COMPONENT LIBRARY RETRIEVAL USING PROPERTY MODELS

Ph.D. Thesis

University of Durham

Richard Morgan

February 1991

Abstract

The re-use of products such as code, specifications, design decisions and documentation has been proposed as a method for increasing software productivity and reliability. A major problem that has still to be adequately solved is the storage and retrieval of re-usable 'components'. Current methods, such as keyword retrieval and catalogues, rely on the use of names to describe components or categories. This is inadequate for all but a few well established components and categories; in the majority of cases names do not convey sufficient information on which to base a decision to retrieve.

One approach to this problem is to describe components using a formal specification. However this is impractical for two reasons; firstly, the limitations of theorem proving would severely restrict the complexity of components that could be retrieved and secondly the retrieval mechanism would need to have a method of retrieving components with 'similar' specifications.

This thesis proposes the use of formal 'property' models to represent the key functionality of components. Retrieval of components can then take place on the basis of a property model produced by the library's users. These models only describe the key properties of a component, thereby making the task of comparing properties feasible. Views are introduced as a method of relating similar, non identical property models, and the use of these views facilitates the re-use of components with similar properties. The language Miramod has been developed for the purpose of describing components, and a Miramod compiler and property prover which allow Miramod models to be compared for similarity, have been designed and implemented.

These tools have indicated that model based component library retrieval is feasible at relatively low levels of the programming process, and future work is suggested to extend the method to encompass earlier stages in the development of large systems.

Acknowledgements

I would like to thank Professor Keith Bennett for his excellent supervision and guidance throughout this research project.

Many thanks are also due to Roberto Garigliano, for introducing me to functional programming and providing me with much valuable help and advice.

This thesis has kept me away from home for many an evening and weekend and I thank my wife Jenny for her unselfish support as well as her help in preparing the manuscript.

Contents

Chapter 1

Introduction

The existing gap between the demand and our ability to produce high quality software cost-effectively calls for improvement of the software engineering process. The re-use of products from the software engineering processes has been identified as a potential method of increasing productivity and reliability of software. This stems from the observation that many software projects (development as well as maintenance) have strong similarities in the knowledge and methods applied to the project as well as in the products of the project[36, 4]. These similarities can occur at many levels of granularity, from the coding of small subroutines to the complete project level. The most obvious example of re-use is at the level of source code, where similar or identical modules may be used by entirely different systems. Rather than developing the module twice, once for each project, it should be possible to develop it only once and re-use it in subsequent projects. After the initial work of developing the module, two advantages are gained by re-using the module during subsequent projects; the work of re-specifying and implementing the module is saved and the reliability of the module is known from the fact that it has already been tested and used.

There are many possible products that can potentially be re-used. At the lowest level, code segments, sub systems and complete systems can be re-used as well as modules. At a higher level designs, prototypes, specifications, and requirements can be re-used. Usually the higher the level of product, the greater the potential savings to be made; for example, the discovery of a set of requirements that are close to the system that is needed might mean that the majority of the system is already implemented. To facilitate high levels of re-use there are several difficult problems to overcome.

Composition - Powerful methods for combining components are needed to increase the forms of components that can be combined and the ways in which they can be combined.

Extraction - Tools are needed to assist identification of the reusable parts of a system and aid their extraction from the system and conversion to a re-usable form.

Formalising Existing experience - Reusability can be increased if candidate objects for re-use are encoded according to a formal syntax and semantics. For example, design decision are conventionally recorded in natural language (if at all), but the use of a suitable syntax and semantics for such decisions might enable them to be re-used automatically or semi automatically.

Generation - Tools capable of applying reusable components such as design decisions to new situations must be available. Some such tools already exist (for example compilers and fourth generation languages); however, these currently work at the later stages of the software engineering process.

Management and Organisational issues - The non technical problems of software re-use are possibly the most serious. These include such problems as: motivation of staff to design components that are re-usable; the identification of costs when components are re-used between different projects and organisations; and the problems of ownership, particularly if a component is developed by one company as part of a system being built for another company.

Component Libraries - Methods of storing large collections of components must be available, as well as the tools for locating desired components efficiently. Due to the potentially vast number of reusable artifacts available these tools must be highly selective in what they retrieve.

The work of this thesis focuses on this last problem of storage and retrieval of components in a library. In particular, it provides and investigates a new method for describing components, so that they may be stored and retrieved effectively.

1.1 Property Models

When attempting to retrieve components from a library, the re-user must have some way of describing the properties of the components required. Such properties can be classified in two main groups: the functional properties which describe what a component does, and the environment properties which describe the environment of the component (eg. the component's source language). From the point of view of a person retrieving components from a library, the most important properties of any component are ones which describe its functionality. Unfortunately these are just the properties that are most difficult to describe satisfactorily. There are essentially two approaches to the problem. One is to use a large set of commonly used names that describe functionality; the other is to use a formal language that can accurately describe complex functional properties. The name based approach has several disadvantages: it is inadequate for describing complex functionalities and it hinders re-use of components between application domains because different names will be used for the same functionality — particularly for larger/ more abstract components. Although names are useful for extremely common functions, such as 'sort', the majority of component functions do not have a name that would be obvious to many of the library's users. The formal language approach overcomes these problems, but has disadvantages of its own: Firstly, the property description will be a partial or possibly complete formal specification of the component and could constitute a large effort to produce. Ideally we would like to re-use the specification rather than having to rebuild it from scratch. Another problem is that the formal specification can be too precise, preventing similar but non identical components from being retrieved. Finally the task of comparing specifications (to decide if a component meets its specification) is difficult to automate and *undecidable* in the general case. Hence the idea of using complete formal specifications for large components is infeasible.

This thesis suggests that an alternative to writing a complete property description is to produce a property model that describes only the essential features of the required functionality. In this case, each component in the library will be described by one or more such models, and retrieval takes place if any of these models match the one given by the library's user (hereafter referred to as the *re-user*). This means that the effort of producing the description of the required component can be greatly reduced. Another gain is that components similar to the ones required by the user can be retrieved. The problems of comparing specifications (which amounts to a theorem proving task) is aided by the reduced size of the specifications and the fact that knowledge of how to prove properties of a particular model can be stored along with the model.

It is likely that the model specified by the re-user and the models stored with a reusable component will differ in parts, particularly if the models are at different levels of abstraction (ie. contain more or less detail). To overcome this problem, a two way check can be performed and appropriate views are used to compare the model. This two way check means that the component is checked to see if it has the required properties, and if not a check is made to see if the components properties are all required. If either check succeeds then the component is potentially re-usable. This means that if the requirement model is less abstract than the components model (ie. contains more detail) then retrieval is still possible and is probably desirable, as the actual component will contain far more detail than its models. Views are used to provide a relationship between properties at different levels of abstraction. For example one model of an office might include desks as well as documents; however another model might not recognise the existence of desks. Part of the view between these models might state that removing a document from the office in the second model is equivalent to removing the document from all desks in the first model. Without taking this view into consideration, it might not be possible to show that the first model has the property of being able to remove a document from the office.

1.2 Scope of the thesis

The central aim of the research described in this thesis is to establish the feasibility of using property models to describe components so that they may be stored in, and retrieved from, component libraries. The work divides into four parts: the development of the property model method; the design of a language for writing component models; the design and implementation of tools for comparing the property models; and the assessment of the method through an experimental library and retrieval system.

During development the property model method for component library retrieval particular attention has been paid to the following objectives:

Re-use across application domains; this is important because it allows for a much greater level of re-use than is allowed by constraining re-use to take place within just one application domain.

Precise retrieval; keeping the ratio of wanted to unwanted components retrieved as high as possible to ensure that the re-user is not swamped with components that are of no use.

Complete recall; keeping the ratio of appropriate components retrieved to appropriate components not retrieved as high as possible.

The scope of this work has been limited to libraries of relatively low level components. It is not intended to demonstrate that the method can be successfully scaled up for large, high level components — such as the requirements of a complete system. However this is an issue for future research.

1.3 Overview of the thesis

The thesis is organised in the following manner.

Chapter 2 gives an overview of the software engineering process and describes how re-use fits into this process.

Chapter 3 provides an overview of current research in the area of software re-use. It describes the problems of component library retrieval and some of the existing solutions.

Chapter 4 describes the method of retrieving components by formal property models, the language Miramod which is used for writing property models and the method of comparing models using views.

Chapter 5 gives an introduction to theorem proving techniques and then describes the design of the property prover which is used for comparing property models.

Chapter 6 describes the design of a Miramod compiler which produces code in a form suitable for the property prover.

Chapter 7 describes the implementation of the property prover, compiler, and model comparison algorithm as well as the results obtained from the use of these tools.

Chapter 8 gives details of the experimental library and retrieval system, along with the results obtained from its use, including measures of retrieval, precision and completeness.

Chapter 9 summarises the thesis, discussing the extent to which the property model retrieval method meets its stated objectives and suggesting areas for further research.

Chapter 2

Software Engineering

Our current inability to produce correct software on time and within budget has frequently been referred to as the *software crisis.* This crisis has been observed and discussed in many reports, and this thesis will not attempt to demonstrate its existence, or set out to show that information technology is not achieving its potential for the solution of information problems[10].

The goals of reducing or solving the software crisis are central to the subject *software engineering.* Software Engineering can be defined as: the use of sound engineering principles, science and mathematics to produce software systems that are reliable, function on real machines and are useful to man.

For the large systems that are at the centre of the software crisis, a large team of software engineers is required to complete the project within a realistic time scale. With such large teams and/or large systems, it is not feasible to rely on the completeness and consistency of the various mental models of the system held by members of the team, so a common model based on sound engineering principles, science and mathematics must be shared by the project personnel and represented in a form that is accessible to them all.

2.1 The Software Engineering Process

One of the major contributions of Software Engineering is to identify important phases in the so called life-cycle of a software system, from its 'birth' as a concept to its 'death' as it is phased out. The phases, the order in which they are carried out and their interrelationships can be seen as a process or method for producing software systems. An alternative view is that they are a model of how software is produced (A life-cycle model).

The waterfall approach to software engineering[1] involves a well defined sequence of phases, each of which must be completed before the next phase is begun. Typically these phases are taken from:

Feasibility study The definition of a preferred concept for the software product, and determination of its feasibility and superiority to alternative concepts (frequently called *requirements analysis.*) This stage might well involve a cost/benefit analysis.

Requirements definition A complete specification of the required functions, interfaces, and performance of the software product. The specification should be checked with the systems users to ensure that it represents their requirements.

Product Design A complete specification of the overall hardware-software architecture, control structure, and data structure for the product along with other necessary components such as draft user's manuals and test plans.

Detailed Design A complete specification of the control structure, data structure, interface relations, key algorithms, and assumptions of each program component.

Coding A complete set of program components.

Integration A fully functioning operational hardware-software system, including such objectives as program and data conversion, installation, and training.

[1]often called the waterfall model of the software life-cycle

Maintenance A fully functioning update of the hardware-software system. This phase is repeated for each update.

Phase-out A clean transition of the functions performed by the product to its successors (if any).

At the end of each phase, the products of the phase must be checked against the products of earlier phases or checked with the systems users, to try to ensure that what has been produced is correct. This activity is referred to as *verification* or *validation*, where verification is an attempt to prove that the product of a phase meets its specification and validation is an attempt to establish the fitness or worth of a product for its operational mission. The two processes are characterised by the type of information that is being used as a reference for the check: in the case of verification, this information is relatively concrete and well defined (it might be the requirements specification for example); in the case of validation the information is abstract and loosely defined (the validation might take place with respect to the users 'concept' of the system.)

In the event of a verification or validation failure the phase must be re-done until it passes the appropriate test. If an error in a phase is not detected by the verification and validation at the completion of that phase but is detected during some later phase, then it is necessary to back track to the incorrect phase of the 'waterfall' and then redo the stages from the erroneous phase down. The early detection of errors though verification and validation therefore plays a crucial role in keeping the cost and completion time of a project to a minimum, since failure to detect an error during the early phases can lead to a great deal of wasted effort in later phases. As much as possible, iterations of earlier phase products are performed in the succeeding phase.

