
t	ϕ_4, α	ϕ_4, β	ϕ_4, β		
st	ϕ_4, β	ϕ_4, α	ϕ_4, β		
s	ϕ_4, β	ϕ_4, β	ϕ_4, α		
ctr					
meta					

Table 6.6: Table for the Particular Method

restriction 5 is required for further optimisation. These restrictions can be inferred from the program history so no user interaction is required in the particular method. In Burstall and Darlington's system all the transformation rules and arithmetic laws which are valid during the transformation need to be provided by the user.

The table 6.6 shows the performance of the particular method (see section 6.3 for an explanation of the terms α and β).

The condition ϕ_4 means that both programs (taken clause by clause) have zero or different tests and that both program were constructed either using the *sum* or *count* technique.

This method is very efficient for programs which perform this particular class of arithmetic computations. We next consider an extension of the particular method which, although producing slightly less elegant combined programs, places fewer restrictions on the type of program which it can combine. This inspires the general method shown in next section.

6.11 The General Method

The General method automatically builds programs that have a user specification formed from two operands plus extra subgoals without any restriction on the form of the subgoals F_i (like in the particular method). We make a distinction in this case between the join specification and the user specification (see definition on page 15). The *join specification* works over two operands, as in the previous methods, but the *user specification* can be

formed from two operands plus extra subgoals. This user specification can be defined as $T \Leftarrow P, Q, F_1, \ldots, F_n$ where P and Q are join operands, T is the join target and $F_1, \ldots, F_n$ are subgoals which use the output results from predicate P and Q for performing new computations.

In this thesis we distinguish between a *user specification* and *join specification.* A ***user specification*** is the formal definition of the program that the user wants to build and a ***join specification*** is the actual specification which is used by the composition system for combining the programs.

The General method works by selecting two operands from the user specification in which it is possible to perform the composition process. The second stage is to combine these two operands by using a join specification. In the current implementation the two predicates that will be the two operands for the join specification are given as the first and second operands of the user specification.

The *general* method uses the following user specification:

$$T(\vec{I_P}, \vec{I_Q}, \vec{O_T}) \Leftarrow P(\vec{I_P}, \vec{O_P}), Q(\vec{I_Q}, \vec{O_Q}), F(\vec{O_P}, \vec{O_Q}, \vec{O_T}).$$

where the restriction on F in the *particular* method does not apply. This method works by combining program P and program Q to produce a combined program P_Q and finally builds the combined program T which is as follows:

$$T(\vec{I_P}, \vec{I_Q}, \vec{O_T}) :- P\_Q(\vec{I_P}, \vec{O_P}, \vec{I_Q}, \vec{O_Q}), F(\vec{O_P}, \vec{O_Q}, \vec{O_T}).$$

6.11.1 Algorithm

The algorithm shown below is an extension of procedural join but can easily be added to other methods. The justification for the implementation of this method arises from the fact that sometimes the user needs to combine two programs as a first stage of the software

development process and as a second stage just needs to add extra computations using the values computed in the first stage. This method gives this facility automatically.

General Algorithm

1. Definition of a *user specification* defined as:
   $T \Leftarrow \gamma_1, \gamma_2, \cdots, \gamma_n$
   where each γ_i is a call.
2. The system chooses two operands in the *user specification*. In this case γ_1 and γ_2.
3. Select a *join specification*. In this stage, if the system has a join specification for combining γ_1 and γ_2 it will offer this to the user. Otherwise the system provides an interactive interface for defining the new join specification.
4. Apply the most appropriate method to pair of programs to be combined.

6.11.2 Example

For example, we may want to build `len_two/3` using the user specification defined below. In this *user specification*, the user describes the intended program that he wants to build. The join specification should be automatically formed by taking two operands from the user specification. In this particular example the join specification (formed using the first two positions in the user specification) will be `len_len/4` which is defined later in the text. Note that this example corresponds to the (t,t) entry in the table 6.7.

```
len_two(List1,List2,Len) ⇐
            len(List1,Len1),
            len(List2,Len2),
            Len is Len1 + Len2.
```

The algorithm takes `len/2` as the first operand and as the second operand. As a first stage in the combining process the composition system creates the program `len_len/4` by using the following join specification:

```
len_len(List1,List2,Len1,Len2) ⇐
            len(List1,Len1),
            len(List2,Len2).
```

The combined program `len/4` is obtained by using the procedural join method. This program is shown as follows:

```
T1 : len_len([],[],0,0).
T2 : len_len([],[_H2|T2],0,Len2) :-
          len(T2,Lenx2),
          Len2 is Lenx2 + 1.
T3 : len_len([_H1|T1],[],Len1,0) :-
          len(T1,Lenx1),
          Len1 is Lenx1 + 1.
T4 : len_len([_H1|T1],[_H2|T2],Len1,Len2) :-
          len_len(T1,T2,Lenx1,Lenx2),
          Len1 is Lenx1 + 1,
          Len2 is Lenx2 + 1.
```

The resulting combined program `len_two/3` is shown as follows:

```
len_two(List1,List2,Len) :-
          len_len(List1,List2,Len1,Len2),
          Len is Len1 + Len2.
```

The utility of the general method can be shown with an example in which two operands and two subgoals appear in the user specification. The example that we want to build is the program `average/2` using `sum/2` and `count/2` with a user specification that as defined below.

```
average(L,Av) ⇐ sum(L,Sum), count(L,Count), Count \+ 0, Av is Sum/Count.
```

The combined program that the system will produce is a program `average/2` using `sum_count/3` which was described in section 6.6.2. The piece of code for the program `average/2` is shown as follows.

```
average(List,Av) :-
          sum_count(List,Sum,Count),
          Count \+ 0,
          Av is Sum / Count.

sum_count([],0,0).
sum_count([A|B],C,D) :-
          sum_count(B,E,F),
          C is E + A,
          D is F + 1.
```

General (M_7)					
P_1/P_2	t	st	s	ctr	meta
t	ϕ_5, α	ϕ_5, β	ϕ_5, β		
st	ϕ_5, β	ϕ_5, α	ϕ_5, β		
s	ϕ_5, β	ϕ_5, β	ϕ_5, α		
ctr					
meta					

Table 6.7: Table for the General Method

The performance of this method is shown in the table 6.7. (See section 6.3 for an explanation of the terms α and β).

The condition ϕ_5 means that both programs (taken clause by clause) have zero or different tests. Also this method does not restrict to combine programs with restrictions in the subgoals $F_1, \ldots, F_n$ which appears in the user specification (like in the particular method). In particular we can use it for combining programs in which only one program was constructed either using the sum or count technique or none of them were constructed using the sum or count technique.

This method can be used for the pairs of programs (t,t), (st,st) and (s,s) when only one of the programs was constructed using the *sum* or *count* technique.

7

Methods for Different Flows of Control

This chapter gives the set of methods for combining programs with slightly different flow of control. In a similar fashion to Chapter 6, we also discuss the performance of each of the combining methods by analysing their behaviour for each pair of programs from our hierarchy.

7.1 Mutants

This method combines programs with different flows of control. However there is a restriction. The mutant method combines programs which are either in the same mutant class or when one class is the mutant version of the other. This means that it is not possible to combine a mutant of traverse with a mutant of meta-interpreters.

Assumptions

The following set of assumptions are required by the algorithm which tackles the mutants problem.

1. Program P_1 and program P_2 were created using the same skeleton but one or both are mutants of this skeleton.

2. The control of the initial skeleton is a subset of the control of the new program. This means that when we are applying techniques none of these techniques will delete any subgoal that appears in the flow of control of the skeleton.

3. Both programs operate over the same type of data structure.

4. The mutant must have been constructed by using techniques (defined in our knowledge base of techniques defined in Appendix D) which do not modify the way in which the two initial programs traverse the data structure. This means that we are only including extra computations which may conditionally terminate a clause or that add new clauses which process conditions not checked by the initial program.

7.1.1 Algorithm

The *mutant* method is based on our meta-composition method described above. Therefore it imposes the needed synchronisation during the traversal of the data structure for the corresponding clauses. This method handles the problem of extra clauses which do not have a corresponding clause with the skeleton used in its construction. The solution for these mutant clauses is described in the algorithm below. The algorithm creates an instance of the join specification T_k and the values of variables are passed to this instance T_k. As a second stage one of the predicates is unfolded. The selection of which predicate is unfolded is determined as follows: if the mutant clause belongs to program P then P is unfolded, otherwise if the mutant clause belongs to program Q then Q is unfolded and then the meta-folding operation is performed. This process is repeated while there are still mutant clauses in each program. The unfolding operation is applied through the join specification, and each proposed mutant combined clause is offered to the user. This offered clause can be accepted or rejected.

User interaction is required at this stage in order to determine the behaviour of the com-

bined mutant clause. The functionality for the combined clause should not be automated by the system, since more than one option can be chosen according to the wishes of the user. There are two ways to change the behaviour of the offered combined clause: one is that the variables can be unified to specific values according with the wishes of the user and, secondly, subgoals in the body of the clause can be changed if the user wants to conditionally terminate a clause. For instance, the user could redefine the value of the variable used to control the depth of the proof search space with the aim of reducing the number of subgoals to be proved.

The mutant algorithm is as follows, assuming that program P contains the mutant clauses:

Mutant Algorithm

1. Create an instance T_i of the join specification $(T \ :- \ P, Q)$.
2. Unfold P in T_i with respect to P_i a clause of predicate P.
   If P and $P_{i,head}$ unify with the substitution θ_P and P_i is defined as follows:

   $$P_i \ :- \ A_1, \ldots, A_n$$

   Then replace P with $P_{i,Body}$ under substitution θ_P to produce T_i:

   $$T_i \ :- \ ((A_1, \ldots, An), Q)\theta_P \quad with \ \ n > 0$$

3. Apply the meta-fold operation which use the key points information (obtained from the program history) for performing the folding of subgoals.
4. Ask the user to approve the proposed combined clause.
5. Repeat the same process for each mutant clause that appears in program P.

This process can be described in the same way for Q in the case that Q is the mutant clause. Note that we have two cases depending on which is the mutant clause: if the mutant clause belongs to program P then the values of the variables in the head of the clause T_i are passed to Q. In the second case, if the mutant clause belongs to program Q then the values of the variables in the head of the clause T_i are passed to P.

The mutant method requires extra knowledge about which are the mutant clauses. This

information is provided by the *program history*.

7.1.2 Example 1

This example shows a simple combination of two programs with slightly different flow of control. They are combined using the mutant method which controls the unfold operation, and so results in a more optimised combined program (see second stage in the combination process for this example).

Our initial programs are the programs `get_odd/2` and `count/2`. The program `get_odd/2` is a mutant program based on the traverse skeleton which selects the odd numbers from a list and the program `count/2` is an extension of the traverse skeleton which counts the length of the list. Note that this example corresponds to the (mut-t,t) entry in the table 7.1.

```
1: get_odd([], []).
2: get_odd([H|T], [H|O]) :-
       odd(H),
       get_odd(T, O).
3: get_odd([H|T], O) :-
       even(H),
       get_odd(T, O).
```

```
4: count([],0)
5: count([H|T], Count) :-
       count(T,C1),
       Count is C1+1.
```

The program histories are provided using the relation his_prog/9 defined below.

```
his_prog(get_odd,type_mutant-traverse,2,traverse,3,2,no
          [[1,(traverse(T),get_odd(T,O))],[2,(traverse(T),get_odd(T,O))]],
          [[3,(traverse(T),get_odd(T,O))]]).
his_prog(count,type_traverse,2,traverse,2,0,sum_technique,
          [[1,(trav([]),count([],0))],[2,(trav(T),count(T,C1))]],nil).
```

A new program `count_odd/2`, which selects the odd numbers and computes the length of the list at the same time, is generated using the following join specification:

```
6: count_odd(L, C) ⇐ get_odd(L, Os), count(Os, C).
```

Note that this method synchronises the list search performed in `get_odd/2` and `count/2` i.e. enforces the same traversal of the data structure in the clauses.

Firstly, by applying the unfold and fold operation to the definition of `get_odd/2` and `count/2` using the join specification (`count_odd/2`) we obtain clauses 7 and 8 shown below:

```
7:  count_odd([], 0).
8:  count_odd([H|T], C) :-
            odd(H),
            get_odd(T, Os),
            count(Os, COs),
            C is COs+1.
```

Secondly, the system realises that third clause does not have a corresponding clause in the second program (`count/2`) by looking at the program histories, so the system unfolds the first operand `get_odd(L,Os)` of the join specification ($T \Leftarrow P, Q$) and adds to this the second operand of the join specification (`count(Os,C)`). This produces clause 9:

```
9:  count_odd([H|T], C) :-
            even(H),
            get_odd(T, Os),
            count(Os, C).
```

This decision is taken by reading the *program history* which states in its ninth parameter that clause 3 is the mutant clause (for further details see Chapter 5). In this way the composition system avoids the generation of 2 clauses which are redundant in the combined program. These redundant clauses could be obtained if the system unfolds the call count/2 in clause 9 using the procedural join method (which does not control which subgoals can be unfolded).

```
count_odd([], 0).
count_odd([H|T], C) :-
            odd(H),
            get_odd(T, Os),
            count(Os, COs),
            C is COs+1.
count_odd([H|T], C) :-
            even(H),
            get_odd(T, Os),
            count(Os, C).
```

Finally, applying the fold operation (using the join specification) will obtain the following clause:

```
count_odd([], 0).
count_odd([H|T], C) :-
            odd(H),
            count_odd(T, COs),
            C is COs+1.
count_odd([H|T], C) :-
            even(H),
            count_odd(T, C).
```

The system should ask if the user accepts the combined clause number 9. Assuming that the user accept it. Then the combined remains as shown above. This approach is possible if technique descriptions of programs are conveniently available from the techniques editor.

7.1.3 Example 2

The following example shows the combination of two mutants belonging to the meta-interpreters class. The example below shows how the program `explain/4` is obtained by performing the combination process twice: first we combine `proof/2` and `depth/2` obtaining `proof_depth/3`; then we combine a mutant of the program `proof_depth/3` and `result/2` obtaining the program `explain/4` defined on page 168. Note that this example corresponds to the (mut-meta,mut-meta) entry in the table 7.1.

The programs used in this example were taken from [Sterling & Shapiro 86]. The meta-interpreter `result/2` was built using the modulant skeleton `solve/1` defined in Appendix C. This meta-interpreter contains one (extra) clause more than the skeleton used in its construction (P_5) so is a mutant of that skeleton. The meta-interpreter `result/2` always succeeds. The relation `result(Goal,Res)` is true if `Goal` succeeds and `Res=yes` and also if there is a failure branch and `Res=no`.

```
P1 : result(true,yes).
P2 : result((A,B),Out) :-
            result(A,Out1),
            result_conj(Out1,B,Out).
P3 : result(A,yes) :-
            sys(A),
            call(A).
P4 : result(A,Out) :-
            clause(A,B),
            result(B,Out).
```

```
P5 : result(Struct,no):-
          \+ processed_above(Struct).

P6 : result_conj(yes,Goals,Out) :-
          result(Goals,Out).
P7 : result_conj(no,Goals,no).

P8 : processed_above(A) :-
          A = true ;
          A = (_,_);
          sys(A);
          clause(A,_).
```

The program **proof_depth/3** is the result of combining two meta-interpreters **proof/2** (which returns the proof tree associated with the proof of a goal) and **depth/2** defined below. The meta-interpreter depth/2 can be used only if the variable D is instantiated to a number. This value gives the recursion depth limit up to which the program should try to compute the proof tree for a query.

The program **proof_depth/3** is combined using the join specification given as follows:

```
proof_depth(Goal,Proof,Depth) ⇐
          proof(Goal,Proof),
          depth(Goal,Depth).
```

The meta-interpreters proof/2 and depth/2 are given below:

```
proof(true,true).
proof((A,B),(PA,PB)) :-
          proof(A,PA),
          proof(B,PB).
proof(A,sys) :-
          sys(A),
          call(A).
proof(A,(A :- PA)) :-
          clause(A,B),
          proof(B,PA).
```

```
depth(true,D).
depth((A,B),D) :-
            depth(A,D),
            depth(B,D).
depth(A,D) :-
            sys(A),
            call(A).
depth(A,D) :-
            clause(A,C),
            D > 0,
            E is D - 1,
            depth(C,E).
```

The stages of the composition process can be described as follows:

Firstly, we take the first clause of program `proof/2` and the first clause of program `depth/2` and create an instance of the join specification $T_1 :- P, Q$. By unfolding P and Q using P_1 and Q_1 we get the first combined clause:

```
proof_depth(true,true,_D).
```

Secondly, using the second clause of program `proof/2` and the second clause of program `depth` to construct an instance of the join specification $T_2 :- P, Q$. Unfolding P and Q using P_2 and Q_2 we get the clause shown below.

```
proof_depth((A,B),(PA,PB),D) :-
            proof(A,PA),proof(B,PB),
            depth(A,D), depth(B,D).
```

Folding `proof(A,PA)` with `depth(A,D)` we find that the two previous goals can be substituted by `proof_depth(A,PA,D)`. In the same way, `proof(B,PB)` and `depth(B,D)` can be replaced by `proof_depth(B,PB,D)`. Therefore the resulting combined clause is show as follows:

```
proof_depth((A,B),(PA,PB),D) :-
            proof_depth(A,PA,D),
            proof_depth(B,PB,D).
```

Thirdly, we take the third clause of each program and construct an instance of the join specification $T_3 :- P, Q$ After unfolding P and Q using P_3 and Q_3 we obtain:

```
proof_depth(A,sys,_D) :-
            sys(A),
            call(A),
            sys(A),
            call(A).
```

Applying a merge operation which removes all syntactically identical subgoals we get the third combined clause.

```
proof_depth(A,sys,_D) :-
            sys(A),
            call(A).
```

Next, we take the fourth clause of each program and get an instance of the join specification $T_4 :- P, Q$. Unfolding P and Q with respect to P_4 and Q_4 we produce the following clause:

```
proof_depth(A,(A :- PA),D) :-
            clause(A,B),
            proof(B,PA),
            clause(A,B1),
            D > 0,
            E is D - 1,
            depth(B1,E).
```

We cannot fold any subgoals in the body of the previous clause using only the join specification. One candidate for folding is `proof(B,PA)` and `depth(B1,E)`. At this stage we need the knowledge about the initial skeleton and how the subgoals in the skeleton were transformed during the construction of the program. This knowledge is kept in the eighth argument of the *program history*.

```
his_prog(proof,2,...,[[1,(true,true),no_test],
            [2,(solve(A),proof(A,PA)), (solve(B),proof(B,PB)),no_test]
            [3,(sys(A),sys(A)),(call(A),call(A)),no_test],
            [4,(solve(A),proof(B,PA)),no_test]],nil).

his_prog(depth,2,...,[[1,(true,true),no_test],
            [2, (solve(A),depth(A,D)),(solve(B),depth(B,D)),no_test],
            [3,(sys(A),sys(A)),(call(A),call(A)),no_test],
            [4,(solve(A),depth(C,E)),D>0]],nil).
```

Using this information we know that we can fold the subgoals `proof(B,PB)` and `depth(B1,E)` because in this case both subgoals are enhancements of the subgoal `solve(A)`

(see clause number 4) and the variables B and B1 are used with the same functionality (i.e. deconstruct the data structure). If they are derived from the same subgoal in the skeleton and both programs are traversing the same data structure then they have the same functionality. The fourth combined clause is presented as follows:

```
proof_depth(A,(A :- PB),D) :-
            clause(A,B),
            clause(A,B),
            D > 0,
            E is D - 1,
            proof_depth(B,PB,E).
```

The second subgoal can be deleted using the operation *merge* producing the final fourth combined clause.

```
proof_depth(A,(A :- PA),D) :-
            clause(A,B),
            D > 0,
            E is D - 1,
            proof_depth(B,PA,E).
```

The complete program `proof_depth/3` which builds a proof tree and limits the depth at the same time is shown as follows:

```
Q1 : proof_depth(true,true,_D).
Q2 : proof_depth((A , B),(PA ,PB),D) :-
            proof_depth(A,PA,D),
            proof_depth(B,PB,D).
Q3 : proof_depth(A,sys,_D) :-
            sys(A),
            call(A).
Q4 : proof_depth(A,(A :- PA),D) :-
            clause(A,B),
            D > 0,
            E is D - 1,
            proof_depth(B,PA,E).
```

A mutant program is obtained by adding an extra clause Q_2 to the combined program `proof_depth/3` listed above. This new program `proof_depth/3` stops computing the proof tree after the depth bound is met and it uses the atom `overflow` as the last goal in the returned proof tree. So this program addresses the problem of goals which exceed the depth bound in the proof.

```
Q1 : proof_depth(true,true,_D).
Q2 : proof_depth(_A,overflow,0).
Q3 : proof_depth((A,B),(PA,PB),D)):-
          proof_depth(A,PA,D),
          proof_depth(B,PB,D).
Q4 : proof_depth(A,sys,D) :-
          sys(A),
          call(A).
Q5 : proof_depth(A,(A :-PA),D)):-
          clause(A,B),
          D > 0,
          E is D - 1,
          proof_depth(B,PA,E).
```

If we want to run this program the variable D needs to be instantiated. For instance we can ask the query: `proof_depth(Goal,Proof,5)`, where `Goal` is some goal which we expect to be able to prove within depth limit 5. Let consider us the query `proof_depth(son(X,haran),Proof,5)` using the following rules:

```
son(X,Y) :-
          ((father(Y,X)),
          (male(X))).

father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).
male(isaac).

male(lot).
female(milcah).
female(yiscah).
```

The answer to the query gives:

`son(lot,haran):- (father(haran,lot):-true),(male(lot):-true).`

Returning to the construction of the meta-interpreter `explain/4`, this will be constructed using the two meta-interpreters `result/2` and `proof_depth/3` defined above and by using the following join specification:

`explain(Goal,Out,P,D)` ⇐ `result(Goal,Out), proof_depth(Goal,P,D).`

Note that the mutant method enforces the synchronization of the traversal of the data structure for corresponding clauses, and hence the proof `P` is forced to correspond to the output `Out`.