There are several major disadvantages with the waterfall model. The first is its strict sequencing of phases employed. Validation of the requirements is a difficult task because it can only be done through interaction with the customer, who will usually find it difficult to form an accurate picture of the system that is defined by the requirements. In practice this validation can only be done when some or all of the system is implemented, by which time a great deal of effort has been put into the design and coding of requirements that may not be correct. The strict sequence of phases utilised by the waterfall approach exasperate this problem by insisting that the whole system is subjected to each phase before any part of the system can enter the next phase of development. Validation of the requirements specification is also made difficult because in many cases there is a reverse dependency between the implementation of a system and its specification. For example, the system's code will be dependent on the requirements (as well as design decisions etc) but the requirement for a particular function may alter because of the efficiency with which the function can be implemented. Estimates of implementation efficiency can be made during the early phases but these are frequently inaccurate.

The incremental development method includes the same phases as the waterfall method but does not insist on the strict ordering of phases. Using the incremental development method, crucial parts of the system are implemented first and used to validate the specification and design. The rest of the system is built up in suitably sized increments until an implementation for the whole system is achieved.

Another disadvantage with the waterfall approach is that it does not include the development and use of scaffolding products. These are extra products that are developed to make the main job of software development and verification and validation go as smoothly and efficiently as possible. Examples of such products are:

- Dummy software components or stubs that can be used to help validate the high level architecture of a system before lower levels are completed.
- Miniature files or other simulated portions of the future operational environment.
- Test data generators.
- Postprocessors.
- Cross-reference generators.
- Conversion aids.
- Standards checkers
- Requirements and design language processors

The requirements definition, design, coding and testing of these products may occur at any point of the primary product's life cycle that is appropriate to the particular scaffolding product, in many cases preceding the coding of the primary product.

Although the general principle of the waterfall model is agreed upon, the naming and definition of the actual phases differ. For example, the feasibility study is often called a requirements analysis, requirements definition is often called specification, and testing is often identified as a separate phase rather than being considered as part of the verification and validation of the coding phase. For medium to small scale projects, fewer phases are separately identified because the verification and validation overheads associated with each phase are not so easily justified. In practice, the phases and their use will depend on many factors including: size of project; the application domain; procedures and practices of the developers; and the standards required by customers (users).

Software engineering methods such as the waterfall process are being increasingly used in industry, but they are not universally practiced. Code is often written with little or no requirements or design documentation and documentation for users and maintenance staff is written as an afterthought. Verification and validation methods are either not used or are used only after the code has been produced. Maintenance itself is performed directly on code and is made difficult by inadequate documentation of the software's function and design.

2.2 The Generalised Software Engineering Process and its support

The flaws in the waterfall process have led to the definition of a more general process which can be customised to suit particular application domains, organisations or projects. With the generalised approach, all phases are defined in a customised order depending on the type of system being produced or maintained. A phase can be missed out or repeated and maintenance is no longer seen as a separate phase but a repetition of some of the existing phases. With the generalised approach all phases are independently defined; each phase has well defined interfaces to other phases; selected phases can be arbitrarily interconnected with other phases that have matching interfaces and different phases can take place simultaneously on different parts of the system.

Efforts to support the software engineering process can be classified into two general approaches: the *technology fix* approach involves identifying enabling technologies and developing these; the *systems solution* concentrates on improving the organisation and structure of our knowledge, methods and technologies. The systems solution does not exclude technological innovation, which is necessary to facilitate the improved organisation and structure.

These alternative approaches are reflected in the different emphasis of research and development in major national and international programs. Japan's fifth generation program was designed to overcome the software crisis and give the Japanese a lead in some areas of information technology. It identified a number of enabling technologies such as: processing power; programming languages; natural language interfaces; and machine reasoning. The major efforts of this project have been directed towards improving these technologies, and hence the overall approach is one of technology fixes.

The UK based Alvey and European based ESPRIT programs were competitive responses to the Japanese fifth generation program. Along with their continuation programs IED (UK) and EUREKA (Europe) they have placed more emphasis on the systems solution and software engineering. Much of this work has concentrated on the concept of support environments for programming, software engineering and information technology based projects. This concept gained a prominent position in 1980 when Buxton's *Stoneman Report*[15] for the DoD introduced the Ada Programming Support Environment (APSE). This concept has since been expanded and generalised to give Integrated Project Support Environments (IPSEs) and the Information Systems Factory (ISF)[49].

An IPSE is intended to support the whole of the software engineering process by providing an integrated environment of tools that support phases of the process as well as the management of users and machine components. In [40] Mair lists desirable features of an IPSE as:

- Integrated, compatible tools environment
- Project control and management
- Configuration control
- Multi Language
- Software development support
- Hardware development support
- Distributed host/target

The Alvey program proposed three generations of IPSE:

1st generation - conventional operating systems , with tool builders allowed free use of information storage and user interface facilities; For example UNIX.

2nd generation - kernels as extensions to operating systems, together with standards (called public tool interfaces) to constrain tool builders' design of stored information and the user interface; the Stoneman and PCTE architectures are examples.

3rd generation - systems connection architectures , permitting (a) the free introduction of existing systems, (b) the design of information stores and user interfaces to match requirements. A requirement of 3rd generation IPSEs is that they should be knowledge based.

A further initiative of the Alvey program has been the extension of IPSEs to support hardware development as well as software, particularly allowing design down to the VLSI level as well as the program level. This has given rise to the concept of an Information Systems Factory (ISF). As the concept has matured, the definition has altered. The phrase *Computer Assistance for the Development of Information Systems* (CADIS) has been coined to describe the support system used by an organisation which develops information systems. An ISF is essentially an organisation that uses a CADIS system (though a more detailed definition is given shortly).

One of the major requirements of a CADIS system is that it should be capable of supporting the methods, practices and procedures of any organisation that is 'sufficiently mature'. In particular it should not constrain the organisation to adopt a particular life cycle model or particular tool sets. This gives rise to the concept of a CADIS architecture or framework into which an organisations existing tools and/or applications can be integrated to produce a CADIS virtual machine. This virtual machine would then be programmed to support the activities of the organisation thus producing the CADIS system that is actually used to develop information systems. The definition of an ISF therefore becomes:

> an **organisation** in which various **information systems** (tools and / or applications) are integrated within a **CADIS architecture** to form a **CADIS virtual machine**, which is programmed to support activities in the organisation.

(taken from the Information Systems Factory Study final report [49].)

The programming of the virtual machine for the support of activities is commonly called *process programming* and *process modelling.* A process program is a description of the relationships to be maintained between a set of interacting human and machine activities. As a result, process programming takes place at a higher level than conventional programming since it is concerned with the composition of activities to form methods as opposed to the composition of components to form programs.

2.3 Software Engineering Process Cost Analysis

Although the systems solution to software engineering provided by IPSEs and CADIS architectures may be crucial to the solution of the software crisis, to be useful they must be populated by suitable methods and tools. These methods and tools (which constitute the technology solution) are not only important in the context of IPSEs and CADIS architectures but are also useful and interesting in their own right.

Since many of the methods and tools of the software engineering process are distinguished by the phase of the life-cycle to which they apply, it is useful to examine the relative costs of different phases. This gives an indication of the potential savings to be made with a particular method or tool in relation to the costs of the overall life-cycle.

The results of estimates and surveys allocating costs to software life cycle phases are not easy to compare because their definitions of cost and life cycle phases are different. However, table 2.1 attempts to summarise these results in terms of the life cycle phases previously defined.

Phase \ Source	[63]	[37]
Feasibility Study	3%	46%
Requirements Definition	3%	
Product Design	5%	
Detailed Design		
Coding	15%	
Integration	7%	
Maintenance	67%	48%

Table 2.1: Software Engineering phase costs

It is clear from the above results that the majority of the cost of software originates from the maintenance phase. Coding is the second most costly phase; however it is only responsible for a small percentage of the overall cost.

Due to the lack of published results, there are several important dimensions of software cost that are not included in the above surveys. The proportion of cost arising from management and organisation of software projects is one example and the proportion of costs arising from verification and validation is another.

2.4 Cost Reduction

If substantial cost reduction of the overall software engineering process is to be achieved it is clear that it must be based on the reduction of maintenance costs or reductions in the cost of several phases. The use of IPSEs and CADIS architectures previously described should provide cost reductions to many or all phases of the software engineering process. Other approaches to cost reduction are described in the remainder of this chapter.

Firstly approaches to reduction in maintenance costs are covered, since this is the phase of the life-cycle in which cost reductions have the largest potential impact. The next section discusses specifications, both formal and informal. These are used mainly in the requirements phase, however they have a profound effect on later phases as errors that are introduced in the requirements phase give rise to increased costs in the latter phases. Formal methods also affect several phases of the software engineering process, however their main objective is to increase the reliability of software and the correspondence between earlier and later phase products of the process. Finally software re-use is presented as a major technology for decreasing costs in all phases.

2.5 Software Maintenance

Maintenance activities can be divided into three groups according to their cause [58]:

adaptive maintenance Performed to adapt software to changes in the hardware, software or data environments. Initiated by a change in the users environment or the computing environment.

corrective maintenance Performed to identify and correct software failures, performance failures, and implementation failures. Initiated by the discovery of a fault.

perfective maintenance Performed to enhance functionality, performance, cost-effectiveness, processing efficiency, or maintainability. Initiated by user requests.

Surveys indicating the relative costs of different maintenance activities have been carried out, and the results of some of these are summarised in table 2.2.

metric & source	adaptive	corrective	perfective
personnel hours [37]	18.2%	17.4%	60.3%
personnel hours [38]	23.6%	21.7%	51.3%

Table 2.2: Maintenance cost distribution

In [37] Lientz et.al. published results of a survey of 69 organisations. One of the interesting results of this survey is the ranking of problem areas by the organisations.

1. User demands for enhancements and extensions (perfective maintenance) was seen as the most serious problem area, the consensus of opinion rating it somewhere between a 'minor' problem and a 'somewhat major' problem.
2. Quality of system documentation was seen as a minor problem area.
3. Competing demands on maintenance personnel time were seen as a minor problem area.
4. Quality of original programs was seen as a minor problem area.

All other problems were rated less than minor in importance. Of these four problems 2 and 3 are clearly technical issues whereas problems 1 and 3 are management orientated problems. Lientz et.al. also carried out statistical tests on the complete set of results and concluded that the management areas were seen as significantly more problematic than the technical areas. However the actual difference is only small (5%).

In conclusion, these surveys indicate that to reduce software cost the management and technical problems of perfective maintenance must be tackled, and that the problems of adaptive and corrective maintenance are of lesser importance.

The following two sections describe methods of reducing costs in the problem areas that have been identified. One section covers methods which emphasise finding more cost-effective ways of performing the maintenance function. The other section describes methods based on designing maintainable systems. A combination of approaches is preferable, but the latter is of no value to maintainers of existing systems.

2.5.1 Performing Maintenance

From the definition of adaptive maintenance, it is clear that adaptive maintenance occurs as a result of the user's desire to change the system specification. There are three possible approaches to performing adaptive maintenance. The first is to modify and add code directly without reference to specification and design representations[2]. Although this approach may appear to involve less work it eventually leads to higher cost because without the appropriate documentation it is difficult to gain sufficient understanding of the code to maintain it successfully and avoid side effects to any modifications. It also makes future maintenance work more difficult as the design and specification will no longer fully correspond to the code. The second approach is to modify all of the products affected by the change, in which case verification must be performed to ensure consistency between specification design and code. The problem with this approach is that code modification is a non trivial task, even with the help of appropriate design and specification documents. The third approach is to modify the specification and then regenerate the design and code from the modified specification. The choice between the second and third approaches depends on the cost of regeneration compared to the cost of modification. In some cases regeneration is inexpensive (ie. regeneration of object code from source code by a compiler) and in other cases it is very expensive (ie. producing code from design). Advances in software production techniques, particularly automated or computer aided implementation and optimisation are liable to make regeneration a more viable option. Another approach is to use regeneration on parts of the system that are radically altered and modification on parts that remain the same or are only slightly different.

[2]eg documents and formal notations

There are a number of maintenance problems that derive from low quality or non existent development phase products. For instance poorly structured code, bad or non existent design documentation and poorly verified, badly documented or non existent specifications will all tend to increase the cost of any maintenance. The best solution to these problems is for the management to insist that these items are produced in the development phase and are of a sufficiently high standard. Unfortunately, for the majority of code currently in existence, this has not been done and retrospective solutions must be sought. One such solution is to totally rebuild the system using modern methods with maintainability as part of the new specification. This is an expensive option and can only be justified if the cost of future maintenance is likely to be large in comparison to the cost of rebuilding. Another solution, called inverse engineering, is to work backwards producing design and specification from the code. Work in this area is currently being undertaken at Durham[16]. The problem of maintaining low quality code that is badly structured can be solved by restructuring the code and some tools are available to help perform this task[17]. The layout of a program is also important if it is to be easily understood by a maintainer. Definition and implementation of company standards provide a maintainer with a layout that they will recognise and *pretty printers* are commonly available for automatically laying out code in a standard fashion. Cross referencers can be used to help a maintainer understand code and predict possible side effects of changes by providing information on the location and distribution of objects, such as variables and procedures, within a program as well as the structure of the program.

Inability to predict maintenance costs is a serious problem affecting maintenance management and decision making. Without the ability to accurately predict the cost of continuing to maintain a badly designed product or the cost of rebuilding the product, it is impossible to confidently decide on one of the alternatives.

Another maintenance management problem is that maintenance is seen as a non challenging and unskilled activity. As a result the personnel assigned to maintenance tasks are often the most junior, least skilled and lowest paid of those available. Motivation is often poor because they perform most of their work on low level objects (code), a large proportion of their effort is dedicated to understanding and discovering information rather than creative tasks and there are few career opportunities. Martin and McClure[42] identify several management decisions that can help to solve these problems. For example: establishing a separate maintenance staff with project leaders, managers etc can improve control, motivation and productivity. Another example is the selection of experienced staff for maintenance to ensure that the 'best' personnel are contributing to the most problematic and costly part of the software engineering process.

2.5.2 Designing for Maintenance

An alternative approach to many maintenance problems is to design and implement 'maintainable' systems. This approach is also important for software re-use, since increasing the maintainability of a system is also likely to increase its re-usability. Martin and McClure list the following properties of maintainable systems.

Understandability is defined as the ease with which we can understand the function of a system, how it achieves this function and how it was designed to achieve this function. Understandability is important in reducing adaptive, perfective and corrective maintenance costs.

Reliability is defined as the extent to which a program correctly performs its functions in a manner intended by the users as interpreted by its designers. A reliable system is correct, complete and consistent and also has low corrective maintenance costs.

Testability is defined as the ease with which program correctness can be demonstrated. Although this feature is important in the design phase, its importance is carried through into the maintenance phase as well. It is particularly helpful in reducing the cost of corrective maintenance.

Modifiability is defined as the ease with which a program can be changed. A modifiable program is general, flexible and simple. Generality allows a program to be used for a variety of changing functions without making modifications, while flexibility and small size allow a program to be modified easily thus reducing the cost of adaptive and perfective maintenance.

Portability is defined as the extent to which a program can be easily and effectively operated in a variety of computing environments. Building a highly portable system reduces the need for adaptive maintenance.

Efficiency is defined as the extent to which a program performs its intended functions without wasting machine resources such as memory, mass storage utilisation, channel capacity, processor capacity and execution time. If a program with a high standard of efficiency is produced it is less likely to need perfective maintenance requiring greater efficiency.

Usability is defined as the extent to which a program is convenient, practical and easy to use. This is another feature which reduces the possibility of users requesting adaptive maintenance.

Unfortunately some of these features of a maintainable system are (at least superficially) contradictory. For example, efficiency is usually achieved at the expense of modifiability, understandability, portability and reliability. The only way of achieving these goals simultaneously is to generate efficient code automatically or semi automatically from the high level code. Thus the high level code can be used for understanding and modification while the low level code is used for efficient implementation. However, in the foreseeable future there will always be a tradeoff to be made due to the gap between our ability to formally describe what must be done and our ability to automatically implement these descriptions efficiently [7].

There are several basic methods of improving maintainability:

Setting explicit software quality objectives and priorities.

For a maintainable system to be produced, management must explicitly ask for the system to have appropriate maintainability. Specific objectives must be set and monitored throughout the systems development.

Use of modularisation.

Modularisation is a construction technique that uses a set of conceptually and operationally independent pieces to make up a system. Modules improve the modifiability of a system by hiding their own implementation from other parts of the system so that if the module must be re-implemented due to some perfective or adaptive maintenance, the rest of the system need not be affected. Modules can also improve understandability and testability because they can be viewed and tested independently from the rest of the system.

Use of structured techniques.

Structured techniques define how modules can be interconnected. They vary according to whether or not the language is procedural. For a procedural language constructs are restricted to concatenation, selection and repetition. Generally, abstract constructions are favoured over low level ones (such as goto). Variables should also serve only one program purpose each and their scope should be apparent and limited. These techniques make a system easier to understand and modify.

Use of appropriate languages.

Programming languages define the level of abstraction at which a system is described and also encourage or discourage the use of modular and structured techniques. Abstract languages lead to shorter more concise programs that are easier to understand and modify. Increasing the level of abstraction in data types also increases the portability of a system.

Production and storage of development phase products.

Development phase products such as the design document and specification crucially affect the maintainability of the system. The specification is of particular importance for corrective maintenance as it defines how the system should behave. A record of the design decisions made is important for both perfective and adaptive maintenance. An explicit record of design decisions allows those decisions affected by maintenance to be identified and altered accordingly. An important issue is the method of describing and storing design decisions. A recent approach is to record the design as a series of transformations (see Section 3.1.2), however the transformations used by current systems are not sufficiently abstract or powerful to constitute design decisions.

2.6 Specifications

The aim of a specification is to provide a precise, clear and consistent description of the problem that is to be solved, without stating how it should be solved. Although specifications may be produced at many stages of the software life cycle, the main use of specifications is to record the results of the requirements analysis phase.

The practice of specifying software before designing and implementing it has many advantages. One is that the specification can be verified with the users of the system before any effort is wasted on designing and implementing a solution to the wrong problem. Another is that a requirements specification also ensures that the system's designers and implementors have a specific target to aim at — they can assess their work in terms of the requirements they are given.

The specifications or requirements of a system can be expressed in a formal or informal language. Informal languages include natural language, subsets of natural language and diagrams (though diagrams may also be formal — for example syntax diagrams). They have the disadvantage of allowing the specification to be inconsistent and incomplete, however they are more powerful than formal languages and can be understood by the system's users as well as the software engineers. Formal languages make the detection of inconsistencies and incompleteness easier but they are far harder to produce and understand than informal specifications. Formal specifications are also harder to validate since the users are unlikely to be able to understand them. However they are more suited to the rigorous or formal verification of design and code. Some formal specification languages have had paraphrasers developed for them [7]. These take the specification and produce a stylised natural language description of the system. Though these are often an improvement on the formal specification they are still difficult to understand.

The emphasis on specifications, whether formal or informal, is to describe what the system should do rather than how it should be done. As a result specification languages are very different from implementation languages which describe how tasks should be performed.

A small and simple example of a specification is provided by the square root function:

'sqrt' is a function that takes a positive number as an argument and returns another positive number which, if squared will produce the value of the original argument to 'sqrt'.

The formal specification of this function can be written:

```
sqrt x >= 0 and abs (sqrt(x)² - x) < e
```

(where 'e' is a small positive number which indicates the minimum acceptable accuracy of the result and abs is the function which gives the absolute positive value of a number).

The above specification has the important property that it does not suggest a method of calculating the value of the function.

2.7 Formal Methods

Formal methods are methods based entirely on the form of objects rather than their intended meaning. Two common uses for formal methods in the software engineering process are the production and verification of programs from specifications [32, 41]. In the case of the verification of a program from its specification, the formal method could provide a number of steps (which may be automated as transformations - see 3.1.2) that are guaranteed to produce true statements about the program. The software developer can then apply these steps to the program to produce the original specification. If successful, then they have proved that the program meets its specification.

Formal methods can be applied either manually, automatically or semi-automatically. With the manual approach, the user must apply each step himself, recording the result before continuing with the next step. The semi-automatic approach lets the computer apply the individual steps; however the user must still decide the order in which steps must be applied as well as the particular part of the problem to which they apply. The automatic approach suggests that all the steps from starting point to goal are chosen and applied automatically by the computer.

Unfortunately, formal methods are too clumsy to be used in large scale developments. Single steps which an experienced programmer or system designer might see as obviously correct, can take a large number of formal steps which would be too time consuming to specify explicitly. An alternative is to use rigorous methods in which every step need not be precisely defined by the method, but the person responsible for generating that step should be confident that they could derive it from a sequence of formal steps if necessary. Rigorous methods do not guarantee success, because it is possible for an 'obviously correct' step to be incorrect.

Advances in artificial intelligence are making it increasingly possible to automatically generate the formal steps from the rigorous steps, thus guaranteeing success (subject to the correctness of the method), without involving the user of the method in too many low level steps.

2.8 Re-use

One method for achieving the goals of decreased cost and increased quality in initial development or maintenance is broadly termed re-use. This involves using any products of previous projects in the current project. The word 'products' should be taken to mean anything that may be subsequently useful, including complete systems, system components, design strategies, specifications, requirements and the results of domain analysis. There are two major advantages gained by re-using software. One is the decrease in software cost, achieved because the cost of the product that is re-used can be spread over the many projects in which it is used. The other is the improvement in reliability due to the increased testing performed on the re-used products (presumably the re-used products will be tested either directly or indirectly each time it is re-used).

The most common form of re-use practiced today is human based. Algorithms are either stored mentally or in books in an abstract form, so that the programmer adapts and instantiates the algorithm according to its current usage. Complete programs or sections of code which are actually stored on the development system are often re-used in a similar way with the programmer physically copying and altering the code. This is one of the motivations behind making the operating system sources available on UNIX systems [33].

Chapter 3

Software Re-use : An Overview

To survey the current work being done on re-usability it is helpful to draw up a framework of different approaches to software re-use (see table 3.1). The framework initially divides into two different approaches, namely the building block approach and the generator approach. In the building block approach, re-use is achieved via the storage and use of passive objects such as procedures, functions, code skeletons, programs, specifications, and design or requirements information. In the generator approach re-use is automatically or semi-automatically achieved by some active agent. This could be a high level language compiler, an applications generator or a transformation system. In the generator approach, re-use is less a matter of manipulating components than one of executing them[8]. An important dimension of software re-use not explicitly recognised in this framework is the level of abstraction involved. Most of the approaches can be targeted at many different levels of abstraction, from code at the low level to requirements specifications and designs at a higher level. The majority of the current research into software re-use is targeted at relatively low levels.

Approaches	Research Areas	Examples
Building Blocks	Composition and adaptation principles	Object oriented programming Higher order functions Abstract data types
	Storage and retrieval	Component catalogues Keyword retrieval Software function frames [60]
Generators	Language based systems	Compilers Applications generators Draco [48]
	Transformation systems	Transformational implementation [6]

Table 3.1: A framework for software re-use

3.1 Generator approach

The generator approach to re-use is based on active agents which generate the desired behaviour or code. The re-use occurs through execution rather than the manipulation of the generator, and the re-usable patterns or components are contained within the generator itself. A characteristic of generator based re-use is that it is hard to identify re-used patterns or components in the generators output since they are usually more global and diffuse. Generators can be grouped into language based systems, application based systems and transformation systems. Generator based methods of re-use differ according to the extent which the user is involved. For example a compiler will generate code without any help from the user; however a transformation system might ask the user which transformation to apply next. In both cases, however the generator itself actively performs the re-use.

3.1.1 Language based systems

In language based systems, the emphasis is on the notation used to describe the desired behaviour. Included in this category are compilers and interpreters of general purpose as well as domain specific languages. Most language based systems apply their re-use automatically, either producing code or the desired behaviour directly. The re-use achieved by these systems is considerable if measured in terms of the object code generated; however its level of sophistication is relatively low in that there is a close correspondence between the language given to the generator and the generator's output. General purpose systems achieve only limited re-use but cover a wide range of applications. On the other hand, domain specific systems provide a much higher level of re-use but are limited in their application. Another advantage of domain specific systems is that the language may employ terminology and notations specific to the domain, making it easier for experts in the domain to achieve re-use . For example the UNIX tool "yacc", which is used to generate compilers and parsers, has an input language which is very similar to a notation commonly used for describing programming languages: BNF (Backus-Naur Form).