The process consists of taking clause 1 from program `result/2` and clause 1 from program `proof_depth/2`. By applying the unfolding operation we obtain the following clause:

```
explain(true,yes,true,_D) :- true, true.
```

which is equivalent to following clause:

```
explain(true,yes,true,_D).
```

Secondly, we have the mutant clause which we introduced into program `proof_depth/3`. The way that we deal with this is to create an instance $T_2 :- P, Q$. The clause Q_2 is a mutant clause which does not have a corresponding clause with program P. Doing unfolding of Q with respect to Q_2, Q_2 unifies with the substitution $\theta_Q = \{$`overflow/P,0/D`$\}$. Then by replacing $P_{2,body} =$ `true` in T_2 we obtain following clause:

```
explain(Goal,Out,overflow,0).
```

This clause is proposed by the combination system and could be accepted or rejected by the user. If the clause is rejected it must be redefined by the user in order to obtain the most suitable clause for his purpose (ie. the final functionality depends of the application that the user wants to build). For instance the previous clause can be redefined as follows:

```
explain(Goal,overflow,overflow,0).
```

By this redefinition the program is able to return the proof tree, until the bound is reached, instantiating the result label to the atom overflow.

Thirdly, taking clause 2 from program `result/2` and clause 3 from program `proof_depth/3` we can create the instance of the join specification. By unfolding P and Q with respect to P_2 and Q_3 in T_3, we obtain the clause shown below.

```
explain((A,B),Out,(PA,PB),D) :-
            result(A,Out1),
            result_conj(Out1,B,Out),
            proof_depth(A,PA,D),
            proof_depth(B,PB,D).
```

Applying the folding operation the subgoals `result(A,Out1)` and `proof_depth(A,PA,D)` can be combining together obtaining `explain(A,Out1,PA,D)`

```
explain((A,B),Out,(PA,PB),D) :-
            explain(A,Out1,PA,D),
            result_conj(Out1,B,Out),
            proof_depth(B,PB,D).
```

At this stage we need to handle the modulation of the skeleton `solve/1`. The subgoals `result_conj(Out1,B,Out)` and `proof_depth(B,PB,D)` are combined together using a second join specification defined below. This fact is deduced from the program history. In it we have recorded that the program `result/2` is created using the meta-interpreter modulant skeleton (defined in Appendix C) and from knowing the skeleton we deduce that the clause number 2 in program result/2 calls in its body an auxiliary predicate which can be used for further optimisation using another join specification. This join specification is defined by the user interactively.

```
explain_conj(Result1,Goal,Result,P,D) ⇐
            result_conj(Result1,Goal,Result),
            proof_depth(Goal,P,D).
```

Applying the folding operation using the second join specification (`explain_conj`) the subgoals `result_conj(Out1,B,Out)` and `proof_depth(B,PB,D)` can be combined together to obtain `explain_conj(Out1,B,Out,PB,D)`. So our third combined clause is as follows:

```
explain((A,B),Out,(PA,PB),D) :-
            explain(A,Out1,PA,D),
            explain_conj(Out1,B,Out,PB,D).
```

Fourthly, taking clause 3 from program `result/2` and clause 4 from program `proof_depth/3` we can create the instance from the join specification $T_4 :- P, Q$. Unfolding in T_4 P and Q with respect to P_3 and Q_4, we obtain the clause that is shown below.

```
explain(A,yes,sys,_D) :-
            sys(A),
            call(A),
            sys(A),
            call(A).
```

Applying the merge operation, we obtain the following clause:

```
explain(A,yes,sys,_D) :-
            sys(A),
            call(A).
```

Fifthly, taking clause 4 from program `result/2` and clause 5 from program `proof_depth/3` we can create the instance from the join specification $T_5 :- P, Q$. By unfolding P and Q using P_4 and Q_5, we obtain the clause that is shown below.

```
explain(A,Out,(A :- PA),D) :-
            clause(A,B),
            result(B,Out),
            clause(A,B1),
            D > 0,
            E is D-1,
            proof_depth(B1,PA,E).
```

Applying the meta-folding operation using the join specification (`explain`) the subgoals `result(B,Out)` and `proof_depth(B1,PA,E)` can be combined together to obtain `explain(B,Out,PA,E)`. This is again performed by using the fact that variable `B` and `B1` have the same functionality i.e. both are used for traversing the same data structure. The folded clause is shown as follows:

```
explain(A,Out,(A :- PA),D) :-
            clause(A,B),
            clause(A,B),
            D > 0,
            E is D-1,
            explain(B,Out,PA,E).
```

Applying the *merge* operation, the clause is transformed into the following clause:

```
explain(A,Out,(A :- PA),D) :-
            clause(A,B),
            D > 0,
            E is D-1,
            explain(B,Out,PA,E).
```

Next, we have the mutant clause P_5 in our example. The way that we deal with this clause is the same as described above. By performing unfolding of P with respect to P_5, P_5 unifies with the substitution $\theta_P = \{$`A /Goal ,no/Out`$\}$. Then, by replacing $P_{5,body} =$ `\+ processed_above(A)` in T_6 we get the following clause:

```
explain(A,no,P,D) :-
            \+ processed(A).
```

This clause can be redefined by the user. Assuming that the user wants to redefine the proposed combined clause, a plausible redefinition for the clause could be the clause shown as follows:

```
explain(A,no,A,_D) :-
            \+ processed(A).
```

This redefinition is required if we want the program to return the failing goal as a "proof" tree for the clause that returns a no when a computation fails.

We now have a mutant clause P_6 in our example. The way in which we deal with this clause is the same as was described above: to create an instance of the join specification. By doing unfolding of P with respect to P_6, P_6 unifies with the substitution $\theta_P = \{$`yes/Result1 ,Goals/Goal,Out/Result`$\}$. Then by replacing $P_{6,body} =$ `result(Goals,Out)` in $T_7 =$ `explain_conj(no,Goals,no,P,D)` we have following clause:

```
explain_conj(yes,Goals,Out,P,D) :-
            result(Goals,Out),
            proof_depth(Goals,P,D).
```

After applying the folding operation using the join specification (`explain`) the subgoals `result(Goals,Out)` and `proof_depth(Goals,P,D)` can be combined together to get `explain(Goals,Out,P,D)`. The folded clause is shown as follows:

```
explain_conj(yes,Goals,Out,P,D) :-
        explain(Goals,Out,P,D).
```

This clause is the proposed combined clause but it can be redefined by the user.

Next, we have a mutant clause P_7 in our working example and as before we create an instance of the join specification T_8 :- P, Q. By performing unfolding of P with respect to P_7, P_7 unifies with the substitution θ_P = {`no/Result1,Goals/Goal,no/Result`}. Then by replacing $P_{7,body}$ = `true` in T_7 = `explain_conj(no,Goals,no,P,D)` we have the following clause:

```
explain_conj(no,_Goals,no,P,_D) :-
        true.
```

This proposed combined clause can be redefined by the user. A plausible redefinition for the clause is shown as follows:

```
explain_conj(no,_Goals,no,unsearched,_D).
```

The atom `unsearched` is used as the proof tree for the conjunction of goals that following a failing goal or an aborted goal due an overflow in depth is reached.

The final combined program is as follows:

```
T1 : explain(true,yes,true,_D).
T2 : explain(Goal,overflow,overflow,0).
T3 : explain((A,B),Out,(PA,PB),D) :-
          explain(A,Out1,PA,D),
          explain_conj(Out1,B,Out,PB,D).
T4 : explain(A,yes,sys,_D) :-
          sys(A),
          call(A).
T5 : explain(A,Out,(A :- PA),D) :-
          clause(A,B),
          D > 0,
          E is D-1,
          explain(B,Out,PA,E).
T6 : explain(A,no,A,_D) :-
          \+ processed(A).
T7 : explain_conj(yes,Goals,Out,P,D) :-
          explain(Goals,Out,P,D).
T8 : explain_conj(no,_Goals,no,unsearched,_D).
```

This program **explain/4** returns `overflow` as the result when a computation terminates by depth cut-off, and the failing goal itself as proof tree for the clause that returns a **no** when a computation fails. The meta-interpreter **explain/4** constructed above could be used as a component of a tracer. The combination process can be applied several times allowing construction of more complex programs.

The explain/4 meta-interpreter gives a better answer than the answer that can be obtained by combining the pair of programs as a standard conjunction in Prolog. Let consider us the query explain(son(X,haran),Out,Proof,1) using the following rules.

```
son(X,Y) :-
          ((father(Y,X)),
          (male(X)).

father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).
male(isaac).

male(lot).
female(milcah).
female(yiscah).
```

The answer to the query gives:

	Mutant (M_8)									
P_1/P_2	t	st	s	ctr	meta	mut-t	mut-st	mut-s	mut-ctr	mut-meta
t						α				
st							α			
s								α		
ctr									α	
meta										α
mut-t	α					α				
mut-st		α					α			
mut-s			α					α		
mut-ctr				α					α	
mut-meta					α					α

Table 7.1: Table for the Mutant Method

`Out=overflow` and `Proof =(son(X,haran) :- overflow)`

which is correct because there is not a proof at depth 1.

If we run the pair of programs as a conjunction in Prolog:

`result(son(X,haran),Out), proof_depth(son(X,haran),Proof,1)` then `Out=yes`

which wrongly claims a proof and `Proof=(son(lot,haran):-overflow)`.

In summary, the combination of clauses which do not have a corresponding clause is performed by unfolding just one of the operands in the join specification because the program does not have a corresponding clause with the other program. This is safe because we are restricted to combining programs which are mutants belonging to the same class and for which new clauses must not alter (only add to) the original flow of control. This means the similarity between program and skeleton remains unchanged.

The restriction for the mutant method is as follows:

It requires user interaction because the combined mutant clause (which does not have a corresponding clause in the other program) needs to be approved by the user.

The table 7.1 shows the performance of the mutant method. (See section 6.3 for an explanation of the term α).

7.2 Combining Programs in Traverse and Short_Traverse Classes

Searching for another class of programs with different control flow that could be combined, we found that programs built using the *traverse skeleton* and programs built using the *short_traverse skeleton* can be combined using the method described as procedural join in section 6.7. This was an interesting result in the composition process analysis because the class of programs that can be built using *traverse* and *short_traverse* skeleton is quite large. Examples of some programs which can be constructed using the traverse skeleton are append/3, length/2, merge/3, or sum/2, etc. Examples of some programs which can be developed using short_traverse are find-nth-element, delete-nth-element, etc. These predicates are defined in Appendix B.

Programs constructed using the traverse and the short_traverse skeletons can be combined using procedural join because the only difference between the skeletons *traverse* and *short_traverse* is the base case. The base case for the traverse skeleton ensures that the entire list will always be processed, while short_traverse will either traverse the entire list or stop when a condition has been met.

Programs belonging to these classes cannot be combined in a single recursive program because the combined program might call one of the original programs in some of its clauses. This means that a full self-contained recursive program might not be generated.

Example

The program `split/3` is built using the *traverse* skeleton and the program `del/3` is constructed using *short_traverse*. The program `split/3` divides a list into two lists one containing the positive numbers, and the other the negative numbers. The program `del/3` removes an element X from a list and gives back the remainder of the list. The definition of these two programs is shown as follows:

```
split([],[],[]).
split([A|B],[A|C],D) :-
            A > 0,
            split(B,C,D).
split([A|B],C,[A|D]) :-
            A =< 0,
            split(B,C,D).

del(X,[],[]).
del(X,[X|T],T).
del(X,[_|T],R) :-
            del(X,T,R).
```

This example corresponds to the (t,st) entry in the table shown on page 118.

The combined program `split_del/5` which is obtained by using the procedural join method is shown below. The combination process was performed using the following join specification:

```
split_del(L1,L2,L3,L4,Elem) ⇐ split(L1,L2,L3), del(Elem,L1,L4).
```

```
split_del([],[],[],[],A).
split_del([A|B],[A|C],D,B,A) :-
            A > 0,
            split(B,C,D).
split_del([A|B],[A|C],D,E,F) :-
            A > 0,
            split_del(B,C,D,E,F).
split_del([A|B],C,[A|D],B,A) :-
            A =< 0,
            split(B,C,D).
split_del([A|B],C,[A|D],E,F) :-
            A =< 0,
            split_del(B,C,D,E,F).
```

The main properties in this combined program is that it generates all the set of solutions hence is generated using the procedural join. The new type for this combined program is traverse type.

7.3 Combining Programs in Traverse and Search classes

In this section we combine programs derived from the *traverse* skeleton with those derived from the *search* skeleton. These are two different flows of control. However, the programs created from these skeletons can be combined using the procedural join method. The traverse skeleton ensures that the entire list will always be processed whilst the control flow for the search skeleton will continue processing the list until a condition for terminating search has been found. The traverse and search skeleton were defined in Appendix C. As in the previous section (7.2) programs belonging to these classes cannot be combined in a single recursive program. The combined program might call one of the initial programs to be combined.

For instance, consider the problem of combining `split/3` (defined in the previous section) and `member/2` using the join specification defined below. This example corresponds to the (t,s) entry in the table shown on page 118.

```
split_mem(L1,L2,L3,Elem) ⇐
                    split(L1,L2,L3),
                    member(Elem,L1).
```

The program member checks if an element X is on a list.

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```

The combined program (generated using procedural join), which divides a list into two lists (one containing the positive numbers and the other the negative numbers) and checks if the element in the fourth argument is on the initial list, is defined below.

```
split_mem([A|B],[A|C],D,A) :-
            A>0,
            split(B,C,D).
split_mem([A|B],[A|C],D,E) :-
            A>0,
            split_mem(B,C,D,E).
split_mem([A|B],C,[A|D],A) :-
            A<0,
            split(B,C,D).
split_mem([A|B],C,[A|D],E) :-
            A<0,
            split_mem(B,C,D,E).
```

The combined program `split_mem/3` generates all the solutions that can be generated using a join specification written as a Prolog program. The type for the combined program is search and this combined program uses the definition of the initial program **`split/3`**.

7.4 Combining Programs in Short_Traverse and Search Classes

A similar problem to that of Section 7.3 arises when we want to combine programs in the class *short_traverse* and class *search*.

For instance, consider the program `translate/4` which was built using the short_traverse skeleton. The program `translate/4` translates a phrase from English (e) to Spanish (s) or from English to Italian (i). Each phrase in the List of phrases is formed by two elements f(L,P) where L is the language in our case English (e) and P the phrase expressed as list of words that are forming the phrase. The first argument is the phrase to be found in the list, the second is a list of several phrases, the third argument is the new language and the fourth is the variable in which the translation of the required phrase is returned. This program translate/4 is used as follows: by asking this query `translate(f(e,[the,car]),X,s,Tra)` and having instantiated `X=[f(e,[the,car]),f(e,[the,plane])]`, the program should translate the phrase to Spanish (s), returning `Tra = [el,coche]`.

Note that this example corresponds to the (st,s) entry in the table shown on page 118.

```
translate(X,[],Lan,[]).
translate(X,[X|Xs],Lan,Tra) :-
            aux_tra(X,Lan,Tra).
translate(X,[Y|Ys],Lan,Tra) :-
            X \= Y,
            translate(X,Ys,Lan,Tra).

aux_tra(f(L,P),Lan,Tra) :-
            aux_tra2(P,Lan,Tra).
aux_tra2([X|Xs],LanY,[Y|Ys]) :-
            dict(X,LanY,Y),
            aux_tra2(Xs,LanY,Ys).
aux_tra2([],LanY,[]).

dict(the,s,el).
dict(car,s,coche).
dict(plane,s,avion).
dict(dog,s,perro).
dict(the,i,il).
dict(car,i,automobile).
dict(plane,i,aereo).
dict(dog,i,cane).
```

The second program is formed by the predicate `del/3`, which was constructed using the search skeleton. The predicate `del/3` removes an element from a list and returns the remainder of the list when the element is found.

```
del(X,[X|T],T).
del(X,[_Y|T],R) :- del(X,T,R).
```

The join specification for combining the two programs defined above is shown below.

```
tra_del(Elem,List,La,Tra,List1) ⇐
            translate(Elem,List,La,Tra),
            del(Elem,List,List1).
```

The program `tra_del/5` finds a phrase in a list, gives its translation to an specified language, deletes that phrase from a list of phrases and returns the reminder of the list.

A query for this program (`tra_del/5`) can be

```
tra_del(f(e,[the,car]),[f(e,[the,car]),f(e,[the,plane])],s,Tran,L1).
```

By asking this query the program should translate the phrase `[the,car]` to Spanish, returning `Tra=[el,coche]` and `L1=[f(e,[the,plane])]`

The two programs `translate/4` and `del/3` can be combined by using the procedural join method. The combined program using this method is as follows:

```
tra_del(A,[A|B],C,D,B) :-
            aux_tra(A,C,D).
tra_del(A,[A|B],C,D,E) :-
            aux_tra(A,C,D),
            del(A,B,E).
tra_del(A,[A|B],C,D,B) :-
            A \= A,
            translate(A,B,C,D).
tra_del(A,[B|C],D,E,F) :-
            A \= B,
            tra_del(A,C,D,E,F).
```

This resulting combined program uses the `translate/4` definition. So it is not a full self-contained recursive program. The type for this combined program is search and it generates all the solutions because it was generated performing the full cartesian product of possibles combinations of pairs of clauses.

7.5 Summary Table

A summary table is shown below (see table 7.2). This table presents the set of methods which can be applied to a pair of programs under different conditions (ϕ_i) obtaining different evaluation of the computational behaviour (α,β,γ). The notation used in the summary table is as follows:

	Method
M_1	synchronisation
M_2	join 1-1
M_3	procedural join
M_4	meta-composition
M_5	DS
M_6	particular
M_7	general
M_8	mutant

The conditions ϕ_i are defined in previous sections.