3.1.2 Transformation Systems

Before examining the possibilities for re-use provided by transformation systems, it is necessary to define in a general way what is meant by a transformation system and some of the related terms. The following is an abbreviated version of the definitions given by Partsch [50].

A program scheme is a representation of a class of related programs and is derived from those programs by parameterisation. A transformation is a relation between two program schemes. A transformation is said to be valid if a certain semantic relation holds between the two program schemes. This relation is usually either equivalence, weak equivalence or descendance. Weak equivalence ignores undefined situations whereas true equivalence does not. A descendant relation occurs with nondeterminate program schemes when the possible results of one scheme are a subset of the possible results of the other. In this case the relation holds only over a particular order of mapping. Transformation rules are representations of transformations and are either procedural or related schemes. Conditions are often attached to these rules so that they are only valid in certain situations. Transformational programming is a method of producing programs from other programs such that a certain semantic relation holds. The relation is usually equivalence, the original program some form of specification and the produced program an implementation of the specification. A transformation system is an implemented system for supporting transformational programming.

It is not intended to provide a complete survey of transformation systems in this thesis[1]. Instead the features of transformation systems that involve or promote re-use in some form are discussed and examples given.

Transformation systems are usually aimed at either implementation or optimisation. Implementation is achieved by applying sequences of transformations to produce procedural constructs from specification constructs and data type implementations from abstract data types, in effect removing specificational freedoms. Optimisation is usually achieved by rearranging rather than removing language constructs. The type and quantity of transformations available vary considerably. Some systems use catalogues of transformations [6, 56, 48] while other systems provide small but powerful transformations that can be combined to generate larger transformations [20]. In both these cases the information provided by the transformations themselves is being re-used. Further re-usability is achieved by the storage of sequences of transformations used to implement a specification. In effect this information can constitute the design decisions made to implement a system and is therefore potentially re-usable as it stands or in modified form. This opens up the possibility that if the original specification is modified or re-used in a different environment, the previous design can be replayed to create a new implementation. Although this technique sounds promising it is hampered by the fact that the choice of transformations represents a very low level of decision making. As a result of his early work on 'Transformational Implementation' [6] Balzer stated that:

"Instead of a concern for maintaining consistency, the equally consuming task of directing the low level development has been imposed. While the correctness of the program is no longer an issue, keeping track of where one is in a development and how to accomplish each step in all its fine detail diverts attention from the tradeoff question."

This not only means that the selection of transformations constitutes a great deal of work, but it also narrows the re-usable potential of those decisions because they are vulnerable to changes in the specification.

One solution to this problem that has been proposed is to raise the level at which

[1]See [50] for a survey of transformation systems

decisions are made by generating many of the choices of transformation automatically [23]. If this is done the human developer need only guide the development by providing high level goals to a problem solver which then finds the appropriate subgoals or transformations that will achieve the goal. The high level goals can then be made available for re-use, and due to their abstract nature their potential is far greater.

The re-use achieved by the Draco system [48] could be classified both as language based and as transformation based. Prior to producing software in some domain using the Draco system, a domain analysis must be performed and it is the products of this analysis that are re-used when producing the software. The products of a domain analysis are:

Domain language The domain language is a language whose objects and operations represent analysis information about the problem domain. Analysis information is being re-used every time a new problem is expressed in the domain language.

Parser The parser for a particular domain language has two functions. It acts as a verification tool for problems expressed in the domain language and it converts instances of the domain language into a tree form which is easily manipulated by other parts of the system.

Pretty-printer This pretty-printer is a de-parser which allows parts of the internal form of the problem description to be converted to the external form for interaction with people in the language domain.

Transformations The domain transformations are source to source transformations within the domain language. They represent the rules of exchange between objects and operations of the domain and can be used to make optimisations at a high level of abstraction.

Components Each software component provides one or more implementations of an object or operation in the domain. These implementations are called refinements and may be in terms of the original domain language or some new domain language. The components of a domain define the semantics of a domain language in terms of other domains which can be used to implement programs described in the component's domain language.

Procedures Procedures are used to describe domain transformations which are algorithmic by nature.

Once the appropriate domain analysis has been performed, software within that domain is specified using the domain language. The Draco system then uses the parser to check the specification and the catalogue of transformations to annotate the specification with possible transformations which the designer can select. The designer also controls the refinement process, repetitively restating the problem originally specified in other domains known to Draco by selecting appropriate refinements of objects and operations. Eventually the level of abstraction of the developing program must decrease to an executable language domain. The choice of refinements presented to the designer may be reduced because each refinement records its implementation decisions explicitly so that refinements that contradict any previous implementation decisions can be excluded. The designer also has the responsibility of deciding what kind of modular structure is produced from each refinement. If the refinement is instantiated 'in line', the refinement is expanded in the developing program as a macro (with renamed variables etc). If the refinement is instantiated as a 'function' then the appropriate function call is inserted in the program and the function definition provided if it is not already present. Alternatively a 'partial' instantiation is possible, whereby some parameters are instantiated in line and others are passed. By using default refinement tactics provided by the domain analysis, software can be developed automatically. It will tend to be highly inefficient because of the lack of optimisation and the use of inappropriate implementation decisions, however it can be used as a rapid prototype for validation of the original specification.

The major criticisms of the Draco system are that it is incapable of applying re-use across domains, the domain analysis is an extremely costly process and is only feasible in domains that are already well understood and it is of little use for designing systems in new application domains.

Both of the transformation systems described above are based on catalogues of transformations. The work by Darlington [20], however, is based on four transformation rules: definition; instantiation; unfolding; and folding. The system is designed to optimise recursion equations in the NPL language. The syntax of the main part of NPL is:

Primitive functions A primitive function symbol is denoted by a lower case identifier and may have zero or more arguments.

Variables A variable is written as an upper case identifier.

Recursive functions A recursive function is denoted by a lower case identifier and may have zero or more arguments.

Expressions An expression is built in the usual way out of primitive function symbols, parameter variables and recursive function symbols.

Left hand expressions A left hand expression is of the form $f(e_1, \ldots, e_n)$, where $n > 0$ and $e_1, \ldots, e_n$ are expressions involving only parameter variables and primitive function symbols.

Right hand expressions A right hand expression is an expression, or a list of expressions, qualified by conditions.

Recursion equations A recursion equation is written $E \Leftarrow F$, which means that E is defined as F.

Thus a NPL program is a list of equations such as:

$$\begin{array}{rcll} \text{append(nil,Y)} & \Leftarrow & \text{Y} & (1) \\ \text{append(A :: X,Y)} & \Leftarrow & \text{A :: append(X,Y)} & (2) \\ \text{g(X,Y,Z)} & \Leftarrow & \text{append(append(X,Y),Z)} & (3) \end{array}$$

This example uses the NPL list notation, which writes 'nil' for the empty list and '::' for the list construction operator. Hence the function 'append' joins two lists and the function 'g' joins three lists. The transformation rules operate on the equations and produce new equations. They are:

Definition Introduce a new recursion equation whose left hand expression is not an instance of the left hand expression of any previous equation.

Instantiation Introduce a substitution instance of an existing equation. For example 'g(X,Y,Z)' can be instantiated to 'g(nil,X,Y)' and 'g(A::X,Y,Z)'.

Unfolding If $E \Leftarrow E'$ and $F \Leftarrow F'$ are equations and there is some occurrence in F' of an instance of E, replace it by the corresponding instance of E' obtaining F''; then replace $F \Leftarrow F'$ with $F \Leftarrow F''$.

Folding If $E \Leftarrow E'$ and $F \Leftarrow F'$ are equations and there is some occurrence in F' of an instance of E', replace it by the corresponding instance of E obtaining F''; then replace $F \Leftarrow F'$ with $F \Leftarrow F''$.

With reference to the example NPL program, although the function 'g' is conveniently defined in terms of the append function, its execution is inefficient as it builds up an intermediate list along which iterates. The user can initiate an attempt to optimise the function 'g' by supplying the system with instantiations of one or more of the functions parameters, for example: g(nil,Y,Z) and g(x :: X,Y,Z). Working with one instantiation at a time, the **instantiation** transformation is used to produce a new version of the equation defining g.

$$\begin{array}{rcll} \text{g(nil,Y,Z)} & \Leftarrow & \text{append(append(nil,Y),Z)} & (3) \\ \text{g(x :: X,Y,Z)} & \Leftarrow & \text{append(append(x :: X,Y),Z)} & (4) \end{array}$$

For every possible combination of unfoldings in the new equation, an attempt is made to find any foldings that meet certain criteria (eg they do not produce non terminating recursions). If such a folding is found the result is presented to the user who can either reject the new equation, in which case the search is continued, or accept the new equation, optionally allowing the search to continue. Taking the first instantiation (3) the only possible unfolding comes from equation (1) giving:

$$\text{g(nil,Y,Z)} \Leftarrow \text{append(Y,Z)}$$

For the second instantiation (4) there are two possible unfoldings. The first is:

$$g(x :: X, Y, Z) \Leftarrow \text{append}(x :: \text{append}(X,Y),Z)$$

however there is no acceptable folding possible (the only possible folding is the exact opposite of the folding that has just been performed). The second possible unfolding is:

$$g(x :: X, Y, Z) \Leftarrow x :: \text{append}(\text{append}(X,Y),Z)$$

This can be folded using equation (3) and the result is:

$$g(x :: X, Y, Z) \Leftarrow x :: g(X,Y,Z)$$

Providing that these two equations are accepted by the user the transformed program becomes:

$$\begin{aligned} \text{append}(\text{nil},Y) &\Leftarrow Y \\ \text{append}(x :: X,Y) &\Leftarrow x :: \text{append}(X,Y) \\ g(\text{nil},Y,Z) &\Leftarrow \text{append}(Y,Z) \\ g(x :: X,Y,Z) &\Leftarrow x :: g(X,Y,Z) \end{aligned}$$

The new version of ‘g’ defined by these equations is more efficient than the original definition of ‘g’ because the first list argument is only traversed once (by ‘g’) rather than twice (once for each ‘append’ in the original definition).

Many optimisations can only be achieved by the introduction of an appropriate auxiliary function definition. In [14] Burstall and Darlington describe the definition of these functions as ‘eurekas’ and show how they can be generated from the equation that is being optimised. Darlington also describes additional logic constructs and transformations on these constructs which can be used to synthesise programs from specifications. The two major criticisms of this work are the degree of assistance required from the user and the small scale of the problems for which it has been shown to work. Some user interaction could be eliminated by storing appropriate base cases for data types and using the typing information of NPL to provide a choice of appropriate instantiations. For example, some apropriate instantiations for lists are: (nil), (A::X), (A::B::X), etc.

If the problems of scale can be overcome , this work has two important results for re-usability . Firstly it is capable of re-using information about programs (in the form of transformations) to implement and optimise specifications. Secondly it helps to remove the criticism that highly generalised specification and program components produce inefficient code, because it is capable of removing these inefficiencies.

3.2 Re-use based on passive objects (Building Blocks)

The passive object approach to re-usability addresses two groups of problems. The first concerns the maximisation of re-usable potential with composition, generalisation and adaptation principles and the second deals with the methods of storage and retrieval of appropriate 'building blocks'. For large scale re-use to be achieved both problems must be tackled. The ability to find appropriate re-usable objects is essential if there are a large number of possible objects to be re-used and the complexity of modern software makes it extremely unlikely that appropriate software will already exist so the generalisation, combination and adaptation of existing components is necessary. These problems can apply to active components (ie. generators) as well as passive ones, though they are more apparent for large numbers of small generators (such as transformations) than small numbers of large generators (such as compilers).

3.2.1 Maximising component re-usability

Researchers in this area place different emphasis on the alternatives of component adaptation and generalisation as well as the problem of component composition. Adaptation techniques are based on the view that in many cases, existing components will not be entirely suitable for the current task but the work in modifying the component will be less than the work of re-building it. An alternative approach is to attempt to generalise the component to the extent that it can be re-used without alteration. Composition principles provide for the construction of the desired component from a number of smaller components. Composition principles are usually determined by the form of the objects being composed. Two common levels for composition are program level and sub program level.