	Methods							
(P_1,P_2)	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
(t,t)	γ	ϕ_1, α	ϕ_2, α	ϕ_1, α	ϕ_3, α	ϕ_4, α	ϕ_5, α	
(t,st)	γ		ϕ_2, β		ϕ_3, β	ϕ_4, β	ϕ_5, β	
(t,s)	γ		ϕ_2, β		ϕ_3, β	ϕ_4, β	ϕ_5, β	
(st,st)	γ	ϕ_1, α	ϕ_2, α	ϕ_1, α	ϕ_3, α	ϕ_4, α	ϕ_5, α	
(st,s)	γ		ϕ_2, β		ϕ_3, β	ϕ_4, β	ϕ_5, β	
(s,s)	γ	ϕ_1, α	ϕ_2, α	ϕ_1, α	ϕ_3, α	ϕ_4, α	ϕ_5, α	
(ctr,ctr)	γ	ϕ_1, β	$\phi_2, \beta,$	α				
(meta,meta)	γ	ϕ_1, β	ϕ_2, β	α				
(mut-t,mut-t)	γ							α
(mut-t,t)	γ							α
(mut-st,muta-st)	γ							α
(mut-st,st)	γ							α
(mut-s,muta-s)	γ							α
(mut-s,s)	γ							α
(mut-ctr,mut-ctr)	γ							α
(mut-ctr,ctr)	γ							α
(mut-meta,mut-meta)	γ							α
(mut-meta,meta)	γ							α

Table 7.2: Summary Table

The condition ϕ_1 is defined on page 110.

The condition ϕ_2 is defined on page 118.

The condition ϕ_3 is defined on page 141.

The condition ϕ_4 is defined on page 153.

The condition ϕ_5 is defined on page 157.

Also the definition of each component in each pair of programs was defined on page 98.

In this summary table (table 7.2) the rows are the pairs of programs and the columns are the methods. (See section 6.3 for an explanation of the terms α, β and γ).

The Synchronization method (M1) as we can see works for each pair defined in our table but is very inefficient with a estimation of efficiency γ.

Join 1-1 (M2) is efficient for pairs (t,t), (st,st) and (s,s) but not for programs constructed using the meta-interpreter or counter skeleton.

Procedural join (M3) is efficient for the pairs (t,t), (st,st) and (s,s) but less efficient (estimation of efficiency β) for the pairs (t,st), (t,s) and (st,s). Furthermore, we still get an estimation of efficiency β for programs created using the meta-interpreter or counter skeletons.

Meta-composition (M4) works efficiently for pairs (t,t), (st,st) and (s,s) and for programs constructed using the meta-interpreters or counter skeleton.

The DS method(M5) combines efficiently pairs (t,t), (st,st) and (s,s) under restriction ϕ_3. On the other hand the combination of pairs (t,st), (t,s) and (st,s) is less efficient so they get β as a estimation of efficiency. The reason for this β is that the combined program still uses the initial programs that were to be combined.

Particular (M6) combines efficiently pairs (t,t), (st,st) and (s,s) under restriction ϕ_4. However the combination of pairs (t,st), (t,s) and (st,s) is less efficient (they get β in the estimation of efficiency). Again, the reason for obtaining β is the dependence on the initial programs.

The General (M7) combines efficiently pairs (t,t), (st,st) and (s,s) under restriction ϕ_5. However the combination of pairs (t,st), (t,s) and (st,s) is less efficient (estimation of efficiency β). The β efficiency is again because of the need to keep the initial programs.

The Mutant (M8) works efficiently for all pairs of mutant programs.

7.6 Comparison of the Program History Approach with the Alternatives

The transformation system developed by Burstall and Darlington (1977) requires knowledge about program transformation. User interaction is required to direct the transformation of the program. This guidance is at a very low level; the user needs to define which

equations need to be unfolded or folded. This approach relies largely on the participation of the user and its efficiency depends on the decisions the user makes. It is based in the standard transformation unfold/fold and arithmetic laws. In our approach we rely less on user interaction by using information about the flow of control and programming practices (techniques) during the transformation the programs. Secondly our approach does not require user interaction at key stages of the transformation for instance, in choosing which clause needs to be unfolded or in deciding which arithmetic laws can be applied.

The main difference between Fuchs's approach [Fuchs & Fromherz 91], based on transformation schemata, and our approach based on the program history, is that in Fuchs' approach the user does not have to deal with the problems of the unfold/fold rules but he needs to choose the output schema for the combined program. On the other hand, in our approach the user does not need to worry about the form of the combined program, but programs are required to be built in a techniques editor.

7.7 Conclusions

In this chapter and the two previous chapters, we describe our resulting set of methods, the corresponding program classification system and how these elements can be put together into an almost fully automatic program combination system. Our set of methods is not complete but it does cover a wide range of combinations of programs with the same or slightly different flow of control.

8

System Description

This chapter concentrates on the more practical aspects of the system, showing how our system can be used. The interface to our composition system is based on menus from which the user can decide to use our system in non-automatic or automatic mode to select the combining method. The first mode requires that the user decides which method can be applied in the combination of a pair of programs. Also, the user needs to learn a non-standard classification by navigating through the classification hierarchy before combining the programs. On the other hand the automatic-mode the system will decide the most suitable method (performing the classification automatically based on the program histories). We provide our composition system with these two options because we believe that if the user wants to experiment with how the combination is performed the non-automatic mode can be useful but, if not, the automatic mode can be selected.

8.1 Overview

The implemented composition system is a prototype designed with the purpose of showing our approach to the combination problem. The system will be described with a sequence of screens obtained during the construction of an example program. The system is provided with a library of transformation rules, a knowledge base of skeletons, and a knowledge base of techniques as described in Appendix C and Appendix D. The knowledge base of

skeletons and techniques is not complete and might be extended in order to cover more skeletons and techniques. Currently they contain the most representative skeletons and techniques. The composition system was written in Arity Prolog at Edinburgh University, and allows the composition of programs automatically using the following methods: procedural join, join 1-1, the meta-composition method, the DS method, the particular, the general method, and the method for combining programs in class traverse-short_traverse, traverse-search and short_traverse-search. The mutant method is also implemented, but requires user help as it cannot be completely automated, as was explained in Chapter 7. Before describing the structure of the system and giving an example of it at work, we highlight some of the important features of the system.

- Our system allows the user to build complex programs by combining them several times. This tend to confirm our expectations about construction of more complex Prolog programs. The nature of our approach is basically incremental, so a more complex program is obtained by combining it several times.

- The user interaction is reduced enormously and simplified by using the program history because important decisions are taken automatically rather than asking the user. The sequence of transformation is chosen by the system automatically by using the information recorded in the program history concerning the type of program. User interaction is required only for supplying the name and arity of the top level predicate and the definition of the join specification, plus confirmation of combined mutant clauses in the mutant method.

- Our composition system does not require users to possess any specialist knowledge about program transformation, unlike former transformation systems.

- Our classification of Prolog programs can be extended in order to allow combination of more classes of program. This can be achieved by extending the knowledge base of skeletons and techniques defined in Appendix C and Appendix D.

8.2 Non Automatic Mode

Firstly our composition system will display a menu in which the user can choose to use our system in an automatic or manual way. If the user takes the first option the user needs to decide which method will be used by analysing the characteristics of the pair of programs to be combined and by considering our classification of programs (defined in Chapter 5). Otherwise the combined method will be selected directly by the system.

COMPOSITION SYSTEM

1. Non-automatic selection of the composition method
2. Automatic selection of the composition method
3. exit

Select an option from the menu: 1.

If the user chooses the non-automatic selection mode the composition system will display a menu which is shown below. This menu represents the top level of our classification of programs that can be combined using the composition system.

COMPOSITION SYSTEM

1. compose programs with same control flow (enhancements)
2. compose programs with different control flow
3. exit

Select an option from the menu: 1.

If we choose option 1 the system begins by offering two options for combining programs: using a join specification formed by two operands or a join specification formed by two operands plus extra subgoals which are computing values using the outputs of each operand. In the latter join specification shown in the following menu (see Page 188), extra computations are allowed using local variables (variables that do not appear in the

definition of the join specification). Further explanation is given in Chapter 6.

COMPOSITION SYSTEM

1. compose with JS: T(IP,IQ,OP,OQ) $\Leftarrow$ P(IP,OP), Q(IQ,OQ)
2. compose with JS: T(IP,IQ,OT) $\Leftarrow$ P(IP,OP), Q(IQ,OQ), F(OP,OQ,OT)
3. exit

Select an option from the menu: 1.

In the menu above IP, IQ are input vectors of distinct variables. OP, OQ, OT are output vectors of distinct variables. F is a predicate that produces as output the vector OT using the obtained values in procedure P and in procedure Q.

The user chooses option 1 and the system then offers the following set of methods: the 'synchronization' method, join 1-1, procedural join, the meta-composition method, and a method called DS for combining programs operating with different data structures.

In the menu, 0-0 tests means that none of the pair of programs to be combined have a test.

COMPOSITION SYSTEM

1. SYNCHRONIZATION (JS the input for Q is the result of P ie. T(IP,IQ,OP,OQ) $\Leftarrow$ P(IP,OP), Q(OP,OQ))
2. JOIN 1-1 (same number of clauses & 0-0 or both have the same test & same order of clauses)
3. PROCEDURAL_JOIN (same number of clauses and both programs have different tests & no ordering of clauses is required)
4. META-COMPOSITION METHOD (same number of clauses & local var. in P or Q & 0-0 or both programs have the same test & same order of clauses)
5. programs operating over data structures in a different domain
6. exit

Select an option from the menu: 3.

If the user chooses the non automatic mode, he has the following responsibilities: to select the correct method according to the features of the two programs which will be combined, to provide a user specification or redefinition of the user specification and finally to accept or redefine combined clauses in some methods. For example we can take the program sum_odds(List,SumOdds) which computes the sum of the odd numbers in a list and the program sum_fives(List,SumFives) which computes the sum of all instances of the number 5 that appear in a list. The program `sum_odds/2` and `sum_fives/2` is shown as follows:

```
sum_odds([],0).
sum_odds([X|R],SO) :-
            odd(X),
            sum_odds(R,S1),
            SO is S1 + X.
sum_odds([X|R],SO) :-
            \+ odd(X),
            sum_odds(R,SO).
```

```
sum_fives([],0).
sum_fives([X|R],SF) :-
            five(X),
            sum_fives(R,S2),
            SF is S2 + X.
sum_fives([X|R],SF) :-
            \+ five(X),
            sum_fives(R,SF).
```

The main features of the two programs are: both programs were constructed by using the same *traverse* skeleton consisting of three clauses, both have the same number of clauses and each program has a different test in each recursive case. In the program `sum_odds/2` the test is to check whether or not each element of the list is an odd number, whilst in program `sum_fives/2` the test is whether or not each element is the number five. This information can be obtained from the program history.

The previous characteristics determine a combination method which can be used by the system; for instance, in the combination of the programs `sum_odds/2` and `sum_fives/2`. Figure 8.1 shows a set of conditions which are considerated in the selection of the combining method.

The meta-composition or the join 1-1 method cannot be used because these methods are suitable only when both programs have the same *test*.

Therefore at this selection stage in the composition process the user must choose the procedural join method otherwise the resulting combined program might be wrong. After this the system will ask for the following sequence of questions:

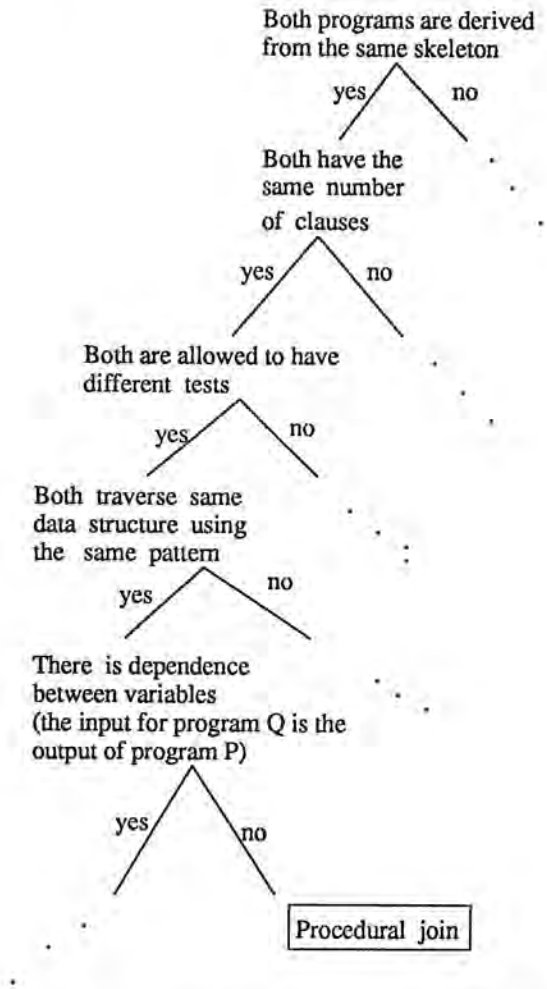

Figure 8.1: A Decision Tree for Combining sum_odds/2 and sum_fives/2

COMPOSITION SYSTEM

- The composition of two programs requires the provision of the names of the top level predicates (i.e. $predicate_1$ and $predicate_2$).
- first predicate: `sum_odds`.
- arity: 2.
- second predicate: `sum_fives`.
- arity: 2.
- File where the programs are defined: `arch_1`.
- File name for the joined program: `result_file`.

This information is used for searching if the system has a pre-stored join specification which involves the top level predicates. So, the system will either provide a join specification (in the case that the system has one pre-stored) or the user may supply one interactively. For our example the system offers the following join specification:

COMPOSITION SYSTEM

- The system has this specification for combining the two predicates provided:

```
sum_of(L,SumOdds,SumFives) ⇐
            sum_odds(L,SumOdds),
            sum_fives(L,SumFives).
```

- Do you want to redefine the join specification?
- answer: no.
- Note that your programs will be joined using the following user specification:

```
sum_of(L,SumOdds,SumFives) ⇐
            sum_odds(L,SumOdds),
            sum_fives(L,SumFives).
```

- answer: ok.

If the user accepts the join specification suggested by the system then the system will ask for the user confirmation. If the user answers *ok* then the programs will be combined using the join specification displayed on the screen. Otherwise the user can define his own join specification interactively.

The combined program `sum_of/3` was built using the procedural join method that we have described earlier in Chapter 6. The resulting combined program is shown as follows:

COMPOSITION SYSTEM

```
sum_of([],0,0).
sum_of([X|R],SO,SF) :-
            odd(X),
            five(X),
            sum_of(R,S1,S2),
            SO is S1 + X,
            SF is S2 + X.
sum_of([X|R],SO,SF) :-
            odd(X),
            \+ five(X),
            sum_of(R,S1,SF),
            SO is S1 + X.
sum_of([X|R],SO,SF) :-
            \+ odd(X),
            five(X),
            sum_of(R,SO,S2),
            SF is S2 + X.
sum_of([X|R],SO,SF) :-
            \+ odd(X),
            \+ five(X),
            sum_of(R,SO,SF).
```

Another example of the use of the composition system is presented by choosing the second option of the menu shown below.

COMPOSITION SYSTEM

1. compose with JS: T(IP,IQ,OP,OQ) $\Leftarrow$ P(IP,OQ), Q(IP,OQ)
2. compose with JS: T(IP,IQ,OT) $\Leftarrow$ P(IP,OQ), Q(IQ,OQ),F(OP,OQ,OT)
3. exit

Select an option from the menu: 2.

On choosing option 2, the system will present a choice between the general and particular methods. The particular method requires that the arithmetic operators in the subgoals of program P Q and F are the same.

COMPOSITION SYSTEM

1. PARTICULAR (arithmetic operators in program P, Q and F are the same)
2. GENERAL
3. exit

Select an option from the menu: 1.

In order to show how the particular method works we select the first option and reconsider the definition of the program `len/2` which computes the length of a list. The definition of `len/2` is shown as follows:

```
len([],0).
len([H|T],Len) :-
            len(T,Lenx),
            Len is Lenx + 1.
```

If at this selection stage of the composition process, the user chooses the particular method by analysing the conditions shown in Figure 8.2.

Then the system will ask the user for the standard information about the top level predicates. This information is obtained in the following order.

COMPOSITION SYSTEM

- first predicate: `len`
- arity: 2
- second predicate: `len`
- arity: 2
- File where the programs are defined: `arch_2`
- File name for the joined program: `result_file_2`

Secondly the definition of the join specification is defined interactively as follows:

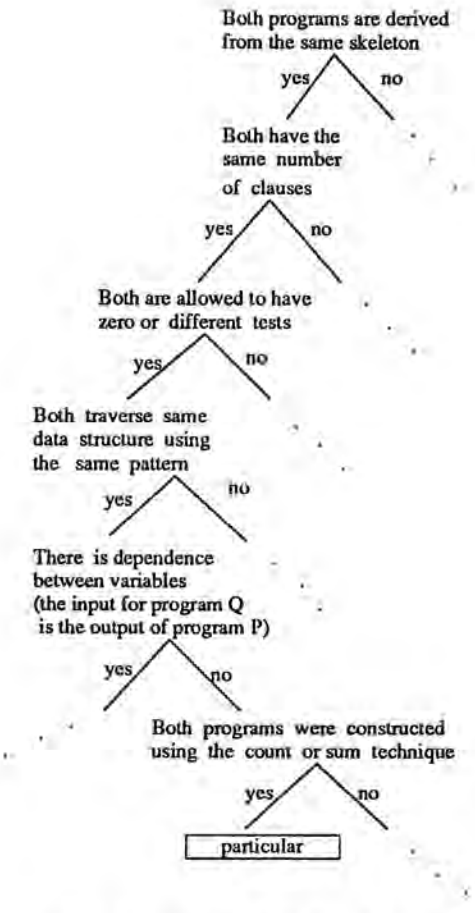

Figure 8.2: A Decision Tree for Combining len_two/3

```
len_two(L1,L2,Len) ⇐
            len(L1,Len1),
            len(L2,Len2),
            Len is Len1 + Len2.
```

The *particular* method can be defined as follows: unfolding predicate P and predicate Q and then rewriting the unfolded clause using transformation schemas with restrictions as defined in Chapter 6. The resulting combined program `len_two/3` is shown on Page 151.

We then return to the first menu of the composition system as shown below.

COMPOSITION SYSTEM

1. compose programs with same control flow (enhancements)
2. compose programs with different control flow
3. exit

Select an option from the menu: 2.

If we choose option 2 the system begins by offering a set of methods for combining programs with different flow of control.

COMPOSITION SYSTEM

1. MUTANTS (meta-interpreters)
2. trav-strav method (trav o short_trav)
3. trav-search method (trav o search)
4. strav-search method (short_trav o search)
5. exit

Select an option from the menu:1.

To show how option 1 works we take as an example the meta-interpreter `result/2` and the meta-interpreter `proof_depth/3` developed in Chapter 7. A more complex program can be produced by combining the meta-interpreter `proof_depth/3` and the meta-interpreter `result/2` using the following join specification.

```
explain(Goal,Out,P,D) ⇐ result(Goal,Out), proof_depth(Goal,P,D).
```

The user should choose the mutant method by considering the conditions given in Figure 8.3. The main characteristic of the mutant method is that it is semi-automatic. It will offer a combined clause for each mutant clause appearing in each program to be combined. This can be accepted or rejected by the user. In the case that the user rejects the option, the redefinition must be performed by the user.

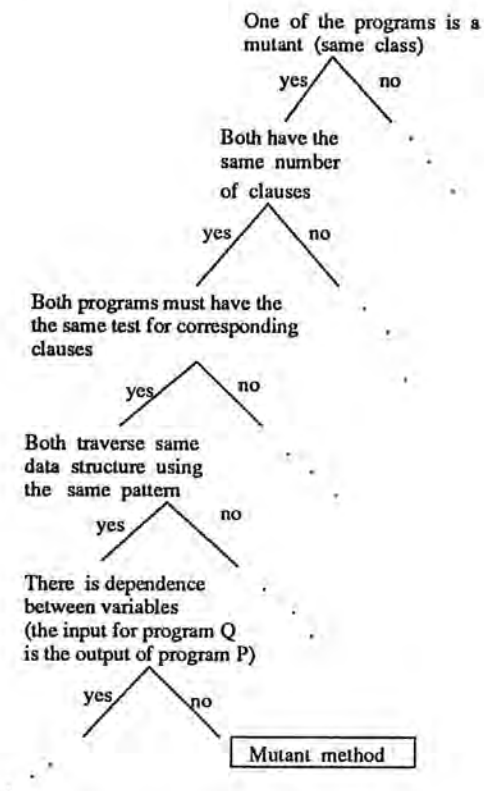

Figure 8.3: A Decision Tree for Combining result/2 and proof_depth/3

COMPOSITION SYSTEM

```
1 : explain(true,yes,true,D).
2 : explain(Goal,Out,overflow,0).
3 : explain((A,B),Out,(PA,PB),D) :-
          explain(A,Out1,PA,D),
          explain_conj(Out1,B,Out,PB,D).
4 : explain(A,yes,sys,D) :-
          sys(A),
          call(A).
5 : explain(A,Out,(A :- PA),D) :-
          clause(A,B),
          D > 0,
          D is D-1,
          explain(B,Out,PB,D1).
6 : explain(A,no,P,D) :-
          \+ processed(A).
```

The redefinition of the mutant clauses is offered to the user in the following dialogue:

COMPOSITION SYSTEM

- Do you want to redefine the mutant clause number 6 ?
- **yes**
- Give me the redefined clause:

```
explain(A,no,A,D) :-
                \+ processed(A).
```

The resulting combined program after redefining the mutant clause is as follows:

COMPOSITION SYSTEM

```
1 : explain(true,yes,true,D).
2 : explain(Goal,overflow,overflow,0).
3 : explain((A,B),Out,(PA,PB),D) :-
            explain(A,Out1,PA,D),
            explain_conj(Out1,B,Out,PB,D).
4 : explain(A,yes,sys,D) :-
            sys(A),
            call(A).
5 : explain(A,Out,(A :- PA),D) :-
            clause(A,B),
            D > 0,
            D is D-1,
            explain(B,Out,PB,D1).
6 : explain(A,no,A,D) :-
            \+ processed(A).
7 : explain_conj(yes,Goals,Out,P,D) :-
            explain(Goals,Out,P,D).
8 : explain_conj(no,Goals,no,unsearched,D).
```

8.3 Automatic Mode

The automatic mode uses the program history in order to decide which method can be applied at the selection stage. Consider again the pair of programs **sum_odds/2** and

sum_fives/2. Assuming that these two programs were constructed using a techniques editor based on ideas of skeletons and techniques such as the described in [Robertson 91]. Their program history is shown below.

```
his_prog(sum_odds,type_traverse,2,traverse,3,2,count_technique,
          [...,[2,(trav(H),sum_odds(R,S1))],[3,(trav(H),sum_odds(R,S0))]],nil).
his_prog(sum_fives,type_traverse,2,traverse,3,2,count_technique,
          [...,[2,(trav(H),sum_fives(R,S2))],[3,(trav(H),sum_fives(R,SF))]],nil).
```

By using this information we know that the skeleton used by both programs was the *traverse* skeleton. They have the same number of clauses (ie. three) and two different tests (one for each recursive case). Also the eighth argument in the relation *his_prog/9* states that the construction used the technique *count*. Therefore the system will choose the procedural join method. Further details for combining these programs are described in page 113. As a result, the composition system will return the combined program, and a new relation will be added to the program history. Note that this program history is derived from information that the user answer about the combined program and information recorded for each program to be combined.

```
his_prog(sum_of,type_traverse,3,traverse,5,4,count_technique,
          [...,[2,(trav(H),sum_of(R,S1,S2))],[3,(trav(H),sum_of(R,S1,F))],
          [4,(trav(H),sum_of(R,S0,SF))]],nil).
```

8.4 Characteristics of the Composition System

Our composition system is not only a transformation system. It makes use of program transformation operations and knowledge about the program in order to achieve efficient combined programs. As a second result, user interaction is reduced by utilising knowledge about the program. Finally, it provides a methodology for constructing programs. Our composition system allows the program to be constructed in a modular way and also encourages reuse of software. A more complex program is obtained after combining the program several times.

The main characteristics of our combination system are as follows:

- It does not requires knowledge about program transformation.
- It takes decisions such as the right sequence of transformation by analysing the pair of programs to be combined.
- It takes decisions such as which clause or subgoal needs to be unfolded or folded.
- It chooses arithmetic rules which can be applied in order to get a more optimised combined program.
- It allows the building of complex programs by combining the programs several times.

8.5 Conclusions

User interaction in the transformation system is clearly enhanced by reducing the number of questions the user needs to answer to comparatively easy questions, such as the name of the program, arity and the definition of the join specification. In addition, our system does not requires the user to possess knowledge about program transformation.

Our approach is a dynamic process in which users do not (unless they so desire) need to face any decisions about which transformations are applied to the initial pair of programs. The system automatically chooses the sequence of transformations. However there is an exception in our composition system; our mutant method requires more user interaction. The user interaction is to accept or reject a proposed combined clause which does not have any corresponding clause in the skeleton (in our terminology this clause is called a mutant clause). Such clauses do not inherit properties of the skeleton, so they are treated differently to the other clauses in the program.

9

Properties of the Composition System

We discuss properties which guarantee that the composition process is sound at three different levels. The first group of properties shows characteristics of each of our combining methods such as whether the method preserve equivalence or just gets a subset of answers. The second group of properties guarantees that the type of the combined program is one of the types in our classification (see Chapter 5). This information helps in the selection of each method at different stages in the combination process and also in the definition of the combined program. Finally, in the third group we define properties which help in the definition of the join specification.

9.1 Terminology and notation

$P_1 \subset P_2$ should be read as P_2 *is an extension of* P_1 where P_1 and P_2 are programs consisting of several predicates.

$X \in Vp$ should be read as X is member of the set of variables of program P.

$X \in Gp$ should be read as X is a member of the set of subgoals of program P.

An extension is a special type of enhancement restricted to add computations which do not affect the control flow of the skeleton (i.e. assignments, subgoals used for constructing

a data structure or arithmetic computations). Our definition of extension is given in terms of the subset relation. The relation $\subset$ (used in the following definition) is a special subset operation differing from the standard definition of a subset. $S \subset P$ states that all the variables in S appear in program P and all the subgoals appearing in program S appear in program P and P must contain some additional variable and/or subgoal than S. Formally:

Definition 9.1 $S \subset P \longrightarrow \forall x \forall g\ (x \in V_S \ \textit{and}\ g \in G_S \longrightarrow x \in V_P \ \textit{and}\ g \in G_P)$ $\textit{and}\ \exists x'(x' \in V_p \ \textit{and}\ x' \notin V_s) \ \vee \ \exists g'(g' \in G_p \ \textit{and}\ g' \notin G_s)$.

Each subgoal g appearing in program S can be found in two ways in program P: as an enhancement (i.e. more arguments and/or different name) or the same subgoal as appears in program S (i.e. same name and same number of arguments).

Each program is constructed using techniques which do not change the flow of control i.e. techniques only add new variables or add subgoals which perform extra computations around the flow of control provided by the skeleton without altering its flow of control. These subgoals can be assignments, subgoals which are constructing a data structure, arithmetic computations but they cannot be *tests*. The tests change the flow of control. A test is either: arithmetic comparison for instance, X=C where C is a constant; a system single-argument test (e.g. atom(X), integer(X), etc.); or a trivial test (i.e. true). So after applying a technique only two cases can occur: the first case is when the effect of a technique is just addition of variables through the head of the clause and the recursive calls. The second case is when the technique adds variables in the head of the clause, in the recursive calls plus new subgoals which are added to the body of the clause.

The definition 9.1 of the extension states that the program P is an extension of the program S if the differences between the program S and P are new variables included in the head of the predicate and through the recursive calls plus new subgoals, which do not affect the control flow of the program (i.e. the structure of the skeleton is preserved in the extension).

Example

The following example illustrates the definition of extension given above. In this example the initial skeleton S is called `transition` and the program P is called `trans_eval/2` which is an extension of S.

```
transition([A|As]):-
   update([A|As],Bs),
   transition(Bs).
```

$\Longrightarrow$

```
trans_eval([A|As],Output) :-
        update([A|As],Bs),
        trans_eval(Bs,Output1),
        <eval>(Output1,As,Output).
```

```
transition([]).
```

```
trans_eval([],0).
```

The extension called `trans_eval/2` is obtained by applying a technique *evaluate* which can be either count or sum technique. The structure of the skeleton `transition/1` is preserved into the program `trans_eval/2`. Each variable appearing in `transition/1` appears in `trans_eval/2`. In a similar fashion all the subgoals appearing in `transition/1` appear in `trans_eval/2` in the same way or as enhancements. A possible instantiation of `<eval>` for the sum technique is

```
sum(Sum,[A|As],Sum1),
Sum1 is Sum + A.
```

Note that in our example the head `trans([A|As])` in clause number 1 is transformed as `trav_eval([A|As],Output)` and the body of this clause which is formed by the subgoals: `update([A|As],Bs),transition(Bs)` is transformed in the following subgoals: `update([A|As],Bs),trans_eval(Bs,Output1),<eval>(Output1,As,Output)`

9.2 Method properties

Our composition system is formed by a set of methods which are defined in terms of a sequence of transformation operations. This section presents a discussion of each method showing under what conditions it guarantees the equivalence of programs. The equivalence

of programs is proved for the synchronization, procedural join and DS, using the proofs developed by Tamaki and Sato [Tamaki & Sato 84]. The join 1-1 and meta-composition method do not preserve the equivalence hence these methods enforce the synchronisation of the traversal of the data structure. Therefore only a subset of solutions are obtained. The particular and general method preserve the equivalence (because the full cartesian product is performed). Finally, the mutant method does not preserve the equivalence.

In what follows we assume that the join specification is always correctly defined (i.e. it meets the property defined in Section 9.4).

Also for each of our methods we are assuming that the initial pair are well defined [Bundy *et al.* 91]. This must be guaranteed by the techniques editor.

9.2.1 Synchronization

In this method the programs are combined in the simplest way, the join specification defined as follows:

$$T(\vec{I_P}, \vec{O_P}, \vec{O_Q}) \Leftarrow P(\vec{I_P}, \vec{O_P}), Q(\vec{I_Q}, \vec{O_Q}).$$

this is transformed into a Prolog program directly. Then the combined program is:

$$T(\vec{I_P}, \vec{O_P}, \vec{O_Q}) \ :- \ P(\vec{I_P}, \vec{O_P}), Q(\vec{I_Q}, \vec{O_Q}).$$

Since P and Q are directly connected by this join specification, then, assuming it is well defined, the combined program is also well defined.

9.2.2 Join 1-1

This method enforces the same traversal of the data structure for each clause and so only gives a subset of answers that would be possible in general. The equivalence of the

programs is not preserved but the correctness of the combined program for that intended behaviour of the program might be maintained. Note that the join specification for this method is defined as

$$T(\vec{I_P}, \vec{I_Q}, \vec{O_P}, \vec{O_Q}) \Leftarrow P(\vec{I_P}, \vec{O_P}), Q(\vec{I_Q}, \vec{O_Q}).$$

where this join specification enforces the same traversal of the data structure used by both programs.

However we noticed that, if we restrict the set of programs which can be handled by the join 1-1 to only programs with the characteristics defined below:

- All the clauses in each program are exclusively independent,
- if each pair of corresponding clauses (taken one from each program) has the trivial condition (i.e. true) or the same test.
- The argument for the data structure (first argument) is instantiated (i.e. it is not a variable) in the head of the clause.

then the equivalence is preserved as the program generated combining corresponding clauses is the same as the program generated performing the full cartesian product using normal fold-unfold (see example in page 107).

9.2.3 Procedural join

This method is an extension of the previous method. The equivalence of the combined program is preserved by the fact that in the combination of pairs of clauses the full cartesian product is performed. So, even unwanted solutions are preserved by using this method. It is based entirely on unfolding and folding operations. Therefore the equivalence of programs is preserved as proved by Tamaki and Sato in [Tamaki & Sato 84]. The final remark for this method is that if would be very difficult (if not impossible) to exclude the unwanted solutions from the combined program as we did in join 1-1.

9.2.4 Meta-composition

The *meta-composition* method is based on unfolding and meta-folding operations. The latter is an extension to the standard definition of the folding operation, which uses knowledge about the program's development to control how the fold operation is used. The *meta-folding* operation verifies before folding that the subgoals that are candidates to be folded into a single subgoal are enhancements of the same subgoal in the skeleton. Otherwise any subgoal is folded.

The candidate subgoals, to be folded, in general are the recursive subgoals from each program. These subgoals can have local variables in their arguments. Sometimes these local variables are used for traversing the same data structure. Therefore these variables can potentially be unified. The restriction on whether they can safely be unified is that they must traverse the data structure in the same pattern (constructing or deconstructing the data structure) or appear in exactly the same environment (see Section 6.8). This information can be inferred from the *program history* which states how the data structure is traversed in each program.

The knowledge of which variables can be unified, is provided in the *program history* as relations (which were defined in Chapter 5). These allow us to infer the name of the local variables which can be safely unified together because they have the same functionality (they operate over the same data structure).

The equivalence between the initial pair of programs (if the join specification is regarded as Prolog program) and the combined program using the *meta-composition* method is not ensured. This method enforces the same traversal of the data structure as the join 1-1 method. If we restrict the set of programs which can be handled by the meta-composition method to the programs with the characteristics defined in section 9.2.2. then the equivalence is preserved, as the program generated combining the corresponding clauses is the same as the program generated performing the full cartesian product (see example on page 107). In this case the extra condition which needs to be guaranteed is that the *pro-*

gram history is correct. This depends on the correctness of the techniques used in the editor which produced the programs and on the correctness of the technique applications in the editor. For the purposes of this thesis we take this as given.

9.2.5 DS

This method uses the procedural join method. The equivalence of the combined program is preserved by the fact that in the combination of pairs of clauses the full cartesian product is performed. So, even unwanted solutions are preserved by using this method. It is based entirely on unfolding and folding operations. Therefore the equivalence of programs is preserved as proved by Tamaki and Sato in [Tamaki & Sato 84].

9.2.6 Particular

The *particular* method can be applied when both of the programs to be combined were constructed by applying the technique *sum* or *count*. This method operates by performing unfold, arithmetic laws and the folding operation. The laws are those which are valid for a specific arithmetic domain such as integer or real numbers, for instance, commutativity and associativity in the real or integer domain.

This method works by performing rewrite rules in order to increase the possibility of folding subgoals. The arithmetic rewrites are always correct because these are based on properties on real/integer numbers (see page 150 for a working example). Also, the particular method preserves the equivalence of programs. It performs the combination of pairs of clauses considering all the possible combinations. Therefore the equivalence is guaranteed.

9.2.7 General

The *general* method is an automatic way of building programs which have a user specification formed by two operands plus extra subgoals, but without any restriction on the form of extra subgoals (as in the particular method). This method is based only on the

standard unfolding and folding operations. The equivalence between initial programs and the combined program is thus guaranteed by the fact that the combination of the clauses is not restricted as in the join 1-1 method. This method uses the following join specification:

$$T(\vec{I_P}, \vec{I_Q}, \vec{O_T}) \Leftarrow P(\vec{I_P}, \vec{O_P}), Q(\vec{I_Q}, \vec{O_Q}), F(\vec{O_P}, \vec{O_Q}, \vec{O_T}).$$

where the restriction on F in the Particular method does not apply (see Section 6.10). This method can be used when F is a subgoal performing a computation in a different domain other than the computations in program P and Q.

This method works by applying a sequence of unfolding/folding operations. The equivalence of the combined program after applying this method is guaranteed by using the proofs defined by Tamaki and Sato [Tamaki & Sato 84].

9.2.8 Mutant

The *mutant* method is based on the *meta-composition* method described above. The extension for this method deals with mutant clauses. This is a hybrid method which combines corresponding clauses using the *meta-composition* method and clauses which do not have corresponding clauses by using the *mutant* method.

The problem of extra clauses which do not have a corresponding clause in program P with respect to a second program Q is handled as follows: first create an instance of the join specification T_k and then spread the values of the variables through the instance T_k by unfolding either predicate P or predicate Q. The selection of which predicate is unfolded is determined as follows: if the mutant clause belongs to program P then the unfold operation is applied only in the first operand of the join specification (P), otherwise if the mutant clause belongs to program Q then the second operand (Q) is unfolded and then the fold operation is performed. This process is repeated while there are still mutant clauses in each program. The unfolding operation is applied partially through the join specification and the final mutant combined clause can be accepted or rejected by the user.

This method controls which clauses have a corresponding clause between the pair of programs to be combined by using the program history (which contains a list of non-mutant clauses). Therefore all the clauses which have a corresponding clause can be combined as usual. The second step is how to combine the clauses which do not have a corresponding clause. For these clauses we are unfolding just one of the operands in the join specification because the program does not have a corresponding clause with the other program. This is safe because we are restricted to combining programs which are mutants belonging to the same class and which extend, rather than replace, the original flow of control. This means that the pair of programs to be combined has some structural similarity.

This method enforces the same traversal of the data structure for each clause (which has a corresponding one in the other program). Therefore we are restricting the set of possible answers. But if we only combine programs restricted to the kind of program as is defined in section 9.2.2 the set of answers should be the same as the set of answers produced performing the full cartesian product (see example on page 107).

The correctness of the combined program is also in danger if the user redefines the offered mutant combined clauses. Then the responsibility for the correctness of the combined mutant clauses lies with the user.

9.3 Extension properties

The set of properties defined in this section are useful for programs which are *extensions* of a skeleton (defined in Section 9.1). These programs are obtained by applying techniques to a skeleton.

Lemma 9.1 *If* $P_1 \subset P_2$ *and* $P_2 \subset P_3$ *then* $P_1 \subset P_3$.

Proof:

If $P_1 \subset P_2$ then by definition 9.1

$\forall\, x \forall\, g\ (x \in V_{P_1}\ and\ g \in G_{P_1} \implies x \in V_{P_2}\ and\ g \in G_{P_2})$ and P_2 must contain some additional variable and/or subgoal than P_1 (1)

and

If $P_2 \subset P_3$ then by definition 9.1

$\forall\, x \forall\, g\ (x \in V_{P_2}\ and\ g \in G_{P_2} \implies x \in V_{P_3}\ and\ g \in G_{P_3})$ and P_3 must contain some additional variable and/or subgoal than P_2 (2)

Therefore

$\forall\, x \forall g\ (x \in V_{P_1}\ and\ g \in G_{P_1} \implies x \in V_{P_3}\ and\ g \in G_{P_3})$ and P_3 must contain some additional variable and/or subgoal than P_1.

Therefore by using (1) and (2)

$P_1 \subset P_3$

Theorem 9.1 *The relation $\subset$ is anti-reflexive, anti-symmetric and transitive.*

Proof:

Anti-reflexivity : $P_1 \not\subset P_1$

$P_1 \not\subset P_1$ by using definition 9.1

Anti-symmetry : $P_1 \subset P_2$ then $P_2 \not\subset P_1$

$P_1 \subset P_2$ then by using definition 9.1 we have the case that P_2 must contain new variables or subgoals than P_1. These subgoals do not affect the flow of control of P_2 with respect to P_1 then

$P_2 \not\subset P_1$ because by construction each technique adds new variables or new subgoals to the new program P_2.

therefore $\subset$ is anti-symmetric.

Transitivity: $P_1 \subset P_2$ and $P_2 \subset P_3$ then $P_1 \subset P_3$.