At the program level, complete programs must be combined, and composition techniques must provide for the transfer of data and control between the programs. As with most forms of composition, standardisation of interfaces is important as it increases the number of programs that can be successfully combined. Most operating systems provide job control languages, which can be used to compose tools by joining together complete programs; the 'shell' language available on UNIX is an example [33]. The principal mechanism for composition in this language is the pipe (represented by a vertical bar '|') which connects the output of one program to the input of another. Reusable programs are then written as filters which perform some action on the data passed through it. The success of this method is dependent on each program producing its output in a straightforward fashion and the use of text as a standard representation. For example, suppose it is necessary to know how many times Fred is currently logged on and the following three utilities are available: `who` outputs a line of information (including user name) for each user that is currently logged on, `grep` prints each line of its input that matches a pattern given as a command argument and `lc` counts the number of lines on its standard input. The command that will answer our question is:

```
who | grep Fred | lc
```

At the sub program level construction from components is a key feature of all modern programming languages. These provide constructs (such as procedure, function and module definitions) for isolating logical units of a program which can then be re-used throughout the program. The fact that these components can be used in place of the basic units of the language, means that they can easily be combined to create many different behaviours. Although this is very effective at promoting low level re-use it fails at the higher level because there is no mechanism for changing lower level details when re-using high level structure. A small example is the re-use of a sorting algorithm: one of the many 'low level' details that we may wish to change is the method of comparing the elements that are being sorted - we might wish to sort on several different keys and to several different orders. Ideally we would like a simple method of defining the 'generic' sorting component and then combining this with a component which defines an ordering over which the sort should take place. A somewhat unsatisfactory solution to this problem at the level of COBOL source code is described by R G Lanergan and C A Grasso [36]. They recognise the need for both re-use by composition from unchanged functional modules and instantiation of 'program logic structures' which are essentially basic program structures with empty lower levels. They describe the re-use practiced at Raytheon's Missile Systems Division, Information Processing Systems Organisation. This is based on a library of approximately 3200 COBOL modules which is organised into eight categories and a small collection of COBOL program logic structures. They report that the results of a classification exercise on existing software, developed without program logic structures at Raytheon, suggested that the principal program logic structures (edit, update and report) where applicable in the following proportions.

- edit 20%
- update 20%
- report 45%

The use of both the programming logic structures and re-usable functional modules was found to give an average of 60% code re-use, but the major gain was found in the area of maintenance. The use of logic structures means that the maintenance programmer is already familiar with the structure and much of the code of the system. The unsatisfactory points in this approach are the level of abstraction at which it is based (COBOL source code), the narrow domain in which the re-use is possible and the informal way in which the program logic structures are instantiated and modified.

Many languages provide forms of these parameterised program structures. For example, imperative languages such as Pascal, allow procedures and functions to occur as arguments to other procedures and functions. A procedure which takes one or more procedures or functions as arguments can be seen as a program logic structure, with the parameter providing a slot for lower level details to be filled in. Functional programming languages are particularly suited to this form of re-use since higher order functions fit very naturally into the languages framework, and functions are often treated in exactly the same way as data. In this context, powerful higher order functions and data structures may be created and used to combine other functions and data structures - these higher order functions take the place of Lanergan's program logic structures. Hughes [30] describes two new kinds of 'glue' for combining components that are provided by some functional programming languages. The first of these is the use of higher order functions and the second is the use of lazy evaluation.

An example of a commonly used recursive structure is that of folding up a list using an operator and an identity element. Using the notation of Bird and Wadler [9]: a list with n elements e_1 .. e_n is denoted `[`e_1`,`e_2`, .. ,`e_n`]`, the empty list is denoted `[]` and the list `x` with the element `a` added to the front is denoted `(a:x)`. Thus if we fold the list `[`e_1`,`e_2`, .. ,`e_n`]` with operator $\oplus$ and identity element `i` the result is $e_1 \oplus e_2 \oplus \ldots \oplus e_n \oplus$ `i`. The ability to define higher order functions means that we can define a general purpose folding function **`fold`** as follows (function application is denoted by juxtaposition):

```
fold (⊕) i [] = i
fold (⊕) i (a:x) = a ⊕ (fold (⊕) i x)
```

In this definition the braces enclosing the operator are necessary to show that the operator is not being applied to any arguments but is simply being passed as a parameter (ie. '`fold (⊕) i`' is not the application of operator $\oplus$ to arguments `fold` and `i`, but the application of function `fold` to the operator $\oplus$ and identity element i.)

The function fold is a highly re-usable combinator that can put together many binary operators and identity elements to form a family of folding functions:

```
sum x = fold (+) 0 x
```

'sum' finds the sum of a list of numbers;

```
product x = fold (*) 1 x
```

'product' finds the product of a list of numbers;

```
and x = fold (&) True x
```

'and' finds the conjunction of a list of boolean values ('&' denotes the conjunction operator);

```
concat x = fold (++) [] x
```

and finally 'concat' is the list concatenation function which produces a list of elements from a list of lists of elements by appending together the inner lists ('++' denotes the list append operator).

The second 'new' type of glue provided by some functional languages is lazy evaluation. Given two functional programs 'f' and 'g' where the type of g's output is compatible with the type of f's input, a new program can be created using the function composition operator '.'. This creates a 'pipelined' program which applies the second argument and then the first argument in turn:

```
(f .  g) i = f (g i)
```

In the case of most programming and job control languages, such a composition would mean that the whole result of 'g' must be produced before f can begin processing. However, using lazy evaluation, 'g' only produces output when asked to do so by f, and even then, it only produces as much as is required by 'f'. Also the application of function 'g' to the input will terminate as soon as the enclosing application of 'f' terminates. This means that g can be defined to produce infinite output (as it will be forced to terminate when 'f' terminates), thus allowing loop bodies (g) to be separated from termination conditions (f) and both to be re-used independently. The data passed between programs this way is not limited to any particular data type since the lazy evaluation mechanism includes the use of any type, including structured ones. For example 'g' could construct an infinite tree and 'g' could search the tree for a termination condition. The lazy evaluation performed by many functional languages is more general than the UNIX pipe mentioned previously, since the UNIX pipe only provides for lists of characters.

Another important re-usability mechanism provided by many functional languages parameterised (or polymorphic[44]) types. Modern functional languages such as Miranda [2][59] are usually strongly typed - that is each expression and each variable has a type that can be deduced by a static analysis of the program text, and any inconsistencies in the type structure result in a compile time error message. The use of strong typing is accepted as an important software engineering technique and an essential part of modern programming languages, as it frequently leads to logical or typographical errors in source code being detected at compile time. Unfortunately, strong typing can act as a major block to component re-use. For example, there are many different types of list generated from the many possible element types for lists. However, there are also many list processing functions that are entirely independent of the element type (the 'head' and 'tail' of a list as well as its length are examples of values that can be computed without any information about the type of the list elements). Without the possibility of parameterising the type 'list' with its element type, we must define new functions for each different list type and effectively resort to re-use by modification rather than generalisation.

Polymorphic types are types with parameters, or "many forms". A language which allows modules, procedures or functions to be defined over polymorphic

[2]Miranda is a trademark of Research Software Ltd.

types allows important generalisations of components that are not provided my monomorphically typed languages. To continue the list example, we can have a polymorphic type 'list of *' where *, ** etc. are used as generic type variables. We can then define various functions over this polymorphic type: 'length' might take a 'list of *' as parameter and return a number denoting the length of the list; 'head' might take a 'list of *' and return the '*' which is the first element of the list; 'tail' might take a 'list of *' and return the 'list of *' that is the remainder of the list with the head removed. Since these functions all have a well defined (polymorphic) type, they can be legally declared and re-used without modification. Any particular re-use of these components may be a specialisation in which the type variable becomes instantiated or a generic re-use in which the type variable remains free (in which case the component it is used to construct will be polymorphic). This contrasts strongly with monomorphically typed languages such as Pascal, where each polymorphic use of the abstract functions would require a modified function definition. Although re-use might still be possible using modification techniques or pre-processors, these are likely to involve more effort in achieving the re-use and will also create larger object code.

An increasingly popular approach to the construction of software from building blocks is the object orientated approach. An object is an entity which consists of a state and a set of operations (or functions) which access and modify the state. The object oriented approach essentially involves the construction of systems as a set of objects which communicate by passing messages. One construction mechanism that is of particular importance to software re-use is inheritance. This allows an object to 'inherit' the behaviour of other objects and a particular object may be inherited by many other objects. Possibilities for re-use can be increased if selected parts of an object can be inherited without change while other parts are redefined by the inheriting object without changing the inherited object. Such an object can be described as the difference between what already exists and what is required. In [19] Curry and Ayers describe three construction mechanisms: extension, variation and union. Extension allows the addition of state and operations to the object and is necessary when the inherited object is correct but insufficient. Variation allows the definition of operations to be changed when either the functionality or the implementation is not quite that which is required. Union is a form of extension achieved by combining the functionality of several inherited objects.

The inheritance mechanisms described above are a more powerful version of the procedure, function and class mechanisms provided in high level languages. Although these are important they do not support re-use at high levels of abstraction. They allow for the re-use of objects whose top level structure must be altered or extended but whose lower levels may remain the same. A mechanism is also needed whereby the top level structure remains the same but the lower level structures can be replaced by suitable alternatives. Parameterisation is such a mechanism and allows instances of parameterised objects to inherit different objects for different purposes, thus allowing the re-use of parameterised objects as well as inherited objects. Since the parameters of an object may define types as well as functions, this mechanism takes the place of both polymorphism and higher order functions used in functional languages. If the techniques of parameterisation are sufficiently powerful then parameterised objects should be capable of taking the place of the program logic frames previously described. A simple example of a parameterised object and its re-use is given by Goguen84 [25].

" As an example of parameterised programming, consider a parameterised module LEX[X] which provides a lexicographic ordering on lists of X's where the parameter X can be instantiated to any set with a designated ordering relation. Thus, if ID is a module that provides identifiers (and in particular, words) with their usual (lexicographic) ordering, then LEX[ID] provides a lexicographic ordering of sequences of words (and thus, for example , on book titles). Similarly, LEX[LEX[ID]] provides a lexicographic ordering on sequences of phrases (such as might be used in sorting a list of book titles), by instantiating the ordering that LEX[X] requires with the one that LEX[ID] provides, namely lexicographic ordering. "

Gougen also describes the use of theories and views to ensure that only suitable actual parameters are substituted for formal parameters. A theory is a collection of properties of an object which fall short of a full description of the object. It may simply be a list of the types and operators expected or may include information about the semantics of an object. For instance in the previous example we would expect X to have an ordering relation over some sort. If we call the ordering relation '$<$' then a property of the actual parameter should be that for all a, b and c where a, b and c are members of the sort, $a < b$ and $b < c$ implies $a < c$. A view describes the relationship between a module and a theory. Taking a hypothetical object for the natural numbers then there are at least two views of the object which would satisfy the above theory. One view would connect the '<' operation in the theory with the usual '<' operation associated with natural numbers, and the other view would connect the '<' with the '>' operation on natural numbers. Either view is valid when applied to the LEX example; one view would give an ascending ordering, the other a descending ordering.

Another important feature of the object orientated approach is the encapsulation or information hiding that can be achieved. Curry and Ayers [19] point out that 'Information hiding is a re-usability mechanism since those parts of a system which cannot 'see' information that must change can be re-used to (re) build the system when that information does change.

3.2.2 Component Libraries

The component library approach to software re-usability divides itself into the issues of what should be stored and how it should be retrieved. Most of the re-usable components currently stored are simply code modules or routines, sometimes with some associated documentation. Examples are the standard C libraries provided on UNIX systems [33] and the growing libraries of re-usable modules being developed by specific enterprises such as the Bank of Montreal [46] and Hartford Insurance [18] to meet their own internal needs. This commonly used technique of storing low level components without additional information or higher level representations [36] produces only limited leverage and a low level of component understandability. Matsumoto [43] focuses on these problems and points out two major features which enhance re-usability ; understandability and abstractness. For any object to be re-used it must be understandable so that it can be easily retrieved and used or modified accordingly. Abstractness enhances the leverage obtained by re-using an object as a more abstract object will be associated with more source code. He suggests storing components at three levels of abstraction: source code, design and requirement. The requirements can then be used as an abstract presentation of the module and traceability between the presentation and re-usable program modules can be established to simplify re-usability .