$P_1 \subset P_2$

and

$P_2 \subset P_3$ then

$P_1 \subset P_3$ by using lemma 9.1.

Therefore the relation $\subset$ is anti-reflexive, anti-symmetric and transitive. □

Theorem 9.2 *If P_1 is an extension of skeleton S with type α and P_2 is an extension of skeleton S with type α then $P_1 \circ P_2$ is an extension of S of type α where α is one of types defined in our classification schema.*

The previous theorem states that the composition (denoted by the symbol $\circ$) of two extensions of the same skeleton and with the same type α is another extension of the same skeleton with the same type α. This property is important for performing further composition stages.

Proof:

If $S \subset P_1$ *and* $S \subset P_2$

then by definition 9.1

$\forall x \forall g$ ($x \in V_S$ *and* $g \in G_S \implies x \in V_{P_1}$ *and* $g \in G_{P_1}$) and P_1 must contain new variable and/or subgoals which do not affect the flow of control of P_1 with respect to S

Similarly $\forall x \forall g$ ($x \in V_S$ *and* $g \in G_S \implies x \in V_{P_2}$ *and* $g \in G_{P_2}$) and P_2 must contain new variable and/or subgoals which do not affect the flow of control of P_2 with respect to S

By combining the program P_1 and P_2 which are extensions of the skeleton S we obtain a program which is formed by the extra arguments present in each program and by the extra subgoals also present in both programs. The combined $P_1 \circ P_2$ contains the same flow of control of the skeleton S because none of combining methods modifies the flow of control

of the initial skeleton and P_1 and P_2 are extensions of S (e.g. none of the techniques used in its construction modify their flow of control by definition of our techniques).

Therefore

$S \subset P_1 \circ P_2$

Then

$P_1 \circ P_2$ is an extension of S of type α.

9.4 Composition properties

These properties are useful when the user defines the join specification. The commutative property states that the order of the operands of the join specification is irrelevant. The one exception is our *synchronization* method. In this method the commutativity is not valid because there is a dependence of variables between the two programs to be combined.

Theorem 9.3 *The relation* $\circ$ *is commutative.*

Formally this theorem states:

$P_1 \circ P_2 \equiv P_2 \circ P_1$

Proof:

$P_1 \circ P_2$ then $P_2 \circ P_1$

Let J_1, the join specification, be defined as:

$$P_1\_P_2 :- P_1, P_2$$

$P_1 \circ P_2$ was joined using join specification J_1

If we change the join specification from J_1 to J_2 defined as:

$$P_2\_P_1 \text{ :- } P_2, P_1$$

Then the joined program using the join specification J_2 has the following properties.

- No new variables are introduced by the composed program due to the join specification restriction. The same variables that appear in $P_1 \circ P_2$ also appear in $P_2 \circ P_1$
- Each subgoal appearing in program $P_1 \circ P_2$ is in $P_2 \circ P_1$, but in a different order. $G_{i,j,P_1}(\vec{X}) = G_{i,j,P_2}(\vec{Y})\theta_2$ where $\vec{X}$ and $\vec{Y}$ are the variable vectors.
- The recursive calls appearing in the program $P_1 \circ P_2$ also are in the program $P_2 \circ P_1$ as follows:

  $G_{i,j,P_1 \circ P_2}(\vec{X}) = G_{i,j,P_2 \circ P_1}(\vec{Y})\theta_2$ where $\vec{X}$ and $\vec{Y}$ represent vector of variables.

therefore

$P_1 \circ P_2 \equiv P_2 \circ P_1$

If we assume that $T \longleftarrow P, Q$ is equivalent to $T \longleftarrow Q, P$ (assuming cut-free code), the vector of variables of P_1 and P_2 are independent and that all transformations preserve the meaning of the join specification then join specifications of the form given above are commutative and so are the programs produced from them.

9.5 Conclusions

We have defined three groups of properties which are useful for the composition process. These properties ensure that the composition process is sound. The first group of properties relates to whether the method preserves the equivalence between programs or just

gives a subset of results; the second group gives properties of program extensions (generated by applying techniques to skeletons) and the third group are properties of the join specification.

10

Conclusions and Further Work

In this chapter firstly we summarise our work on the combination problem for Prolog programs. Secondly we describe a set of restrictions in our composition system. Thirdly we show a list of major improvements which can be done as future work and finally the contributions of the thesis.

10.1 Summary

We found that simple information about the key stages in predicate definition could markedly improve the efficiency of program transformation requiring little user interaction. This information which we call a *program history* can be viewed as a collection of meta-information which is shown (through this thesis) to be very useful for the combination of programs. This program history describes the program in a way an expert programmer might describe it (in terms of flows of control and programming practices). The program history might be obtained from a techniques editor. We have analysed two kinds of editors: based on program schemata and skeleton and techniques notions and found that the sort most suitable (for our purpose) is an editor based on skeletons and techniques as proposed by Kirschenbaum et al. [Kirschenbaum *et al.* 89].

We produced a classification schema which groups Prolog programs according to their patterns of flow of control either directly extended or mutated in a constrained way. Then

we implemented a set of combination methods which combine programs from these classes. We found which methods gave better results on appropriate classes of programs and under certain restrictions. The results are judged qualitatively according to the efficiency of the resulting combined program. By using the program classification and methods, we produced an algorithm which permits users either to combine programs automatically or select a non-standard classification by navigating through the classification hierarchy before the combination of the pair of programs can be performed.

In short we can say that our solution to the combination problem can be defined in two stages. The first stage is the classification of the pair of programs using our hierarchy of classes of program and the second stage is to apply a sequence of transformation operations in order to obtain the combined program.

10.2 Current Restrictions

The assumptions on the implemented version of the composition system are as follows:

1. Our definition of skeleton is restricted to contain one argument which is used for traversing the data structure in our skeleton knowledge base.

2. Our composition system is restricted to the knowledge base of skeletons and techniques defined in Appendix C and Appendix D.

3. The binding of variables performed for our meta-folding operation is restricted to local variables which are used for traversing the same data structure (i.e. they have the same functionality to construct/deconstruct the data structure) or variables X, Y appearing in the same environment.

4. Our Particular method performs a transformation process before the fold operation can be performed. The restriction is that both programs need to be constructed either using the *count* or *sum* technique (for further details see Section 6.10).

5. Program history needs to be available. This can be obtained by means of a specialised editor. The current version of Robertson's editor does not record the program history but might be extended in order to record the *program history.*

6. Cuts were left out of our system: no skeletons or techniques have yet been devised making use of them and the composition system also falls short in combining programs using cuts. Future extensions should offer means to devise skeletons and techniques with sensible cuts and enable the combination of programs built using them.

10.3 Further Work

In this section we propose further work that can be undertaken in this field.

- Our composition system assumes the existence of program history knowledge derived from an editor which bases program construction on initial skeletons defining flows of control. We believe that constructing a planning system might be useful in order to get a program history more easily without a lot of overhead. Otherwise if the user needs to redo the program because he chooses an inappropriate skeleton (which can be used for his application) then the system needs to redo the program history as well. This is not a desirable characteristic, so we consider two plausible solutions to this problem.

  A partial solution to solving this problem is to provide a catalogue of examples in which the programmer can see some examples similar to the program that they want to build with some sort of explanation of how these examples have been constructed. A much more ambitious solution to this problem would be to implement a *planning system* in which the users can describe key aspects of the type of problem that he wants to solve. The output from this module would be a plan for the construction of the program. Therefore this module would be of particular importance at the user level. The specification of the problem requires a formal language in which

the user can express the main features of the problem that he wants to solve. The restriction is that this solution is not a general solution. It needs to be restricted to a specific domain, for instance, list processing. The characteristics that the user needs to describe are firstly the type of data structure, and with this information the system will create the cases for the induction parameter. Secondly a description of input and output modes is required at this stage and, finally, the user needs to describe semantic aspects such as whether the program will stop when a condition is met or whether it must recurse until the empty list is reached.

- It is useful to allow techniques which change the flow of control of the program rather than assuming that the flow of control will be determined at the start of the program design. The program's behaviour can be changed by adding subgoals which can conditionally terminate a clause or by adding new clauses in the program (see example in Chapter 7). A technique which changes the flow of control of the program is the *collect* technique. The *collect* technique is used with a skeleton whose clauses are mutually exclusive, which implies that at least one of the input arguments will be used for determining which clause to take and there is a special data item that is to be collected. The item collected is the item which satisfies the test. In Appendix D we have described a knowledge base of techniques which do not change the flow of control. This knowledge base of techniques was obtained from several sources [Kirschenbaum *et al.* 89]. However, more research needs to be carried out concerning the definition of which techniques map skeletons into 'mutations' (a type of enhancement where the control flow of a skeleton is slightly modified). Currently our set of techniques is restricted to techniques which do not change the flow of control.

- We would like to produce a standard set of skeletons and techniques. This set should be enough to allow the user to build a wide variety of Prolog programs. However, there is a tension between having general techniques which give little help in programming, versus specific techniques which are more useful but we need to

provide lots of them [Vasconcelos 93]. Currently in our set of techniques we have only specific techniques such sum, count, etc. (defined in Appendix D). Future work could be to show that it is possible to make optimisation in the combination of programs constructed using a larger variety of more general techniques.

- Another area of research is to extend the range of Prolog programs that can be combined by allowing the combination of programs which are written with special kind of cuts (green cuts). A plausible solution to this problem is to extend the standard unfolding operation allowing it to unfold clauses using cuts in their bodies. There are several methods that can be used in order to propagate the cut safely. As a first stage we present a program in which the cut is propagated incorrectly by using the standard unfold operation. For instance consider the following program:

```
p(X) :- q(X).
p(2).
q(0) :- !.
q(1).
```

By unfolding `q/1` we obtain the program shown below. This program is not equivalent to the initial program. The incorrectness arises when the cut defined in program `q/1` was propagated into program `p/1`.

```
p(0) :- !.
p(1).
p(2).
```

In order to avoid propagating the cut incorrectly, Venken [Venken & Demoen 88] proposes annotating cuts during unfolding to make their scope explicit. The annotated cut is called the ancestral cut. This is expressed by two predicates: `mark(v)` which succeeds on being called, binds v to a unique value and fails on backtracking; `!(v)` succeeds on being called and removes all choice points back to `mark(v)`. For example, consider the following example:

```
p(X) :- q(X), r(X).
p(2).
q(1) :- !.
q(3).
r(1) :- !.
```

We transform cuts in the programs into ancestral cuts by applying the approach described in [Prestwich 92], obtaining the following program:

```
1:  p(X) :- mark(V1), q(X,V1), mark(V2), r(X,V2).
2:  p(2).
3:  q(1,V1) :- !(V1).
4:  q(3,V1).
5:  r(1,V2) :- !(V2).
```

In order to unfold q we need to create an auxiliary predicate new/2 defined in clause 8 (which is obtained from clause 1 by copying the subgoals which come after the first annotation).

```
6:  p(X) :- mark(V1), new(X,V1).
7:  p(2).

8:  new(X,V1) :- q(X,V1), mark(V2), r(X,V2).
```

q is unfolded in the new clause obtaining the following program:

```
9:  new(1,V1) :- !(V1), mark(V2), r(1,V2).
10:  new(3,V1) :- mark(V2), r(3,V2).
```

By unfolding r in both clauses above, the first clause becomes:

```
11:  new(1,V1) :- !(V1), mark(V2), !(V2).
```

The second clause is removed via failure propagation. Failure propagation can be defined as follows:

Assuming C is a clause defined as $p : -a_1, \ldots, a_n$. If a_i does not match with any clause then C is replaced by a clause $p : -a_1, \ldots, a_n, fail.$ and the fail is propagated back through $a_1, \ldots a_n$.

Finally, new is unfolded in the clause 6.

```
p(X) :- mark(V1), !(V1), mark(V2), !(V2).
p(2).
```

- We propose to incorporate programs written using conventional editors (such as emacs) into this environment by having their components (skeletons and techniques) identified and their history extracted. One plausible approach is to analyse the program and find out which class of our classification it belongs to. Bental's work [Bental 94] appears to take a promising approach in recognising components of programs using a clausal split.

- We propose to find a formal proof for our observations described in Chapter 9 such as the observation on the simplification process performed in the particular method.

- We believe that implementing a single method which combines every pair of programs is useful work which could be undertaken (see page 80). This single method must handle all the conditions which are checked currently by each of our methods. So, we propose to have a method which gives two options: either to enforce the same traversal of the data structure or to do the full cartesian product. This method should use unfolding, meta-folding, arithmetic rules and structural knowledge of each of the programs to be combined (see section 5.3).

- Our composition system was not intended to be a tutoring system but we believe it might be used as component of such a system. We think it is feasible to build an explanation mechanism on top of our system. This explanation mechanism might give an explanation why the composition system choses a particular combination method and also, if the user is interested, it might explain each stage in the transformation of the program. If carefully deployed, using well chosen examples, this could help to make students more aware of the potential for reuse of existing programs (using combination) rather than continually writing code from scratch.

- We propose to improve the currently very limited interface. For example, in the non-automatic mode we would like to provide the possibility of displaying the decision tree, and so show the discriminating steps that the system took during the selection of the combination method. Also, we would expect our system to cooperate with a

techniques editor under a uniform interface (which should be compatible with commercial Prolog systems whenever feasible), however none of the current techniques editors have a significantly better interface. There is a potential here for cooperation between the communities. Our work would help supply the foundation for a usable system but the issue of interface design is outside the scope of this thesis: we would expect the design of an industrial quality interface to require a separate research project, as designing a good interface is a far from trivial task.

- We propose to use the meta-information (stored in the program history) in order to aid a high-level dialogue with the user. The system could talk in the language of control flows, and their associated functionalities, instead of in terms of low-level guidance of the combination method.

10.4 Contribution of this Thesis

This thesis presents a combination system based on program history. The composition system allows users to construct more complex programs by combining simpler Prolog programs, which could be built by means of a techniques editor. This composition system contains a set of methods for combining programs with either the same flow of control or with different flows of control. We compared the performance of different combination methods on pairs of programs noting which give better results on appropriate classes of program and what the restrictions on each of them. A detailed descriptions of these methods can be found in Chapter 6 and Chapter 7. We have implemented the *selection procedure* which automatically selects a composition method according to the features of the programs. The selection of the method which can be used is not an easy task, especially for Prolog beginners. Currently our composition system is capable of assisting in making these decisions by using the *program history.*

The composition system applies a suitable transformation rule automatically, obtained from the library of transformational rules, in order to compose two given programs. In

our transformation rules library we have several kinds of rules such as: unfolding, folding, meta-folding, goal merge and rules based on data domain knowledge (arithmetic rules) such as: commutativity, associativity, etc. The commutativity and associativity rules can be applied to relations such as addition over integer/real numbers.

The transformations necessary to produce efficient combined programs are complex, which makes it difficult for users to apply them reliably by hand. Our approach reduces the user interaction to answering simple questions in a high level language (of skeletons and techniques) and also produces optimised programs. In automatic mode it needs almost no user interaction, provided that it is given appropriate program history information for the classes of program recognised by the system.

In Chapter 9 we discuss some general properties which we look for in this type of composition system. These include: properties which hold after applying each combination method; properties for the type of program which is obtained at each stage of the combining process; and properties for the definition of the join specification.

The use of environments which provide help in the software development process are important in many domains, for instance in teaching environments. Although we do not claim to have produced such an environment, we believe that automated program composition is an important part of its foundations. If the user is provided with this kind of system then the following benefits can be obtained.

- Programmers can improve their performance by making use of methodologies and systems which allow them to concentrate on the problem to be solved, without unnecessary "mechanical" details.
- Our environment provides help in the construction of programs by assisting in the task of integrating several pieces of software, thus building more complex programs from simpler ones.

- This may assist us reusing standard programs for building different pieces of software.
- It may also assist in standardising programming by promoting a uniform coding style.

Bibliography

[Barker-Plummer 90] Dave Barker-Plummer. Cliche Programming in Prolog. In *Proceedings of the Second Workshop on Meta-programming in Logic*, Belgium, October 1990. META-90.

[Bental 92] Diana Bental. Using Clausal Join and Clausal Split to recognise language-specific Programming design Decisions. In *AAAI Conference Workshop on AI and Automated Program Understanding*, pages 37–40, 1992.

[Bental 94] Diana Bental. *Recognising the Design Decisions in Prolog Programs as a Prelude to Critiquing.* Unpublished PhD thesis, Department of Artificial Intelligence, Edinburgh University, 1994.

[Berzins 86] Valdis Berzins. On Merging Software Extensions. *Acta Informatica*, 23:607–619, 1986.

[Bossi *et al.* 90] A. Bossi, N. Cocco, and S. Dulli. A Method for Specializing Programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.

[Bowles 93] Andy Bowles. A Techniques Editor for Prolog novices. Internal note submitted for publication, DAI, 1993.

[Bowles *et al.* 94] Andy Bowles, David Robertson, Wamberto Vasconcelos, Maria Vargas-Vera, and Diana Bental. Applying Prolog Programming Techniques. *International Journal of Man Machine Studies*, 41(3):329–350, 1994.

[Bratko 86] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1986.

[Brna *et al.* 87] Paul Brna, Alan Bundy, Helen Pain, and L. Lynch. Programming Tools for Prolog Environments. In J. Hallam and C. Mellish, editors, *Advances in Artificial Intelligence*,

pages 251–264. Society for the Study or Artificial Intelligence and Simulation of Behaviour, John Wiley and Sons, 1987. Previously, DAI Research Paper No 302.

[Brna *et al.* 91] Paul Brna, Alan Bundy, Tony Dodd, Marc Eisenstadt, Chee Kit Looi, Helen Pain, Barbara Smith, and Maarten Van Someren. Prolog Programming Techniques. *Instructional Science*, 20:111–134, 1991.

[Bundy 88] Alan Bundy. Proposal for a Recursive Techniques Editor for Prolog. Technical Report 394, Department of Artificial Intelligence, University of Edinburgh, 1988.

[Bundy *et al.* 91] Alan Bundy, Gerd Grosse, and Paul Brna. A Recursive Techniques Editor for Prolog. *Instructional Science*, 20:135–172, 1991.

[Burstall & Darlington 77] Rod Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal ACM*, 24(1):44–67, 1977.

[Courcelle 90] Bruno Courcelle. *Handbook of Theoretical Computer Science.* Elsevier Science Publishers B.V., 1990.

[Deville 90] Yves Deville. *Logic Programming Systematic Program Development.* Addison-Wesley, 1990.

[Duncan 92] Gabriel Duncan. TX: Prolog Explanation System. Unpublished M.Sc. thesis, Department of Artificial Intelligence, Edinburgh University, 1992.

[Elgot 70] Calvin C. Elgot. Remarks on One-Argument Program Schemes. In Randall Rustin, editor, *Courant Computer Science Symposium 2*, pages 59–64. Prentice Hall, September 1970.

[Feather 78] Martin S Feather. ZAP Program Transformationn System Primer and Users'manual. Technical Report 54, Department of Artificial Intelligence, University of Edinburgh, 1978.

[Fuchs & Fromherz 91] Norbert E. Fuchs and Markus P. J. Fromherz. Schema-Based Transformations of Logic Programs. In *Logic Program Synthesis and Transformation, Workshops in Computing.* Springer Verlag, 1991.

[Gegg-Harrison 89] Timothy S. Gegg-Harrison. Basic Prolog Schemata. Tr 89-20, Department of Computer Science, Duke University, 1989.

[Gegg-Harrison 91] Timothy S. Gegg-Harrison. Learning Prolog in a Schema-Based Environment. *Instructional Science*, 20:173–192, 1991.

[Horwitz *et al.* 88] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Non-Interfering Versions of Programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[Kirschenbaum *et al.* 89] Marc Kirschenbaum, Arun Lakhotia, and Leon S. Sterling. Skeletons and Techniques for Prolog Programming. Tr 89-170, Case Western Reserve University, 1989.