Wood and Sommerville [60] identify six types of re-usable component as follows.

Functions These are stand alone components which return the same value irrespective of the environment in which they are invoked and do not affect that environment.

Procedures These can be normal procedures as well as value returning procedures.

Declaration packages These are collections of declarations which define an environment.

Objects These are components which have both an associated set of operations and an in-built state.

Abstract data types These are components which are, simplistically, templates for objects which may be created.

Sub-systems These are usually collections of components which are devoted to some particular task.

The advent of transformation systems [50] has added two types of component to this list.

- Active components (transformations).
- Design components, which provide information about design steps used.

Complete systems and programs are not included in this list, essentially because they are not normally used as components. However, re-using systems without modification is the most cost effective form of re-use, the only difficult task being the location of the most appropriate system.

There are three important measures associated with component library retrieval: *completeness*, *recall* and *precision*. In a given retrieval situation (ie. when trying to retrieve components suitable for a particular task), completeness is a measure of suitable components retrieved relative to the number of suitable components in the library; recall is the overall number of components retrieved (both suitable and unsuitable); and precision is a measure of the number of suitable components retrieved relative to the recall. Ideally a retrieval method should produce a high level of completeness and precision along with a low (but non zero) level of recall (so that the re-user is given some components but is not swamped with too many). This requirement is often unrealistic because the interests of high levels of completeness and precision will often contradict the requirement for low but non zero levels of recall. This stems from the fact that the above definitions assume a component is either suitable or unsuitable. This is obviously unrealistic in most cases, and a better model is achieved using levels of suitability. Ideally the level at which a component is accepted as a candidate for retrieval gets higher as the number of potentially retrievable components grows, thus keeping the recall reasonably small. Two diverse examples might be a request for a component that does not exist in the library but is similar to a few components and the request for a component that exists in many different forms some of which are more suitable than others. In the first case the retrievable sample is small and so a low level suitability is used. Thus a system that retrieves a few components that are not highly suitable, may still be considered as precise. In the second example, the retrieval sample is large and a high level of suitability is used. Thus a system that leaves many suitable components unretrieved whilst retrieving a collection of more suitable components may still be considered as complete.

There are a number of existing approaches to component library retrieval. The crudest of these relies on an expert's knowledge of where to find appropriate components. Improvements can be made using keyword matching techniques; however these suffer from the problems of excessive generality, over-rigid classification and difficulties in describing component semantics using keywords[60].

Prieto-Diaz and Freeman [21] describe a faceted classification scheme that provides solutions to the problems of excessive generality and over-rigid classification. Each component in the library is described using a term for each of six facets: Function, Objects, Medium, System type, Functional area and Setting. These terms are each taken from the set of terms defined for that facet. Table 3.2 gives some examples of terms from each facet.

Function	Objects	Medium	System type	Functional area	Setting
add	arguments	array	assembler	accounts payable	advertising
append	arrays	buffer	code generation	accounts receivable	parts repair
close	backspaces	cards	code optimisation	analysis structural	parts store
⋮	⋮	⋮	⋮	⋮	⋮

Table 3.2: Example terms for each facet.

To retrieve a component, six terms including zero or more wild card symbols, are listed, one for each facet and the system then retrieves the list of matching components. Facets are ranked, in the order given above, for relevance; component function is the most relevant and component setting is the least. Using this ranking, components which are retrieved using wild cards are listed according to relevance. Although the use of wild cards provides a crude method of generalising component specifications, query expansion provides a more sophisticated approach to looking for 'close' matches. Within a facet, each term is related to every other by a measure of "conceptual closeness". A request can be expanded by specifying the facet and maximum conceptual separation. In this case, any components which match on the other facets and lie within the maximum separation specified are listed in ascending order of separation. An additional feature of the system described by Prieto-Diaz and Freeman is the ability to rank retrieved components according to their re-usability, thus helping the user to quickly decide on the most suitable component.

Although the work of Prieto-Diaz and Freeman is an improvement on the straightforward keyword and classification techniques it replaces, there is still a dependency on names (or 'terms' according to Prieto-Diaz and Freeman) to describe the function of components. Although the 'Objects' facet provides a little extra information, this only concerns the objects manipulated by the component and not the nature of the manipulation. Also, it is not possible to specify components that manipulate several different objects.

Wood and Sommerville [60] attempt to overcome this problem by using a method for searching libraries based on semantic descriptions of components. Their method uses a collection of software function frames which describe 'basic' software functions by relating them to the objects they manipulate. Some examples of functions for which frames might be provided are: compile, search, control, print, communicate. Each software function frame has a number of labelled slots that relate it to the objects that perform the function, the objects that are manipulated by the function and the objects that are produced by the function. Both the objects and function frames may have modifiers describing them attached. A component is described by taking a software function frame and filling in one or more of the slots. An example given by Wood and Sommerville is based on the UNIX software component 'grep' which searches a file for a specified pattern.

Function: *search*
  Actor: *grep*
  Object that is searched for: *pattern*
  Object that is searched: *file*

In this case the slots have been filled by the grep, pattern and file objects, however the function and objects do not have any modifiers. A component 'fgrep' that searches ordered files using a binary chop could be described by the addition of modifiers:

Function: *search* Has property: *binary chop*
  Actor: *fgrep*
  Object that is searched for: *pattern*
  Object that is searched: *file* (has property: *ordered*)

Components are described by selecting an appropriate software function frame and then filling in as many of the slots as possible. Retrieval is achieved by matching the specified function frame with the function frames of components in the library and listing any matches found.

Although this method provides considerable advantages over the key word approach by providing some details of the semantics of components it still relies on names to describe the components function.

3.3 Deficiencies of current approaches

The principal deficiency of current approaches to component library retrieval is that they all rely on names to describe the basic function of a program. This is inadequate because the single most important characteristic of a component (from the re-users point of view) is its function and individual names are insufficient to describe anything other than a few common functions.

The assertion that a components function is generally the single most important characteristic of a component, is well supported in the literature. The ordering of facets used by Prieto-Diaz and Freeman [21] to classify components is arranged so that the most important facet appears first - they place the 'function' facet first. Wood and Sommerville [60] state that ".. in general, software components perform a function and furthermore it is this function which characterises the software component ..". Other characteristics of components that can be used for component library retrieval are: objects manipulated or returned; environment of component; source language and application domain. Whilst a mismatch in one of these characteristics may mean that the effort of re-use is too high to be worth while, there is still a reasonable chance of re-usability provided the function of a library component is close to the desired function. However, if the 'source code' characteristic matches but the 'function' characteristic does not, the chances of any re-use are slight. An interesting point to note is that there is a strong connection between the 'object' and 'function' characteristics - in many situations the objects manipulated form an important part of the components functionality. For this reason, the 'object' characteristics can often be used as a guide to the components functionality. However, it is clear that for a particular group of objects, there are usually many functions that could be applied, and the use of 'object' characteristics without any additional information about functionality would lead to a high level of imprecision in retrieval.

Having established that, from the point of view of retrieval, the single most important characteristic of components is their functionality, the assertion that individual names are inadequate for describing functionality must be justified.

Given that the names used must be ones that describe the same or similar functionality to different people, the number of appropriate names available is small compared to the number of components in a reasonable size of library. It follows that the number of different components described by a particular name will be large and retrieval precision low. An exception to this might be for highly domain specific programs and systems. In this case the terminology of the particular domain may provide names that describe very specific functions and are commonly used within the domain. However, this is precisely the area in which most of the re-use currently practiced is taking place. To take full advantage of the re-usable potential of components, the names that describe them must not be application domain specific.

For many components there is no obvious name that provides even a good abstraction of its function. The only alternatives are to use a name which expresses a very general abstraction and therefore covers a very large number of components, or to use an obscure name (if a suitable one exists) that is unlikely to be used by anyone else.

As an example, consider the UNIX program 'make' which takes as parameters the name of a file, a description of the inter dependencies between a group of files and a set of rules describing how to build a file from its dependencies. Each file has an associated 'time of last update' which is used to ensure that the file to be made is up to date with respect to its dependencies. If this is not the case then the given file and any out of date dependencies are re-built according to the given rules.

Some possible names for this components function are: 'make', 'build', 'create','maintain' and 'update'. The first three of these names could be considered as synonyms for the same abstraction. Unfortunately this is an abstraction common to many components: If it is used to retrieve components from the standard UNIX library, [3] then 48 out of approximately 1000 components in the library are retrieved. This represents a precision ratio of 1 in 48 (since only one of the components is suitable for re-use). In the context of large libraries (more than 10,000 components) this precision ratio is clearly unacceptable. 'maintain' and 'update'

[3]This can be done using the unix command 'man -k'; however since this does not cope with synonyms it is necessary to use this command once for each synonym

represent altogether different abstractions of the components function. 'maintain' is somewhat obscure and so is unlikely to be used by the re-user and 'update' represents another very general abstraction of component function and gives a precision ratio of 1 in 10.

Another problem with the use of names to describe components functions is the fact that as libraries become larger, an increasing number of components must be described using the same name, since the name space (number of names) cannot grow, without the introduction of domain specific names. As a result, the use of names on a library of 1000 components may work reasonably, but extending the library to 10000 components reduces performance by a factor of ten.

Finally, names are likely to prevent re-use occurring across application domains, where completely different domain based names are used to describe similar functionalities.

3.4 A new approach to component library retrieval

Current methods of component library retrieval suffer from our inability accurately to summarise the behaviour of software components. The use of key words is dependent on the existence of a commonly agreed word for the components function, and works well for components such as 'sort' but fails for more diverse components. The use of keywords is also a barrier to re-usability across application domains which have different terminology for functions that are similar or identical. Retrieving components using a classification scheme suffers from similar problems to the keyword method of retrieval: It relies on the existence of a commonly agreed classification for the component. Also, classification schemes are orientated around the component's use, and hence tend to divide components according to application domains. As a result, finding suitable components from different application domains is very difficult. The use of software function frames is an improvement on the keyword and classification approaches since it attempts to describe some of the properties of the component. However it still relies on names of software functions to a very large extent because it does not provide a mechanism for describing complex functions as a combination of simple functions.

The principal objective of the work described in this thesis is to fill this gap in the field by finding and evaluating improved methods of describing components and retrieving components from libraries using these descriptions.

These methods must:

- Rely on the form of a description rather than the names.
- Allow the user to describe the component at any level of detail they require. It is particularly important that the component can be described in a more informative fashion than by simply categorising or naming its function and the objects that it manipulates.
- Provide for the possibility of re-use across application domains.
- Retrieve components within reasonable time scales. A reasonable time for retrieving a component is one which is substantially less than the time it would take to obtain the component by another method (including rebuilding the component).
- Be capable of high precision and a high level of completeness.

The solution proposed involves two main parts. These are firstly a method of describing components, and secondly a method of comparing component descriptions so that components matching a given description may be retrieved from a library. The components are described using formal property models. These are abstract descriptions of components, and only include the key properties of the component they describe. The are written in a formal language and thus their meaning is derived from their form rather than the names they use. It is the formal nature of the component models which radically differs from existing approaches and provides the possibility of accurate and unambiguous descriptions that can retrieve components across application domains. The language in which models are written is sufficiently powerful to allow the model writer to describe a component at any level of detail. Models can be compared by proving the properties of one from the properties of the other or vice versa. A view between the two models may be used to assist this proof and allow similar components to be compared successfully.

Each component in the library has one or more models describing it and these models are compared with models given by the libraries users. Any models that are successfully compared are ranked by suitability and re-usability and a list of the highest ranked components is presented to the library user.