[Lakhotia & Sterling 87] Arun Lakhotia and Leon Sterling. Composing Logic Programs with Clausal Join. Tr 87-25, Computer Engineering and Science Department, Case Western Reserve University, 1987.

[Lakhotia 89] Arun Lakhotia. Incorporating Programming Techniques into Prolog Programs. In Ewing L. Losk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 426–440. MIT Press, October 1989.

[Lloyd 87] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 2nd edition, 1987.

[O'Keefe 90] Richard O'Keefe. *The Craft of Prolog.* The MIT Press, 1990.

[Partsch & Steinbruggen 83] H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3):200–236, 1983.

[Péter 81] Rózsa Péter. *Recursive Functions in Computer Theory.* Ellis Horwood Limited, 1981.

[Prestwich 92] Steve Prestwich. An Unfold Rule for Full Prolog. In Kung-Kiu Lau and Tim Clement, editors, *Logic Program Synthesis and Transformation*, pages 199–213, Manchester, UK., 1992. Springer Verlag.

[Proietti & Pettorossi 92] Maurizio Proietti and Alberto Pettorossi. Best-first Strategies for Incremental Transformations of Logic Programs. In

Second International Workshop on Logic Program Synthesis and Transformation, 1992.

[Rich & Walters 86] Charles Rich and Richard C. Walters. *Artificial Intelligence and Software Engineering.* Morgan Kaufmann, 1986.

[Robertson 91] Dave Robertson. A Simple Prolog Techniques Editor for Novice Users. In G. A. Wiggins, C. Mellish, and T. Duncan, editors, *3rd UK Annual Conference on Logic Programming*, pages 190–205. Springer Verlag, April 1991.

[Sterling & Kirschenbaum 91] Leon S. Sterling and Marc Kirschenbaum. Applying Techniques to Skeletons. Tr 91-179, Case Western Reserve University, 1991.

[Sterling & Lakhotia 88] Leon Sterling and Arun Lakhotia. Composing Prolog Meta-Interpreters. In Kowalski and Bowen, editors, *5th Symposium of Logic Programming*, pages 386–403, 1988.

[Sterling & Shapiro 86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[Tamaki & Sato 83] Hisao Tamaki and Taisuke Sato. A Transformation System for Logic Programs which Preserves Equivalence. Tr 83-18, ICOT Research Center, 1983.

[Tamaki & Sato 84] Hisao Tamaki and Taisuke Sato. Unfold/Fold Transformation of Logic Programs. In *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Sweden, 1984.

[Ullman 86] Jeffrey D. Ullman. *Principles of Database Systems.* Computer Science Press, 1986.

[Vasconcelos 93] Wamberto Vasconcelos. Designing Prolog Programming Techniques. In Deville. Y., editor, *Third International Workshop on Logic Programming Synthesis and Transformation.* Springer Verlag, July 1993.

[Venken & Demoen 88] Raf Venken and Bart Demoen. A Partial Evaluation System for Prolog: some Practical considerations. *New Generation Computing*, 6:279–290, 1988.

Appendix A

Gegg-Harrison's schemata

Schema A processes all elements of a list. Programs included in this group are first_last/3 (which moves the first element to the end of the list) and last_first (which moves the last element to the front of the list). Other programs included in this group are: append/3, length/2 and reverse/2 defined in Appendix B.

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

An example of the use of the schema_A is the predicate append/3 which concatenates two lists into a third list. This program is produced by making the substitutions {append/schema_A, L,L/&1} and {L,[H|R]/&2, L,R/ &5} in schema_A and by instantiating pre_pred and post_pred to the null string. The program append/3 is defined as follows:

```
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

Schema B invokes itself twice (double recursion). One subschema of schema B calls itself recursively for each element that is a list, e.g. flatten. The other subschema is divide-and-conquer where the idea is to subdivide the list in two parts and process each part by invoking the main predicate recursively. Examples are quicksort/2,

mergesort/2, find_min/2 (finds the element having the lowest value in a list of integers), find_max/2 (finds the maximum element of a list of integers), binary_tree/1 (recognises if the first parameter is a binary tree) and the predicate height/2 (which computes the height of a binary tree assuming that the height of the empty tree is 0 and that of a one-element tree is 1). These examples are defined in Appendix B.

```
schema_B([], ≪ &1 ≫).
schema_B([H|T], ≪ &2 ≫) :-
            par_pred(≪ &3 ≫, H, ≪ &4 ≫),
            schema_B(&5, ≪ &6 ≫)
            schema_B(&7, ≪ &8 ≫)
            <,process(≪ &9 ≫)>.
```

An example of schema_B is the *find_max/2* predicate which finds the element having the highest value in a list of integers. This program can be obtained by applying the substitutions:

{find_max/schema_B, Max/&1, Max/&2, divide([H|T],H,A,B)/part_pred(≪ &3 ≫,H,≪ &4 ≫)}

and {A/&5, M1/&6, B/&7, M2/&8 } and by instantiating process(≪ &9 ≫) to the null string.

```
find_max([Max],Max).
find_max([H|T],Max) :-
            divide([H|T],H,A,B),
            find_max(A,M1),
            find_max(B,M2),
            find_max_aux(M1,M2,Max).
find_max_aux(M1,M2,M1) :-
            M1 > M2.
find_max_aux(M1,M2,M2) :-
            M1 >= M2.
```

Schema C processes the elements in a list until a specified element is reached. Examples of programs in this group are member/2 and adjacent/3 (which succeeds if the elements X and Y are consecutive elements of a list) defined in Appendix B.

```
schema_C([E|T], E, ≪ &1 ≫).
schema_C([H|T], E, ≪ &2 ≫) :-
            <E \= H>,
            < pre_pred(≪ &3 ≫, H, ≪ &4 ≫), >
            schema_C(T, E, ≪ &5 ≫)
            <, post_pred(≪ &6 ≫, H, ≪ &7 ≫) >.
```

An example is member/2 which succeeds if an element X is in a list. The predicate is defined in two clauses. X is an element of a list if it is the head of the list (first clause) or if it is a member of the tail of the list (second clause). The program `member/2` can be obtained by making the substitutions {`member/schema_C, null/&1, null/&2`} and by instantiating all the optional arguments and subgoals which are denoted by angle brackets i.e. the data argument:`<E\= H>`, and subgoals: `pre_pred` and `post_pred` to a null string.

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

Schema D processes all elements in the list after a specified element. An example of a program which can be built with this schema is find_sum/3 (which succeeds if the third argument is the summation of all the elements following the first occurrence of the given element in the list). The definition of the schema D is as follows:

```
schema_D([E|T], E, ≪ &1 ≫):- rec_pred(T, ≪ &2 ≫).
schema_D([H|T], E, ≪ &3 ≫) :-
        <E \= H>,
        < pre_pred(≪ &4 ≫, H, ≪ &5 ≫), >
        schema_D(T, E, ≪ &6 ≫)
        <, post_pred(≪ &7 ≫, H, ≪ &8 ≫) >.
```

The program find_sum/3 is obtained by making the following substitutions:

{`find_sum/schema_D, Sum/&1, Sum/&2`} and {`Sum/&3, Sum/&6`} and by instantiating all the optional arguments and subgoals such as: `rec_pred(≪ &4 ≫,H,≪ &5 ≫)` to `sum(T,Sum)` and subgoals `pre_pred` and `pre_pred post_pred` to null string.

```
find_sum([E|T],E,Sum):- sum(T,Sum).
find_sum([H|T],E,Sum) :-
            find_sum(T,E,Sum).
```

Schema E should be used if the goal is either to move an element in a specified position to the front of the list or to move the first element in the list to a specified position. Examples of programs in this group are find_nth/3, del_nth/3, subs_nth (find, delete and substitute the nth element of the list). These predicates are defined in Appendix B.

```
schema_E(L, 1, ≪ &1 ≫).
schema_E([H|T], P, ≪ &2 ≫) :-
        < pre_pred(≪ &3 ≫, H, ≪ &4 ≫), >
        (Q is P-1),
        schema_E(T, Q, ≪ &5 ≫)
        <, post_pred(≪ &6 ≫, H, ≪ &7 ≫) >.
```

An example is `find_nth/3` which succeeds if the element appears in the nth position in the List. This example can be produced by making the substitutions {`find_nth/schema_E, H/&1, R/&2, R/&5`} and by instantiating `pre_pred` and `post_pred` to null string. The program `find_nth/3` is defined as follows:

```
find_nth([H|T],1,H).
find_nth([H|T],Pos,R) :-
        N1 is Pos - 1,
        find_nth(T,N1,R).
```

Schema F should be used if the goal is either to move an element in a specified position to the end of the list or to move the last element in the list to a specified position. An example of a program constructed using schema F is pos_append/4 which inserts the second list after the nth element in the first list. This predicate is defined in Appendix B.

```
schema_F(L, 0, ≪ &1 ≫ ) :- rec_pred(L, ≪ &2 ≫).
schema_F([H|T], P, ≪ &3 ≫) :-
        < pre_pred(≪ &4 ≫, H, ≪ &5 ≫), >
        (Q is P -1),
        schema_F(T, Q, ≪ &6 ≫)
        <, post_pred(≪ &7 ≫, H, ≪ &8 ≫)>.
```

This program is obtained by making the substitutions:

{`pos_append/schema_F, L2,L3/&1, L2,L3/&2, append(L1,L2,L3)/rec_pred(L, ≪ &2 ≫)`}

and {`L2,[X|Zs]/&3, L2,Zs/&6`} and by instantiating `pred_pred` and `post_pred` to null string. This program `pos_append/4` is defined as follows:

```
pos_append(L1,0,L2,L3) :- append(L1,L2,L3).
pos_append([X|Xs],P,L2,[X|Zs]) :-
        Q is P -1,
        pos_append(Xs,Q,L2,Zs).
```

Gegg-Harrison also defines eight complex Prolog Schemata, from G to N, divided into two groups.

- Schemata G-J correspond to schemata C-F respectively, except that the processing the elements of the list is in reverse order.
- Schemata K-N are a composition of schemata C-D and G-H with schema D. The processing is before/after the nth occurrence of an element from the front/back of a list.

Appendix B

Definitions of some Predicates

first_last moves the first element in a list to the end of that list.

```
first_last([],[]).
first_last([H|T],L) :- append(T,[H],L).
```

last_first moves the last element in a list to the front of that list.

```
last_first([],[]).
last_first([H|T],[L,H|R]) :-
                append(R,[L],T).
```

append/3 concatenates two lists into a third list. This can be defined by the relation append(Xs,Ys,Zs) where Xs and Ys are the two lists that are to be joined and Zs is the joined list.

```
append([],Y,Y).
append([H|T],Y,[H|Z]) :- append(T,Y,Z).
```

len/2 computes the length of a list.

```
len([],0).
len([H|T],Len) :- len(T,Laux), Len is Laux+1.
```

The program *reverse/2* is defined as reverse(List,InList) where `InList` is the result of reversing the list `List`.

```
reverse([],[]).
reverse([H|T],L) :- reverse(T,M), append(M,[H],L).
```

member/3 finds if an element is in a List. X is an element of a list if it is the head of the list (first clause) or if it is a member of the tail of the list (second clause).

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

quicksort/2 sorts a list by choosing an arbitrary element and splitting the list into elements smaller than the chosen element and the elements larger than the chosen element. All elements greater than the element at the split point are placed in one list, and the elements less than the element at the split point are placed in another list. Then each of the resulted lists are themselves quick sorted. The sorted lists are appended to create the final sorted list.

```
quicksort([H|T],SortedList) :-
            split(T,H,Smalls,Bigs),
            quicksort(Smalls,Sm),
            quicksort(Bigs,Bs),
            append(Sm,[H|Bs],SortedList).
quicksort([],[]).

split([X|Xs],Y,[X|Sm],Bs) :- X =< Y, split(Xs,Y,Sm,Bs).
split([X|Xs],Y,Sm,[X|Bs]) :  X > Y, split(Xs,Y,Sm,Bs).
split([],Y,[],[]).
```

mergesort(List,Sorted)

In this method of sorting a list is first divided into two lists where the first element of List becomes part of List1, the second element of list becomes part of List2, the third element of List becomes part of List1, and so on. The process continues with the new lists being divided until single element lists are created. The single element lists are compared and merged according to which element should come first.

```
mergesort([],[]):- !.
mergesort([A],[A]) :- !.
mergesort(List,Sorted) :-
            divide(List,L1,L2),
            mergesort(L1,AL1),
            mergesort(L2,AL2),
            merge(AL1,AL2,Sorted).
```

divide(List,L1,L2)

This predicate divides a list into two halves. A list is divided by placing the first element in L1, the second element in L2, the third element in L1 and so on.

```
divide([],[],[]) :- !.
divide([A],[A],[]) :- !.
divide([A,B|T],[A|T1],[B|T2]) :-
            divide(T,T1,T2).
```

The merge predicate merges two list together. The first element of List $[Xs|Ys]$ is compared with the first element in list $[Y|Ys]$ and the larger element is added to the merged list. Then the first elements of each list are compared until all the elements have been compared and merged or until one list is empty, in which case all the elements in the remaining list are appended to the merged list.

```
merge([],L,L) :- !.
merge(L,[],L) :- !.
merge([X|Xs],[Y|Ys],[X|Zs]) :-
            X < Y,
            merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) :-
            X = Y,
            merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) :-
            X > Y,
            merge([X|Xs],Ys,Zs).
```

find_max finds the maximum element of a list of integers.

```
find_max([X],X).
find_max([X,Y|T],Max) :-
            X > Y,
            find_max([X|T],Max)
            ;
            find_max([Y|T],Max).
```

find_min finds the minimum element of a list of integers.

```
find_min([X],X).
find_min([X,Y|T],Min) :-
            X < Y,
            find_min([X|T],Min)
            ;
            find_min([Y|T],Min).
```

binary_tree recognises if the first parameter is a binary tree.

```
binary_tree(nil).
binary_tree(t(Left,X,Rigth)) :-
            binary_tree(Left).
            binary_tree(Right).
```

The predicate height(Binarytree,Height) computes the height of a binary tree assuming that the height of the empty tree is 0 and that of a one-element tree is 1.

```
height(nil,0).
height(t(Left,X,Right),Height) :-
            height(Left,LH),
            height(Right,RH),
            max(LH,RH,MH),
            Height is 1+MH.
```

flatten(+List,-FlatList) reorganises a list into a plain list (where the elements cannot be lists).

```
flatten([H|T],FlatList) :-
            flatten(H,Flathead),
            flatten(T,Flattail),
            append(Flathead,Flattail,FlatList).
flatten(X,[X]) :-
            constant(X),
            X \== [].
flatten([],[]).

constant(X) :-
            integer(X) ;
            atom(X).
```

adjacent/3 succeeds if the elements X and Y are consecutive elements of a List.

adjacent(+X,+Y,+List)

```
adjacent(X,Y,[X,Y|_]).
adjacent(X,Y,[_|Z]) :- adjacent(X,Y,Z).
```

el_last/3 moves a specified element to the back of the list.

```
el_last([H|T],1,R) :- append(T,[H],R).
el_last([H|T],N,[H|R]) :-
        N > 1,
        M is N - 1,
        el_last(T,M,R).
```

last_el/3 moves the last element of the list to precede a specified element.

```
last_el([H|T],1,[Last,H|R]) :-
        append(R,[Last],T).
last_el([H|T],N,[H|R]) :-
        N > 1,
        M is N - 1,
        last_el(T,M,R).
```

find_nth/3 succeeds if the element appears in position Pos to the List.

find_nth(+List,+Pos,+Elem)

```
find_nth([H|T],0,H).
find_nth([H|T],Pos,R) :- N1 is Pos-1, find_nth(T,N1,R).
```

del_nth/3 removes the nth-element from a List. del_nth+(List,+Pos,Elem)

```
del_nth([H|T],1,T).
del_nth([H|T],Pos,[H|R]) :- N1 is Pos - 1, del_nth(T,N1,R).
```

subs_nth/4 substitutes the nth element of the List.

subs_nth(+List,+Pos,+Value,+Elem)

```
subs_nth([H|T],1,V,[V|T]).
subs_nth([H|T],Pos,V,[H|R]) :- N1 is Pos-1, subs_nth(T,N1,V,R).
```

find_sum/3 succeeds if the third argument is the summation of all the elements following the first occurrence of the given element in the list.

```
find_sum([E|T],E,Sum) :- sum(T,Sum).
find_sum([H|T],E,Sum) :-
        find_sum(T,E,Sum).
```

post_append inserts the second list after the nth-element in the first list.

directionality: post_append(+List_1,+Pos,+List_2,-Final_List)

```
post_append([X|Xs],1,L2,[X|Zs]) :- append(L2,Xs,Zs).
post_append([X|Xs],P,L2,[X|Zs]) :-
            P > 1,
            Q is P -1,
            post_append(Xs,Q,L2,Zs).
```

Appendix C

Skeleton Knowledge Base

In this appendix a skeleton knowledge base is defined. This skeleton knowledge base contains the definitions of skeletons which can be used by the *techniques editor*. This classification was defined by Sterling and Kirschenbaum [Sterling & Kirschenbaum 91]. They divided skeletons into three categories. However our library only contains two categories meta-interpreters and manipulation of recursive data structures. The parser category of skeletons is not considered. This is not a fundamental limitation of our approach. We decide to limit the scope of our system just for simplicity.

Meta-Interpreters

A meta-interpreter for a language is an interpreter for the language written in the language itself. Prolog is a powerful language for meta-programming due to its symbol manipulation capabilities. A Prolog meta-interpreter takes a Prolog program and a Prolog goal and executes the goal with respect to the program. In others words, the meta-interpreter attempts to prove that the goal logically follows from the program. An example of a meta-interpreter is the well known basic meta-interpreter `solve/1` defined below, where `solve(Goal)` is `true` if `Goal` is true with respect to the program being interpreted. This meta-interpreter interprets a subset of Prolog excluding *cut*. In our work the skeleton `solve/1` has been used for the construction of programs such as debuggers and expert

systems (see Chapter 6).

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- sys(A), call(A).
solve(A) :- clause(A,B), solve(B).
```

A modulant skeleton from the previous skeleton `solve/1` is shown below. Here the `solve(B)` goal in clause number two is replaced by a new predicate `solve_conj(B)`. Unfolding the `solve_conj(B)` subgoal returns us to the first definition of `solve/1`.

```
solve(true).
solve((A,B)) :- solve(A), solve_conj(B).
solve(A) :- sys(A),call(A).
solve(A) :- clause(A,B), solve(B).

solve_conj(B) :- solve(B).
```

Manipulating Recursive Data Structures

In Prolog there are many skeletons concerned with list traversal. One of the skeletons that belongs to this family is the skeleton *traverse* which fully traverses a list. In this skeleton a list is deconstructed by removing head elements until the empty list is reached. The definition of this skeleton in the more general form is shown on page 11.