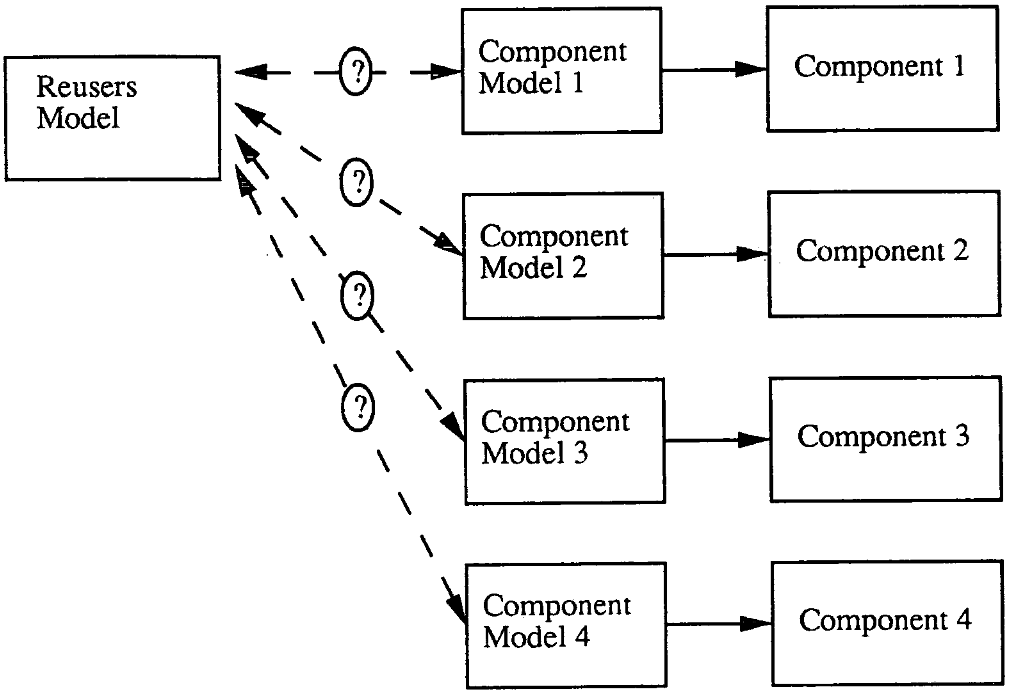

Figure 3.1: Model Based Retrieval

The feasibility of this approach depends on several factors:

- The availability of a suitable language for writing component models.
- The effort required to write models of desired components. If this effort starts to approach the effort of rebuilding the component from scratch the retrieval method is no longer useful.
- The ability of librarians and component contributors to produce models that will be similar to the models requested by library users.

- The practicality of defining a 'similarity' relation between models.
- The practicality of automatically searching libraries for 'similar' models.

Ideally these factors should be tested in a full scale library (>10000 components) containing a variety of component types (code, specifications, requirements, design etc). To carry out such a test, a suitable library would need to be set up or an existing one used. Each component in the library would need to be given one or more model(s) to describe it. Obviously such an experiment is beyond the resources of the research carried out in this thesis. The alternative approach which has been adopted is to investigate the feasibility of the method in a limited context. This context involves a small library of components that are all written in the functional language Miranda.

Miranda and its associated library were chosen for the following reasons:

- A library of Miranda components already existed at Durham.
- The nature of the language Miranda means that components written in Miranda are highly re-usable . Some of the relevant features of Miranda that have been introduced earlier in this chapter are: Higher order functions, parameterised types and lazy evaluation.
- The pure (no side effects) nature of Miranda means that components written in the language are more tractable to mathematical reasoning than components written in languages which permit side effects.
- Many of the properties of a component can be inferred from the component itself thus reducing the effort needed to set up a model based retrieval system for the library.

The remainder of this thesis describes the method of model based component library retrieval that has been devised and the subsequent investigation into the feasibility of this method.

Chapter 4

Retrieval by formal property descriptions

There are two essential ingredients to any method of component library retrieval. The first of these is a method of describing components and the second is a method of searching for components using these descriptions. The initial section of this chapter deals with the language for describing components and the latter sections describe the method for comparing components.

4.1 Component descriptions

The primary goal of the component description language is to describe the function of a component and the type of objects manipulated by the component without relying on the meaning of names contained in the description. This suggests the use of a language in which meaning is derived only from the form of the description rather than the names used by the description — in other words a formal language.

A complete functional description of a component could be provided by a full formal specification of the component; formal specifications are unsuitable for a number of reasons[60]:

- They require too much effort to produce.
- Specifications that are semantically identical but syntactically different are very difficult to compare.
- Defining a 'similarity' relationship between specifications is difficult.

These reasons for the unsuitability of formal specifications for retrieving components are worthy of further discussion. Firstly, for the re-use of a component to be advantageous, the effort of re-using should be (substantially) less than the effort of rebuilding the component. Although the effort of producing a formal specification for the component should be less than the effort of building an efficient implementation, it is closely related to the size of the component and forms a substantial part of the effort needed to produce the component[27]. It is possible that the desired component is itself a specification, in which case a formal specification is clearly an unacceptable means of retrieving the component.

Secondly, the comparison algorithm must take the semantics of the specification language into account, and must therefore have the capabilities of a theorem prover. For substantial proofs, such as the ones required to prove theorems about a specification and hence compare specifications, it is essential to divide the overall proof into a number of sub proofs (which may in turn need to be divided). Such a proof therefore takes on a similar structure to a large program or specification. The limitations of current theorem proving technology mean that whilst a theorem prover may be capable of proving the sub proofs and also of constructing the overall proof from the sub proofs, it is not capable of the strategic planning necessary to identify the sub proofs and is therefore not capable of completing large and complex proofs without human assistance[13].

Finally, a retrieval method should be capable of retrieving not just components that meet the given requirements precisely, but also ones that meet only some of the requirements or are similar in some sense. To use specifications for component library retrieval, the notion of 'similarity' between specifications must be precisely defined and an algorithm for detecting similarity must be available.

The alternative proposed in the research reported by this thesis is to describe the component in terms of an abstraction that covers only the most important aspects of the component. The description should still be in a formal language, but provided the level of abstraction is sufficiently high, it should be small enough to produce without a prohibitive effort and the task of proving equality of descriptions should become feasible. To emphasise their differences from formal specifications and the fact that they are scaled down specifications which only describe the key features of a component, the component descriptions are referred to as "models" of the component or "property models".

Although a property model is similar to a specification in that it describes the function of a component without necessarily describing a method of computing the function, there are several key differences between the two which reflect their different objectives:

- A specification is intended to provide a precise and unambiguous description of a component which completely describes its desired behaviour. A property model may be incomplete in the sense that it does not describe all of the desired behaviour.
- A property model need only describe the key objects and operations of the component it models, whereas a specification should describe all objects and operations.
- A property model can replace groups of functions or objects with abstractions which may consist of fewer functions, or fewer objects or functions with fewer arguments.
- A specification can be arbitrarily large, but a model must always remain small, even if a large component is being described.
- A specification is used to ensure that the component described by the specification fulfils its intended role, and also to provide a representation against which the implementation can be verified. A property model is used to describe the functions and operations of a component so that the component may be retrieved from a library by comparison of required and stored models.

Although property models assist the retrieval of components "similar" to the desired component, the ability to compare models for similarity rather than equality is important. In addition to the property models, this thesis proposes a similarity relation based on the existence of a 'view' between models, over which they can be shown as equivalent. Before giving details of this relation, the formal language used to describe components is introduced.

4.2 The Property Model Language "Miramod"

One of the requirements of a component library retrieval system is its applicability to a wide variety of components. This implies that the property model language must be capable of describing many types of component. A common method of coping with such a variety of components is to describe each in terms of its function and the objects manipulated by it [21, 60]. As the objective is not to rely on the meaning of names to describe the functions and operations, it is necessary to be more precise about what is meant by these terms. A components 'function' may either be a pure function in the mathematical sense, or an operation which depends upon and/or alters a state. Likewise the objects manipulated by the component may be data types or objects which contain a state. It is also possible that the component is best described directly as an object or data type, rather than as a function or operation.

The following list gives examples of some common classes of component, and their descriptions in terms of functions, operations, data types and objects.

Firstly some examples of components which are sub programs are listed, along with the method of describing these components:

Functions (pure) can be described by their function and their domain and range types.

Procedures (and value returning procedures) can be described by their operation, the state object(s) that they manipulate and their parameter types.

Modules (in the sense of component collections) can be described in terms of each individual component (in other words as a collection of functions, operations, data types and objects).

Objects (with procedures and a built-in state) can be described as a collection of operations on a state.

Data types (Abstract or concrete) can be described directly as data types.

Similar methods can be used for describing programs:

Single function programs (e.g. a compiler) can be described by their function and domain and range types.

Interactive / Multiple operation programs can be described as a collection of operations on a state. They can also be described as functions, however the 'collection of operations' model often provides a better abstraction.

Finally, complete systems can also be described:

Interactive / Multiple operation systems can be described as a collection of operations on a state.

Rather than providing separate notations for functions and operations, it is possible to describe operations as functions which take a state as one of their parameters and return a state as part of the result. Likewise, objects can be described as collections of operations over a state data type.

The component description language is therefore capable of describing the properties of one or more functions and data types. It is based on extensions to the functional programming language Miranda[59] - hence the name "Miramod". Miranda has been used as a basis because:

- The pure functions used by Miranda are easy to reason about since they contain no side effects.

- The notation is clear and concise, allowing models to be produced with the minimum of effort.
- The notation is based on the standard ASCII character set, and hence standard tools such as text editors, can be used to manipulate the models.
- The test library is populated by Miranda components and in many cases these act as their own models, thus avoiding the need to produce a model for every component in the library.
- The similarity between component and model notation reduces the learning necessary for users of the retrieval system
- Higher order functions lead to a powerful notation and a high degree of reusability.
- The principles of strong typing allow a large percentage of errors to be detected automatically.

Although Miranda is used as a basis for the component description language, it has several shortcomings from the point of view of describing component models (Miranda is essentially a programming language rather than a specification language).

- Miranda only provides a notation for defining functions and therefore components must be described through their definitions. The notation for defining functions is restricted so that all defined functions are executable and hence many important properties of functions cannot be described directly and functions are generally hard to describe.
- The Miranda notation for algebraic types is unsuitable for describing types in general.

Miramod contains extensions to Miranda that overcome these problems. A property statement notation is introduced to allow the properties of components to be described more abstractly than they can be described using the Miranda definition notation. As a result the user can concentrate on the properties of a component or 'what the component does' rather than how it should be evaluated. These property statements are based on Miranda Boolean expressions but provide a number of additional constructions and allow universally quantified variables to be introduced. The notation for types is also extended to allow more versatile algebraic type descriptions.

The following sections introduce first the notation for describing functions and then the notation for describing data types. Finally the built-in types of the language, which can be used as a basis for component models, are described. No previous knowledge of Miranda is assumed.

4.2.1 Properties of functions

A function can be described in two ways, firstly in terms of its type and secondly it can occur as part of a property statement which describes its properties in terms of other functions.

A simple *property statement* is based on a (Miranda) Boolean expression, usually containing one or more equalities. For example, to describe the function for reversing lists by stating that for all `x` the result of reversing `x` twice is `x`, the following model can be written:

```
{x} reverse (reverse x) = x
```

The notation `{x}` introduces a property statement and also lists any universally quantified variables in the expression. If several variables are used they must be separated by at least one white space character. As with Miranda, function application is denoted by juxtaposition i.e.

'`f x`' denotes the function `f` applied to one argument.

'`f x y`' denotes the function `f` applied to two arguments.

This notation may seem unnatural to those who are familiar with the standard mathematical notation which uses structured arguments enclosed in braces and separated by commas (for example '`f(x)`' and '`f(x,y)`'). It can be justified on two accounts. Firstly the reduction in the number of braces appearing in expressions prevents them from becoming cluttered and makes them easier to read. Secondly, it provides a powerful and elegant method of describing and using *higher order functions* (functions which take functions as parameters or return functions as results). For example, if there is a function `add` which adds two numbers, it can be used in three ways:

1. '`add 1 2`' denotes the result of adding 1 to 2.
2. '`(add 1)`' denotes the function which "adds 1 to things".
3. '`add`' denotes the function which adds two numbers and can appear as the argument to a function as well as in the above contexts.

For the function application notation to work in a consistent manner, it must be considered as a left associative binary operator. The expression '`add x y`' is parsed as '`(add x) y`' rather than '`add (x y)`' and can be interpreted as "the function that 'adds `x` to things' applied to `y`".