The simplest case is when there is no test in the skeleton. This skeleton is shown as follows:

```
traverse([H|T]) :- traverse(T).
traverse([]).
```

The skeleton *traverse* (basic case) shown above is the skeleton for many programs such as length/2, append/3, etc. This skeleton requires complete traversal of a list (until the empty list is reached). In fact this skeleton itself defines a list. We can generalise this to other structured terms. For instance, the following program defines a binary tree and also forms the skeleton for programs operating on trees.

```
tree_LR(nil).
tree_LR(t(L,X,R)) :-
            tree_LR(L), tree_LR(R).
```

The above skeleton performs a left to right depth-first traversal, if a different order (for instance right to left depth-first traversal) is required, a different skeleton must be defined.

```
tree_RL(nil).
tree_RL(t(L,X,R)) :-
        tree_RL(R), tree_RL(L).
```

The skeleton *short_traverse* will either traverse the entire list or stop when a condition is met. This skeleton allows for the general case when the elements of the list need to be distinguished in 'n' different ways.

```
short_traverse_n([H|T]) :- c1(H), short_traverse_n(T).
short_traverse_n([H|T]) :- c2(H), short_traverse_n(T).
                    ⋮
short_traverse_n([H|T]) :- cn(T).
short_traverse_n([]).
```

The simplest case is shown as follows:

```
short_traverse([H|T]) :- short_traverse(T).
short_traverse([H|T]) :- c1(H).
short_traverse([]).
```

The skeleton *search* traverses the list until what is being searched for has been found, and fails otherwise. This skeleton allows for the general case when the elements of the list need to be distinguished in 'n' different ways.

```
search_n([H|T]) :- c1(H), search_n(T).
search_n([H|T]) :- c2(H), search_n(T).
                ⋮
search_n([H|T]) :- cn(H), search_n(T).
```

The simplest case is shown as follows:

```
search([H|T]) :- c1(H), search(T).
search([H|T]) :- c2(H).
```

The skeleton for Prolog terms such as list and trees can be defined in a similar form. An example skeleton that can be used for traversing an arbitrary Prolog term is the skeleton *trav_term/1*, shown below.

```
trav_term(Var) :- var(Var).
trav_term(Term) :-
            functor(Term,F,A),
            term_aux(A,Term).
term_aux(0,_Term).
term_aux(Arg,Term) :-
            Arg > 0,
            arg(Arg,Term,A),
            trav_term(A),
            N1 is Arg - 1,
            term_aux(N1,Term).
```

The *double_recursion* skeleton can be useful in the construction of programs such as quicksort/2, mergesort/2, find_min/2 (find the minimum element of a list of integers), find_max/2 (find the maximum element of a list of integers) , binary_tree/1 (recognise if the given first parameter is a binary tree) and the predicate height/2 (which computes the height of a binary tree, assuming that the height of the empty tree is 0 and that of a one-element tree is 1). All these examples are defined in Appendix B.

```
double_rec([H|T]) :- split([H|T],R,S), double_rec(R), double_rec(S).
double_rec([]).
```

Where *split* is a predicate which divides the initial list into two lists.

The *count_down* skeleton counts down a number. This skeleton is defined as follows:

```
count_down(N) :- N=0.
count_down(N) :-
            N>0,
            N1 is N - 1,
            count_down(N1).
```

In a similar fashion the *count_ up* skeleton, which counts up a number, is defined as follows:

```
count_up(N,C) :- N=C.
count_up(N,C) :-
            N1 is N + 1,
            count_up(N1,C).
```

Currently our library has the of set of skeletons defined above, but more research is being carried out on extending this set.

Appendix D

Techniques Knowledge Base

Lakhotia classifies techniques into three types: propagate context down, propagate context up and the accumulator technique [Lakhotia 89]. The techniques corresponding to propagate context down are used only in input mode. They propagate information down in an execution tree. The techniques classified as propagate context up use context only in output mode. They are used to propagate upwards in the proof tree information computed at deeper levels. Finally, the techniques belonging to the accumulator type use some context variables for input and others for output. The results from partial computations are propagated down the execution tree until the base condition is reached. At this stage the incoming context is used to compute outgoing context and the result is propagated up.

Some examples of techniques are as follows. The notation *name_technique*(*name_skeleton*) denotes the resulting schema after the technique is applied to the skeleton and *traverse_n* denotes the traverse skeleton with n clauses.

The technique *context* or also called *add_carrier* can be classified as propagate context down. It adds arguments with the purpose of providing input to each clause to be used for case analysis or tests. This context argument, in our case D, should be thought of as a term that is passed unchanged through a recursion. The technique *context* applied to the skeleton traverse can be defined as follows:

context(traverse_n) =

```
traverse_n_context([H|T],D) :- c1(H), traverse_n_context(T,D).
traverse_n_context([H|T],D) :- c2(H), traverse_n_context(T,D).
                    ⋮
traverse_n_context([H|T],D) :- cn(H), traverse_n_context(T,D).
traverse_n_context([],D).
```

The technique *calculate* can be classified as "propagate context up". It adds an extra argument in the skeleton and an extra arithmetic subgoal to the body of each recursive clause in order to relate the calculation from the body to the final result in the head of the clause. This technique can be thought of as family of techniques, one for each type of skeleton. There are various arithmetic operations that could be added to the skeleton as an extra goal. Each of these operations leads to a specific form of "calculate". The following two techniques (*count* and *sum*) are special cases of the calculate technique.

The technique *count* adds one to the value returned from the recursive call. The technique *count* applied to the skeleton traverse can be defined as shown below.

count(traverse_n) =

```
traverse_n_count([H|T],N) :- c1(H), traverse_n_count(T,N1), N is N1+1.
traverse_n_count([H|T],N) :- c2(H), traverse_n_count(T,N1), N is N1+1.
                    ⋮
traverse_n_count([H|T],N) :- cn(A), traverse_n_count(T,N1), N is N1+1.
traverse_n_count([],0).
```

The technique *sum* is similar to *count*, except that instead of always adding one to the argument the value of some argument is added. The technique *sum* applied to the skeleton traverse can be defined as shown below.

sum(traverse_n) =

```
traverse_n_sum([H|T],N) :- C1(H), traverse_n_sum(T,N1), N is N1+A.
traverse_n_sum([H|T],N) :- c2(H), traverse_n_sum(T,N1), N is N1+A.
                    ⋮
traverse_n_sum([H|T],N) :- cn(H), traverse_n_sum(T,N1), N is N1+A.
traverse_n_sum([],0).
```

The techniques *two_accumulator* and *back_accumulate* are examples of techniques belonging to the accumulator type.

The technique *two_accumulator* is used frequently in recursion. This technique allows building up a structure in recursive subgoals. Two arguments are added, one is used as a stack to push down data and the second argument returns the final object from the stack. In the definition of *two_accumulator/3*, the second argument is called *accumulator* and the third argument is called *result* (this output argument corresponds to the final state of the variable). The two arguments `S` and `F` together are called an accumulator pair. The technique *two_accumulator* applied to the skeleton traverse can be defined as follows:

```
two_accumulator([H|T],S,F) :-
            two_accumulator(T,[H|S],F).
two_accumulator([],F,F).
```

The technique *back_accumulate* (in Robertson's terminology) allows the addition of an accumulator to a recursive program. The technique *back_accumulate* works by adding an extra argument `Result` to the head of the clause; adding an extra `Partial_Result` to the recursive subgoal; and finally adding an extra subgoal (represented by `update_value/2`) for obtaining the value of `Result` from `Partial_Result` [Robertson 91]. The technique *back_accumulate* applied to the skeleton traverse can be defined as shown below.

```
traverse([H|T],Result) :-
            traverse(T,Partial_Result),
            update_value(Partial_Result,Result).
traverse([],Result).
```

The technique `max(X,Y,Z)` succeeds if `Z` is the largest number of `X` and `Y`. This predicate is defined as follows:

```
max(X,Y,Y) :- X <= Y.
max(X,Y,X) :- X > Y.
```

The technique which computes the certainty factor named `compute_cf` is defined as follows:

```
compute_cf(not(A)) :- 1 - compute_cf(A).
compute_cf(A & B) :- min(compute_cf(A),compute_cf(B)).
compute_cf(A or B) :- max(compute_cf(A),compute_cf(B)).
```

The above list of techniques is not exhaustive. More research is required to determine a broad coverage of techniques which will be enough for the construction of a large set of Prolog programs.

Appendix E

Tamaki and Sato's Algorithm

Tamaki and Sato formulated an unfold/fold transformation method for logic programs in such a way that the transformation always preserves the equivalence of programs in the least Herbrand model [Tamaki & Sato 84]. They define a set of transformation operations such as definition, unfolding, folding and goal merge. These operations are defined in a restricted way in order to preserve total correctness of logic programs, i.e. to produce equivalent programs.

The transformation process based in definition, unfold and fold rules can be described in the algorithm which is shown below. For this algorithm a program is a set of *definite clauses* as defined in Chapter 2.

The initial conditions for the algorithm are defined as follows:

1. All the clauses in the program are in a set $P_0 := \{p_1, \ldots, p_n\}$ where n is the number of clauses in program P .
2. $D_0 := \{\}$ is the set of definitions of the new predicate.

Every clause in P_0 is 'foldable'. A clause is 'foldable' if it can be used in the folding process.

1. For $i := 1$ to arbitrary N apply any of the transformation rules to obtain P_i and D_i from D_{i-1}.

Definition rule:

Let C be a clause of the form $p(X_1, \ldots, X_n) \ :- \ B_1, \ldots, B_m$ where

- p is an arbitrary predicate which does not appear in P_{i-1} or D_{i-1}.
- $X_1, \ldots, X_n$ are distinct variables
- $B_1, \ldots, B_m$ are subgoals whose predicates all appears in P_0. Therefore let P_i be $P_{i-1} \cup \{C\}$ and D_i be $D_{i-1} \cup \{C\}$. Note that C is not marked as 'foldable'.

The auxiliary predicates introduced by the definition rule are called *new predicates.*

Unfolding rule:

Let C be a clause in P_{i-1}, A a subgoal in its body and $C_1, \ldots, C_n$ all the clauses in P_{i-1} whose heads are unifiable with A. Let C_i' be the result of resolving C with C_i on A. Therefore let P_i be $(P_{i-1} - \{C\}) \cup \{C_1', \ldots, C_n'\}$ and D_i be D_{i-1} and mark each C_i' as foldable unless it is in P_{i-1}.

Folding rule:

Let C be a clause in P_{i-1} of the form $A \ :- \ A_1, \ldots, A_n$ and C_1 be a clause in D_{i-1} of the form $B \ :- \ B_1, \ldots, B_m$. Assume there is a substitution θ and subset $\{A_{i1}, \ldots, A_{im}\}$ of the body of C such that the following conditions hold.

(a) $A_{ij} = B_j\theta$ for $j = 1, \ldots, m$

(b) θ substitutes distinct variables for the internal variables of C_1, and these variables do not occur in A .

(c) C is marked as foldable or $m < n$

Then let P_i be $(P_{i-1} - \{C\}) \cup \{C'\})$ and D_i be D_{i-1} where C' is a clause with head A and body $(\{A_1, \ldots, A_n\} - \{A_{i1}, \ldots, A_{im}\}) \cup \{B\theta\}$.

For example, consider the program `len/2` which computes the length of a list and the program `sum/2` which computes the sum of the elements of a list. For this example, $P_0 = \{\underline{C1}, \underline{C2}, \underline{C3}, \underline{C4}\}$ and $D_0 = \{\}$.

```
C1 : len([H|T],Len) :-
          len(T,Len1),
          Len is Len1 + 1.
C2 : len([],0).

C3 : sum([H|T],Sum) :-
          sum(T,Sum1),
          Sum is Sum1 + H.
C4 : sum([],0).
```

We define C5 as the following predicate:

```
C5 : sum_len(List,Sum,Len) :-
          sum(List,Sum), len(List,Len).
```

Then $P_1 = \{\underline{C1}, \underline{C2}, \underline{C3}, \underline{C4}, C5\}$ and $D_1 = \{C5\}$, where underline indicates foldable clauses.

The second stage is to unfold $C5$ at its first and second subgoal to obtain P_2 such as $P_2 = \{\underline{C1}, \underline{C2}, \underline{C3}, \underline{C4}, C6, C7\}$ and $D_2 = \{C5\}$ where the clauses C6 and C7 are shown below.

```
C6:  sum_len([H|T],Sum,Len) :-
          sum(T,Sum1),
          Sum is Sum1 + H,
          len(T,Len1),
          Len is Len1 * 1.
C7:  sum_len([],0,0).
```

In Tamaki's work only clauses in D_{i-1} are allowed to be used in folding. The previous example can be continued by folding the body of clause C6 by using C5. We thus obtain $P_3 = \{\underline{C1}, \underline{C2}, \underline{C3}, \underline{C4}, C7, C8\}$ and $D_3 = \{C5\}$ where C8 is defined as follows:

```
C8 : sum_len([H|T],Sum,Len) :-
          sum_len(T,Sum1,Len1),
          Sum is Sum1 + 1,
          Len is Len1 + 1.
```

The complete example is formed by the set P_3 which is equivalent (in the least Herbrand model) to $P_0 \cup D_3$.

As was described before, there are several systems (including the one in this thesis) which make use of transformation rules. These transformation rules provide a way of achieving program efficiency by applying well defined sequences of transformations such as folding and unfolding. The key stage in the application of a correct sequence of transformations normally relies on human interaction, or use of a heuristic. We have attempted to reduce the need for interaction as far as possible.

Appendix F

Example new_fuzzydepth/4

In this appendix we take the meta-interpreters `new_fuzzy/3` and `fuzzydepth/3` from section 6.8.1 and form the *naive* combination by use of the procedural join method. The resulting combined program is inefficient and furthermore we shall see that it is probably not the program that the user would want. For a better method to combine these meta-interpreters see section 6.8.

Recall that the meta-interpreter `new_fuzzy/3` returns the associated certainty factor and explanation for a query and the meta-interpreter `fuzzydepth/3` attempts to compute the certainty factor for a query but is limited by the depth to which it can recurse. Additionally, in order to test the program, we shall use hypothetical (unrealistic) data given by the following set of definitions of the predicates `rule/3` and `fact/2`.

```
rule(disease(flu),Conditions,1.0):-
            Conditions = symptom(temperature,high).
rule(disease(flu),Conditions,1.0):-
            Conditions = not(symptom(temperature,high)).
rule(disease(flu),Conditions,1.0):-
            Conditions = symptom(not_dead) & symptom(temperature,high).
rule(symptom(not_dead),Conditions,1.0):-
            Conditions = symptom(moves).


            fact(symptom(discharge_from_eyes,present),1.0).
            fact(symptom(discharge_from_nose,present),0.5).
            fact(symptom(temperature,high),1.0).
            fact(symptom(moves),1.0).
```

Now suppose that the user wants to combine these two meta-interpreters. Most transformation systems will take the join specification to be the Prolog program.

```
new_fuzzydepth1(Goal,Cf,Exp,Depth) :-
        new_fuzzy(Goal,Cf,Exp),
        fuzzydepth(Goal,Depth,Cf).
```

and then convert this to a logically equivalent program. We can do this conversion by applying the procedural join method to the pair of meta-interpreters, and so obtain

```
new_fuzzydepth1(A,B,fact(A,B), _Depth ) :-
        fact(A,B).
new_fuzzydepth1(not(E),B,fact(not(E),B), Depth ) :-
        fact(not(E),B),
        fuzzydepth(E,Depth,F),
        B is 1-F.
new_fuzzydepth1(A,B,fact(A,B), Depth ) :-
        fact(A,B),
        Depth >0,
        D is Depth-1,
        rule(A,E,F),
        fuzzydepth(E,D,G),
        B is F*G.
new_fuzzydepth1((D & E),B,fact((D & E),B), Depth ) :-
        fact((D & E),B),
        fuzzydepth(D,Depth,F),
        fuzzydepth(E,Depth,G),
        min(F,G,B).
new_fuzzydepth1((D or E),B,fact((D or E),B), Depth ) :-
        fact((D or E),B),
        fuzzydepth(D,Depth,F),
        fuzzydepth(E,Depth,G),
        max(F,G,B).
new_fuzzydepth1(not(D),B,not(D,B,F),_Depth) :-
        new_fuzzy(D,E,F),
        B is 1-E,
        fact(not(D),B).
new_fuzzydepth1(not(E),B,not(E,B,F),Depth) :-
        new_fuzzy(D,E,F),
        B is 1 - E,
        fuzzydepth(D,Depth,G),
        B is 1 - G.
```

```
new_fuzzydepth1(not(D),B,not(D,B,F),Depth) :-
            new_fuzzy(D,E,F),
            B is 1-E,
            Depth >0,
            H is Depth -1,
            rule(not(D),I,J),
            fuzzydepth(I,H,K),
            B is J*K.
new_fuzzydepth1(A,B,rule(A,B,G),_Depth) :-
            rule(A,D,E),
            new_fuzzy(D,F,G),
            B is E*F,
            fact(A,B).
new_fuzzydepth1(not(E),B,rule(not(E),B,G),Depth) :-
            rule(not(E),D,H),
            new_fuzzy(D,F,G),
            B is H*F,
            fuzzydepth(E,Depth,F),
            B is 1-F.
new_fuzzydepth1(A,B,rule(A,B,H),Depth) :-
            rule(A,D,E),
            new_fuzzy(D,F,H),
            B is E*F,
            Depth >0,
            D1 is Depth -1,
            rule(A,I,J),
            fuzzydepth(I,D1,K),
            B is J*K.
new_fuzzydepth1((D & E),B,rule((D & E),B,G),Depth) :-
            rule((D & E),J,K),
            new_fuzzy(J,F,G),
            B is K*F,
            fuzzydepth(D,Depth,H),
            fuzzydepth(E,Depth,I),
            min(H,I,B).
new_fuzzydepth1((D or E),B,rule((D or E),B,G),Depth) :-
            rule((D or E),J,K),
            new_fuzzy(J,F,G),
            B is K*F,
            fuzzydeph(D,Depth,H),
            fuzzydepth(E,Depth,I),
            max(H,I,B).
new_fuzzydepth1((D & E),B,conj(G,I,B),_Depth) :-
            new_fuzzy(D,F,G),
            new_fuzzy(E,H,I),
            min(F,H,B),
            fact((D & E),B).
```

```
new_fuzzydepth1((D & E),B,conj(G,I,B),Depth) :-
            new_fuzzy(D,F,G),
            new_fuzzy(E,H,I),
            min(F,H,B),
            Depth >0,
            J is Depth - 1,
            rule((D & E), K,L),
            fuzzydepth(K,J,M),
            B is L*M.
new_fuzzydepth1((D & E),B,conj(G,I,B), Depth) :-
            new_fuzzy(D,F,G),
            new_fuzzy(E,H,I),
            min(F,H,B),
            fuzzydepth(D,Depth,J),
            fuzzydepth(E,Depth,K),
            min(J,K,B).
new_fuzzydepth1((D or E),B,disj(G,I,B),_Depth) :-
            new_fuzzy(D,F,G),
            new_fuzzy(E,H,I),
            max(F,H,B),
            fact((D or E),B).
new_fuzzydepth1((D or E),B,disj(G,I,B),Depth) :-
            new_fuzzy(D,F,G),
            new_fuzzy(E,H,I),
            max(F,H,B),
            Depth >0,
            J is Depth-1,
            rule((D or E),K,L),
            fuzzydepth(K,J,M),
            B is L*M.
new_fuzzydepth1((D or E),B,disj(G,I,B),Depth) :-
            new_fuzzy(D,F,G),
            new_fuzzy(E,H,I),
            max(F,H,B),
            fuzzydepth(D,Depth,J),
            fuzzydepth(E,Depth,K),
            max(J,K,B).
```

In this case, if we give the query `new_fuzzydepth1(disease(flu),Cf,Exp,1)` then Prolog will give three answers

```
Cf = 1.0
Exp = rule(disease(flu),1.0,fact(symptom(temperature,high),1.0))

Cf = 0.0
Exp = rule(disease(flu),0.0,not(symptom(temperature,high),0.0,
      fact(symptom(temperature,high),1.0)))

Cf = 1.0
Exp = rule(disease(flu),1.0,conj(rule(symptom(not_Dead),1.0,
      fact(symptom(moves),1.0)),fact(symptom(temperature,high),1.0),1.0))
```

In our opinion, this would usually not correspond to the users intentions. Instead, it seems more likely that the user would want to combine the functionalities of the programs to get

> `new_fuzzydepth2(Goal,Cf,Exp,Depth)`:
> finds `proof` for a given `Goal` with certainty factor `Cf`, and
> explanation `Exp`.

in which case the last solution would be unwanted, because it did not enforce the same traversal in both clauses, and hence the `Depth` limit was not forced to correspond to the explanation provided. Systems that rely only on the join specification written in the Prolog form have no way to know that the user wanted exactly the same `Goal`. In other words, we want to synchronise the tree traversal performed in `new_fuzzy/3` and `fuzzydepth/3`, but there is no way to express this requirement directly in Prolog using only the given predicates.

However, our system does not rely on the Prolog form of the join specification, but effectively has an extended join specification which we can write as

```
new_fuzzydepth2(Goal,Cf,Exp,Depth) ⇐
        new_fuzzy(Goal,Cf,Exp),
        fuzzydepth(Goal,Depth,Cf).
```

where the underlined arguments provide flows of control which we want to synchronise. Since we assume knowledge of the history of development of the program we can check which arguments were intended to provide the flow of control and assess whether they will be compatible in combination. In this case it will observe (from the program history) that both programs have the same flow of control, called "meta-interpreter"[1], and they were constructed using the same techniques. We assume that the user would also like the flow of control of the output to be "meta-interpreter" and would simply like to have the compute_cf technique (which computes certainty factor) added. As described in Section 6.8 this gives the output program `new_fuzzydepth/4` as defined on page 128 which will indeed only return the first two (desired) solutions and not the last (undesired) solution.

[1]So-named because its function is to allow us to create meta-interpreters

Combined Program	
Meta-composition	Procedural join
5 clauses	19 clauses
fully recursive program	still uses the original programs
Efficiency = α	Efficiency = β

Table F.1: Comparison between Combined Program generated using Meta-composition and Procedural Method

Finally, in table F.1, we give some other differences between the program generated using the procedural join (presented in this appendix) and the program generated using the meta-composition method (see page 128). We can see that the program from the procedural join is much larger than the one from the meta-composition method, and this justifies the lower efficiency ranking. It is for this reason that the (meta,meta) entry in table 6.3 (for the procedural join) only gets β in our estimation of efficiency.

Appendix G

Example Using a Join Specification Involving OR's

The aim of this appendix is to show how we might fail to achieve efficient programs if we combine programs using a join specification which uses in its definition or instead of and. For example, consider the programs new_fuzzy/3 and fuzzydepth/3 defined below. In Section 6.8.1 we also use an enhanced program new_fuzzy/3 and fuzzydepth/3.

```
P1: new_fuzzy(A,B,fact(A,B)) :-
      fact(A,B).

P2: new_fuzzy(A,B,rule(A,B,D,G)):-
      rule(A,D,E),
      new_fuzzy(D,F,G),
      B is E * F.

P3: new_fuzzy(D & E,B,conj(G,I,B)):-
      new_fuzzy(D,F,G),
      new_fuzzy(E,H,I),
      min(F,H,B).
```

```
Q1: fuzzydepth(A,B,C) :-
      fact(A,B).

Q2: fuzzydepth(A,B,C) :-
      B > 0,
      D is B - 1,
      rule(A,E,F),
      fuzzydepth(E,D,G),
      C is F * G.

Q3: fuzzydepth(D & E,B,C) :-
      fuzzydepth(D,B,F),
      fuzzydepth(E,B,G),
      min(F,G,C).
```

The two meta-interpreters new_fuzzy/3 and fuzzydepth/3 defined above will be combined using the join specification shown below. Note that this join specification is defined using or in its body instead that our standard definition which only use ands.

```
new_fuzzydepth(Goal,Cf,Exp,Depth) ⇐
        new_fuzzy(Goal,Cf,Exp);
        fuzzydepth(Goal,Depth,Cf).
```

Step 1: the first combined clause obtained by taking the first clause of program new_fuzzy/3 and the first clause of program fuzzydepth/3 is as follows:

```
T1 : new_fuzzydepth(A,B,fact(A,B),Depth) :-
                fact(A,B);
                fact(A,B).
```

We obtain the combined clause T_1 by applying the merge operation which removes the syntactically identical subgoals.

```
T1 : new_fuzzydepth(A,B,C,Depth) :-
                fact(A,B).
```

Step 2: taking the second clause of program `new_fuzzy/3` and the second clause of program `fuzzydepth/3` and applying the unfolding operation we obtain:

```
T2 : new_fuzzydepth(A,B,rule(A,B,E,H),Depth) :-
                rule(A,E,F),
                new_fuzzy(E,G,H),
                B is F * G;
                Depth > 0,
                D1 is Depth - 1,
                rule(A,E1,F1),
                fuzzydepth(E1,D1,G1),
                B is F1 * G1.
```

By applying the meta-folding operation the system infers that the variable `E` can be unified to `E1`, that the variable `F` can be unified to `F1` and that the variable `G` can be unified to `G1`. So, the previous clause T_2 is as follows:

```
T2 : new_fuzzydepth(A,B,rule(A,B,E,H),Depth) :-
                rule(A,E,F),
                new_fuzzy(E,G,H),
                B is F * G;
                Depth > 0,
                D1 is Depth - 1,
                rule(A,E,F),
                fuzzydepth(E,D1,G),
                B is F * G.
```

By transforming the clause T_2 using De Morgan's laws in order to get the conditions for the application of the folding operation (i.e. a conjunction or disjunctions), we obtain:

```
T2 : new_fuzzydepth(A,B,rule(A,B,E,H),Depth) :-
                (rule(A,E,F) ; Depth > 0),
                (rule(A,E,F) ; D1 is Depth-1),
                (rule(A,E,F) ; fuzzydepth(E,D1,G)),
                (new_fuzzy(E,G,H),Depth > 0),
                (new_fuzzy(E,G,H), D1 is Depth-1),
                (new_fuzzy(E,G,H),fuzzydepth(E,D1,G)),
                (B is F * G ; Depth > 0),
                (B is F * G ; D1 is Depth-1),
                (B is F * G ; fuzzydepth(E,D1,G)).
```

The combined T_2 clause is as follows:

```
T2 : new_fuzzydepth(A,B,rule(A,B,E,H),Depth) :-
            (rule(A,E,F) ; Depth > 0),
            (rule(A,E,F) ; D1 is Depth-1),
            (rule(A,E,F) ; fuzzydepth(E,D1,G)),
            (new_fuzzy(E,G,H),Depth > 0),
            (new_fuzzy(E,G,H), D1 is Depth-1),
            (new_fuzzydepth(E,G,H,D1)),
            (B is F * G ; Depth > 0),
            (B is F * G ; D1 is Depth-1)),
            (B is F * G ; fuzzydepth(E,D1,G)).
```

Finally this clause can be rewritten using associativity as follows:

```
T2 : new_fuzzydepth(A,B,rule(A,B,E,H),Depth) :-
          rule(A,E,F) ; (Depth > 0, D1 is Depth-1, fuzzydepth(E,D1,G)),
          (new_fuzzy(E,G,H) ; (Depth > 0, D1 is Depth-1)),
          new_fuzzydepth(E,G,H,D1),
          (B is F * G; (Depth > 0, D1 is Depth-1, fuzzydepth(E,D1,G))).
```

Step 3: in a similar fashion the combined clause T_3 is obtained. By taking the third clause from each program and by unfolding.

```
T3 : new_fuzzydepth(D & E,B,conj(H,J,B),Depth) :-
            new_fuzzy(D,G,H),
            new_fuzzy(E,I,J),
            min(G,I,B);
            fuzzydepth(D,Depth,F1),
            fuzzydepth(E,Depth,G1),
            min(F1,G1,B).
```

In this clause we need to infer that the variable `G` can be unified to `F1` and that the variable `I` can be unified to `G1`.

```
T3 : new_fuzzydepth(D & E,B,conj(H,J,B),Depth) :-
            new_fuzzy(D,G,H),
            new_fuzzy(E,I,J),
            min(G,I,B);
            fuzzydepth(D,Depth,G),
            fuzzydepth(E,Depth,I),
            min(G,I,B).
```

By applying the merge goal operation we obtain:

```
T3 : new_fuzzydepth(D & E,B,conj(H,J,B),Depth) :-
                new_fuzzy(D,G,H),
                new_fuzzy(E,I,J),
                min(G,I,B);
                fuzzydepth(D,Depth,G),
                fuzzydepth(E,Depth,I).
```

By transforming the previous clause using the associativity laws we obtain:

```
T3 : new_fuzzydepth(D & E,B,conj(H,J,B),Depth) :-
            (new_fuzzy(D,G,H) ; fuzzydepth(D,Depth,G)),
            (new_fuzzy(D,G,H) ; fuzzydepth(E,Depth,I)),
            (new_fuzzy(E,I,J) ; fuzzydepth(D,Depth,G)),
            (new_fuzzy(E,I,J) ; fuzzydepth(E,Depth,I)),
            (min(G,I,B) ; fuzzydepth(D,Depth,G)),
            (min(G,I,B) ; fuzzydepth(E,Depth,I)).
```

Applying the folding operation we obtain:

```
T3 : new_fuzzydepth(D & E,B,conj(H,J,B),Depth) :-
            new_fuzzydepth(D,G,H,Depth),
            (new_fuzzy(D,G,H) ; fuzzydepth(E,Depth,I)),
            (new_fuzzy(E,I,J) ; fuzzydepth(D,Depth,G)),
            new_fuzzydepth(E,I,J,Depth),
            (min(G,I,B) ; fuzzydepth(D,Depth,G)),
            (min(G,I,B) ; fuzzydepth(E,Depth,I)).
```

By applying the associativity laws we get the following clause:

```
T3 : new_fuzzydepth(D & E,B,conj(H,J,B),Depth) :-
          new_fuzzydepth(D,G,H,Depth),
          (new_fuzzy(D,G,H) ; fuzzydepth(E,Depth,I)),
          (new_fuzzy(E,I,J) ; fuzzydepth(D,Depth,G)),
          new_fuzzydepth(E,I,J,Depth),
          (min(G,I,B) ; (fuzzydepth(D,Depth,G), fuzzydepth(E,Depth,I))).
```

The final combined program is shown as follows:

T_1 :
```
new_fuzzydepth(A,B,C,Depth) :-
        fact(A,B).
```

T_2 :
```
new_fuzzydepth(A,B,rule(A,B,E,H),Depth) :-
        rule(A,E,F) ; (Depth > 0, D1 is Depth-1, fuzzydepth(E,D1,G)),
        (new_fuzzy(E,G,H); (Depth > 0, D1 is Depth-1)),
        new_fuzzydepth(E,G,H,D1),
        (B is F * G ; (Depth > 0, D1 is Depth-1, fuzzydepth(E,D1,G))).
```

T_3 :
```
new_fuzzydepth(D & E,B,conj(H,J,B),Depth) :-
        new_fuzzydepth(D,G,H,Depth),
        (new_fuzzy(D,G,H) ; fuzzydepth(E,Depth,I)),
        (new_fuzzy(E,I,J) ; fuzzydepth(D,Depth,G)),
        new_fuzzydepth(E,I,J,Depth),
        (min(G,I,B) ; (fuzzydepth(D,Depth,G),fuzzydepth(E,Depth,I))).
```

In conclusion we can say that our transformation operation is still valid if we pre-process each unfolded clause before the folding operation can be applied, but the combined program is not efficient. Therefore for combining programs using or in its definition of join specification might be better to combine using the synchronization method (which is defined in Section 6.5).

Appendix H

Burstall Terminology

The following set of definitions were taken from Burstall and Darlington [Burstall & Darlington 77].

Primitive functions a set of primitive function symbols $k, l, \ldots$ and $c, d, \ldots$ with zero or more arguments; the subset $c, d, \ldots$ of primitive symbols are the constructor function symbols. Examples of constructor functions are *cons* and *successor*.

Parameters a set $x, y, \ldots$ of parameter variables.

Recursive functions a set $f, g, \ldots$ of recursive function symbols.

Expression is an expression built using function symbols, parameter variables and recursive function symbols. The *where* construction is permitted in the form
E *where* $< u, \ldots, w >= F$ or E *where* $u = F$. Here, E and F are expressions and $u, \ldots, w$ are variables, for instance: $u + u^2$ *where* $u = a + b$.

A *left-hand expression* is of the form $f(e_1, \ldots, e_n)$ where $e_1, \ldots, e_n$ are expressions involving parameter variables and constructor function symbols except *where*.

A *right-hand expression* is an expression as was defined above.

A *recursion equation* consist of a left-hand expression and a right-hand expression written $E \Leftarrow F$. The symbol $\Leftarrow$ gives the direction in the evaluation of the expression. The right-hand expression is the only part of the recursion equation that can be evaluated.

The set of rules for transforming recursion equations is described below. The rules are illustrated using the following example:

Given a scalar product function on vectors defined by

$x.y = \sum_{i=1}^{n} x_i y_i$

We want to compute $a.b + c.d$. Rewriting this in recursive function form we have:

$f(a,b,c,d,n) \Leftarrow dot(a,b,n) + dot(c,d,n)$

where

$dot(x,y,n) \Leftarrow$ if n=0 then 0 else $dot(x,y,n-1) + x[n]y[n]$ fi

where x[n] denotes a function which allows access to the components of the vector x and similarly y[n] denotes a function which allows access to the components of the vector y.

Rule: *definition* introduces a new recursion equation whose left-hand expression is not an instance of the left-hand expression of any previous equation. For example, we can add the new definition shown as follows:

$f(a,b,c,d,n) \Leftarrow dot(a,b,n) + dot(c,d,n)$

Rule: *instantiation* introduces a substitution instance of an existing equation. For example,

if $f(a,b,c,d,n) \Leftarrow dot(a,b,n) + dot(c,d,n)$ is an existing equation then an instance is:

$f(a,b,c,d,0) \Leftarrow dot(a,b,0) + dot(c,d,0)$

Rule: *unfolding* if $E \Leftarrow E'$ and $F \Leftarrow F'$ are equations and there is some occurrence in F' of an instance of E, then replace it by the corresponding instance of E' to obtain F'' and then add the equation $F \Leftarrow F''$. For instance, unfolding with

$dot(x,y,n+1) \Leftarrow dot(x,y,n) + x[n+1]y[n+1]$ $(E \Leftarrow E')$

and

$f(a,b,c,d,n+1) \Leftarrow dot(a,b,n+1) + dot(c,d,n+1)$ $(F \Leftarrow F')$

to

$f(a,b,c,d,n+1) \Leftarrow dot(a,b,n) + a[n+1]b[n+1] + dot(c,d,n) + c[n+1]d[n+1]\ (F \Leftarrow F'')$

Rule: *folding* if $E \Leftarrow E'$ and $F \Leftarrow F'$ are equations and there is some occurrence in F' of an instance of E', then replace it by the corresponding instance of E, to obtain F'' and finally add the equation $F \Leftarrow F''$. For example, if we have the following two recursion equations:

$f(a,b,c,d,n) \Leftarrow dot(a,b,n) + dot(c,d,n)\ (E \Leftarrow E')$

and

$f(a,b,c,d,n+1) \Leftarrow dot(a,b,n) + dot(c,d,n) + a[n+1]b[n+1] + c[n+1]d[n+1]\ (F \Leftarrow F')$

to

$f(a,b,c,d,n+1) \Leftarrow f(a,b,c,d,n) + a[n+1]b[n+1] + c[n+1]d[n+1]\ (F \Leftarrow F'')$

Rule: *abstraction* allows us to introduce a *where* clause derived from a previous equation $E \Leftarrow E'$. The new equation looks like:

$E \Leftarrow E'[u_1/F_1, \ldots, u_n/F_n]$ where $< u_1, \ldots, u_n > = < F_1, \ldots, F_n >$

An example of abstraction rule is shown in the example of trees defined below.

Laws are applied to transform the right-hand expression in an equation. The most used laws are associativity and commutativity obtaining a new equation for example the commutativity of the + operation allows us to rewrite the following equation

$f(a,b,c,d,n+1) \Leftarrow dot(a,b,n) + a[n+1]b[n+1] + dot(c,d,n) + c[n+1]d[n+1]\ (F \Leftarrow F'')$

as

$f(a,b,c,d,n+1) \Leftarrow dot(a,b,n) + dot(c,d,n) + a[n+1]b[n+1] + c[n+1]d[n+1]\ (F \Leftarrow F')$

Appendix I

Graph Terminology

The following set of definitions were taken from Horwitz [Horwitz *et al.* 88].

A *slice* G with respect to s written G/s is a graph containing all vertices on which s has a transitive flow or control dependence (i.e. all vertices that can reach s via flow or control edges)[Horwitz *et al.* 88]:

$V(G/s) = \{w | w \in V(G) \wedge w \rightarrow_{c,f} s\}$

where $V(G)$ is a set of vertices of a directed graph G (see definition later in this section).

The *program dependence graph* for a program P denoted G_p is a directed graph whose vertices are connected by several kinds of edges. The vertices of G_p represent the assignment statements and control predicates that occur in program P. In addition, G_p includes three categories of vertices that are defined below. The term *vertex* is used to refer to elements of dependence graphs.

1. There is a distinguished vertex called the *entry vertex.*

2. For each variable used in program P there is a vertex called the *initial definition of* x. The vertex is labelled as follows:

   $x := initialState(x)$ where ":=" is assignment in languages such as Pascal and Algol.

3. For each variable used in program P there is a vertex called *the final use of* x. The

vertex is labelled finalUse(x).

A *directed graph* G consists of a set of vertices $V(G)$ and a set of edges $E(G)$, where $E(V) \subseteq V(G) \times V(G)$. Each edge $(b, c) \in E(G)$ is directed from b to c. The term vertex is used to refer to elements of dependence graphs.

The edges of G_p represent dependencies between program components. An edge represents either a control dependence or data dependence. Control dependencies are labelled either true or false. A *control dependence edge* from vertex v_1 to vertex v_2 denoted by $v_1 \rightarrow_c v_2$ means that, during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge to v_2, then the program component represented by v_2 will be executed (although maybe not immediately).

A program dependence graph contains a *control dependence edge* from vertex v_1 to vertex v_2 of G_p if and only if one of the following conditions is satisfied:

1. v_1 is the entry vertex and v_2 represents a component of P (program) that is not subordinate to any control predicate. These edges are labelled *true*.

2. v_1 represents a control predicate and v_2 represents a component of P subordinate to the control construct represented by v_1. If v_1 is the predicate of a while-loop the edge $v_1 \rightarrow_c v_2$ is labelled as *true*. If v_1 is the predicate of a conditional statement $v_1 \rightarrow_c v_2$ is labelled *true* or *false* according to whether v_2 occurs in the *then* branch or the *else* branch.

Program dependence graphs contain two kinds of data dependence edges: *flow dependencies* and *def-order dependencies*. A flow dependence from vertex v_1 to vertex v_2 will be denoted by $v_1 \rightarrow_f v_2$. A program dependence graph contains a *flow dependence edge* from vertex v_1 to vertex v_2 if and only if all the following conditions hold:

1. v_1 is a vertex that defines variable x.

2. v_2 is a vertex that uses x.

3. Control can reach v_2 after v_1 via an execution path along which there is no intervening definition of x. This means that there is a path in the standard control flow graph for the program.

Flow dependencies can be classified as *loop independent* or *loop carried.* A flow dependence $v_1 \rightarrow_f v_2$ is *carried by loop* denoted by $v_1 \rightarrow_{lc(L)} v_2$ if the three previous conditions above are satisfied and the following conditions are also met:

4. There is an execution path that both satisfies the conditions in point 3 above and includes a back edge to the predicate of loop L.

5. Both v_1 and v_2 are enclosed in loop L.

A flow dependence $v_1 \rightarrow_f v_2$ is *loop independent* if in addition to 1,2 and 3 above, there is an execution path that satisfies 3 and includes no back edge to the predicate of a loop that encloses both v_1 and v_2.

A program dependence graph contains a *def-order dependence edge* from vertex v_1 to vertex v_2 labelled by $v_1 \rightarrow_{do(v3)} v_2$ if only if the following conditions are satisfied:

1. v_1 and v_2 both define the same variable.

2. v_1 and v_2 are in the same branch of any conditional statement that encloses both of them.

3. There exists a program component v_3 such that $v_1 \rightarrow_f v_3$ and $v_2 \rightarrow_f v_3$.

4. v_1 occurs to the left of v_2 in the program's abstract syntax tree.