A full property statement is made up from a variable list, a sequence of *property expressions* separated by the symbol '`;;`' and an optional `where` part (which allows local definitions to be introduced and is described in section 4.2.1). Each property expression may use one or more of the property statements variables and is either a Boolean expression (typically an equality `=` or inequality `~=`) or a conditional. A conditional property expression has the form '`p |- q`' where `p` and `q` are property expressions, and means "if `p` then `q`" or alternatively "`p` proves `q`". For example, the function `abs` can be described using the following model:

```
{x}
    x<0   |- abs x = -x ;;
    x>=0  |- abs x = x
```

Sensible use of layout is used to determine the end of the property statement. The *off-side rule* insists that the body of the property statement (everything that follows the initial '{') must be to the right of, or in the same column as, the first (non space) character of the body. This allows property statements to be continued over several lines:

```
{x} reverse (reverse x)
    = x
```

is legal but

```
{x} reverse (reverse x)
=x
```

is not.

Boolean expressions are written using standard Miranda notation and may include both Boolean and relational operators. The Boolean operators are listed below in order of increasing binding power:

\/ is the associative operator denoting logical disjunction.

& is the associative operator denoting logical conjunction.

~ is the prefix operator denoting logical negation.

The standard relational operators (=, ~=, <, >, <= and >=) are defined over all non-function types. The = and ~= operators have already been used to describe equalities and inequalities between expressions. A combination of these operators may be written in arbitrary length sequences, for example:

```
    a < b < c
```

The following property statement describes the properties of the ideal[1] **sqrt** function:

```
{x} x>=0 |- sqrt (x^2) = x) ;;
    x<0  |- sqrt x = undef
```

The second line of this example describes the behaviour of the function if it is given a negative argument, and could be omitted if this aspect of the functions behaviour is not considered crucial. The value **undef** is an exception condition or program error and is a member of all types. In fact there are a family of exception conditions produced by the function `error` which takes an error message as parameter.

Using types to describe functions

Functions can also be described by their type. The notation is identical to the (optional) type declarations of Miranda.

For example, consider the function `reverse` from the previous example. Although it was not clear from the property statement, the intended type of `reverse` was a function from a list of numbers to a list of numbers. To make this clear, the following declaration could be added:

```
reverse:: [num] -> [num]
```

where

'*identifier* :: t_1' describes the given identifier as an instance of a particular type 't_1'. In this case the identifier is `reverse` and the type is `[num] -> [num]`.

[1]Although this description would be unacceptable in the context of a specification (in general it is only possible to calculate an approximation for the square root of a number), it is acceptable in the context of component library retrieval since the description of a component does not have to be precise in every detail and there is no obligation to prove that a component meets its description.

As with property statements, intelligent use of layout means that no special symbol is needed to terminate the type description.

't_1 -> t_2' denotes the type of a function which takes an argument of type t_1 and returns a result of type t_2.

'[t_1]' denotes the type of a list with elements of type t_1.

'num' is the built-in type for numbers.

The above example demonstrates how the type of a function with only one argument is described. In the general case of a function with n arguments, the type is denoted:

t_1 -> t_2 -> ... -> t_n -> t_r

where t_1 is the type of the first argument, t_n is the type of the last argument and t_r is the result type. The -> operator is right associative, so the following two descriptions of the add function mentioned previously are equivalent:

```
add:: num -> num -> num
add:: num -> (num -> num)
```

The type notation is therefore consistent with the function application notation which allows add to be applied to two numbers and return a number, or to be applied to a single number and return a function from number to number.

It is important to note the distinction between the types '(num->num)->num' and 'num->num->num'. The first takes a function as its only parameter and returns a number as its result, whereas the second takes two parameters, both numbers, and returns a number as the result. For example the function applyto1 which applies functions to the number 1 could be described:

```
applyto1:: (num->num) -> num
{f}
    applyto1 f = f 1
```

Polymorphic Types

The type of a function need not be described completely (for example **reverse** might be described as a function from a list of any type to a list of the same type). An incomplete type description can be formed by including one or more type variables in the description. In the property model language, as well as in Miranda, type variables are denoted as sequences of asterisks (*,**,*** etc). Types containing variables are called polymorphic (literally "many form") types[44].

The type of a function `f` that takes an unspecified type as argument and returns a result of the same type can be described using the following declaration:

```
f::* -> *
```

Using two distinct type variables is even less informative:

```
f:: * -> **
```

`f` is now a function from any type to any type (it includes functions of the type `* -> *`).

A partial ordering over types is defined by the *specialisation* and *generalisation* relations. The type `A` is a specialisation of the type `B` if there is a substitution for variables in the type `B` which gives the type `A`. If type `A` is a specialisation of type `B` then type `B` is a generalisation of type `A`. The ordering is partial because there exists pairs of types `A` and `B` such that `A` is neither a specialisation nor generalisation of `B`. In this case, provided they are not equivalent, types `A` and `B` are said to be *inconsistent types.*

To give some examples:

'`* -> *`' is a specialisation of '`* -> **`';
'`[*]->[*]`' is a generalisation of '`[num]->[num]`';
'`* -> * -> *`' is inconsistent with '`* -> *`' and
'`* -> *`' is equivalent to '`** -> **`'.

If a type is described using a type declaration, then the declared type must be equivalent to the type implied by any uses of the function in property statements. So if `reverse` is declared as a function `[num] -> [num]` it may only be applied to lists of numbers and return lists of numbers. An attempt to apply `reverse` to a number (as opposed to a list of numbers) in a property statement would result in a type error. This rule is one of several which ensure that any model is strongly typed, and the remainder of these are described in the following section.

Strong Typing

As with Miranda, the property model language is strongly typed. This means that each expression and subexpression has a type; any inconsistencies in the type structure of the model can be detected through a static analysis of the program; and type errors cannot occur whilst proving theorems about a correctly typed model.

The justification for strong typing is that it allows the compiler to detect many typographical and logical errors in a model at compile time, thus increasing the confidence in models that compile successfully. It is interesting to note that strong typing in functional languages can be a far more powerful tool than in imperative languages since the fundamental construction (function application) must conform to type rules, whereas the fundamental construction of an imperative language (statement sequence) does not involve any notion of type. Experience with strongly typed functional languages has shown that a high percentage of logical errors as well as typographical ones are detected at compile time, which compares favourably with imperative language compilers which are good at detecting typographic errors but often fail to detect logical errors.

For a model to be consistently typed, the type of all functions being described and all free variables must be the same in each context that they appear[2].

For example the following model is consistently typed:

[2]This restriction is relaxed for functions whose type is explicitly given (see page 72).

```
{x} reverse (reverse x) = x
```

but the model

```
{x} reverse [1..n] = [n,n-1..1];;
    reverse "abc" = "cba"
```

is not since reverse is applied to a list of numbers in one context and a list of characters in another (`[1..n]` is the list of integers from 1 to n inclusive[3] and `"abc"` is a shorthand notation of the list of characters `['a','b','c']`).

Type Inference

A disadvantage of the explicit type declarations previously introduced is that they can drastically increase the size of the model and hence the work done in producing the model. A useful solution to this problem is provided by type inference. This means that the type of a function need not be declared but can be inferred from the context(s) in which it occurs.

To illustrate this point, consider the following example:

```
{n} reverse [1..n] = [n,n-1..1]
```

Since reverse is applied to a list of numbers and its result is equated with a list of numbers, it can be inferred that:

```
reverse::[num] -> [num]
```

and

[3]This notation should not be confused with notation for subrange types used by languages such as Pascal. Miramod uses the notation `[1..n]` to denote a list value, not a list or subrange type.

```
n::num
```

As this example demonstrates, the type inference applied to property expressions not only infers types for functions, it also infers the types of the free variables. This is important not only as a consistency check — it also affects the meaning of a property statement since the types of free variables determine the meaning of the property statement.

For example, given a property statement with universally quantified variable x:

```
{x}  p x
```

If the inferred type for x is * (any type), then the property statement means that p should hold for any x, regardless of the type of x. If however the inferred type for x is num, then the property statement means that p should hold for any number x. The first version is a much stronger property statement than the second, since the first implies the second but not vice versa.

As the types of functions and variables are critical to the meaning of property statements, it is important that the declared type of a function is the same as its inferred type. Hence whenever a model contains type declarations for functions, then these are also checked for consistency with occurrences of the function.

This gives rise to alternative motivations for the inclusion of type declarations. Firstly they can be used to make the meaning of property statements clearer and provide additional security against logical errors. Secondly they can be used to specialise the inferred polymorphic type of a function which will in turn weaken any property statements including the function. This may sound like an undesirable effect; however it is often as easy to ask for too much (a strong property) as it is to ask for too little (a weak property). For example the property statement:

```
{x} reverse (reverse x) = x
```

is too strong, as the inferred type of `reverse` is `* -> *` and it therefore describes a function which, when applied to anything twice will return the original thing. This is certainly not true of the function that only reverses lists. A function is wanted which, when applied to a list twice will return the original list. Simply adding the appropriate type definition will produce the required meaning of the property statement:

```
reverse:: [*] -> [*]
{x} reverse (reverse x) = x
```

An alternative approach to the problems of specialising and clarifying the meaning of property statements is to declare the types of the free variables themselves, rather than the functions. To facilitate this, type declarations are allowed to appear as parts of property statements. In fact the sequence of expressions in a property statement can be a sequence of expressions and type declarations. This allows the types of the statement's free variables to be declared directly rather than inferred.

```
{n} reverse [1..n] = [n..1];;
    n::num
```

As with all other type declarations, the type declarations in property statements are checked for consistency with the inferred type of the identifier they describe.

The type laws for property statements can be summarised by saying that for all described functions whose type is not given explicitly and for all universally quantified variables, the type of the function or variable implied by its context must be the same for each context in which the function or variable appears.

Defined functions

When describing functions with property statements, it is often useful and sometimes essential, to make use of functions and operators that are defined as part of Miramod. It is also useful to be able to define auxiliary functions and use them in the property descriptions. In both cases these functions are called *defined functions* to distinguish them from the functions described by the property model (the *described functions*). Defined functions in Miramod play the same role as hidden functions in many specification languages [25].

In the following example, the defined function `ident`[4] is used to describe the functions `plus` and '`and`'. `ident` takes a binary function and a value as parameters and returns true if the value is an identity value for that function (in other words if the value appears as the first parameter to the function then the function returns the second parameter's value). The functions `add` which adds numbers and '`and`' which gives the conjunction of two Boolean values may be described using the properties that the identity value for `plus` is `0` (because `0+a=a`) and the indentity value for '`and`' is `False` (because `True & b = b`):

```
{}  ident plus 0;;
    ident and True
```

This example illustrates several points. The first is that `ident` is a higher order function because it takes a function as a parameter. The type of ident is written:

```
ident:: (*->*->*) -> * -> bool
```

Note that the braces are needed to show that `ident` takes two parameters, the first of which is a function, rather than four parameters of the same type.

The second point brought out by this example is that the type implied by both occurrences of ident is a different instantiation of the generic type of `ident`. In the first line * has been instantiated with the type `num` to give:

```
    ident:: (num->num->num) -> num -> bool
```

[4]`ident` is defined as `ident f e = (f e == id)` where `==` is the strong equality operator described on page 112 and `id` is the Miranda identity function.

and in the second * has been instantiated with the type bool to give

```
    ident:: (bool->bool->bool)->bool->bool
```

Under the type rules for described types, this would be illegal. The type rules for defined types allow them to be used in contexts where the type of the function is a specialisation of the generic type. One reason for this apparent inconsistency is that the stricter rule is needed so that the type of described functions can be inferred, however the type of defined functions can be inferred from their definition, so the stricter rule need not be applied to uses of such functions. This weaker rule is also used for described functions whose type is given explicitly, since it is only neccesary to ensure that the given type is consistent with the contexts in which the function appears.

User defined functions

As well as providing a number of predefined functions that can be used to help describe other functions, Miramod allows model writers to define their own functions. These functions are called user defined functions. Since the notation for user defined functions is Miranda[5], for which there exists several descriptions[9, 59], only a brief summary of the notation will be given here. In the following examples, each function is given a type description for additional clarity. These type descriptions are not compulsory.

Simple definitions are written as equations, with the function name and formal parameters on the left hand side of the = symbol and the value of the function on the right:

```
square:: num -> num
square x = x*x
```

[5]The Miranda notation is extended to allow the strong equality operator in expressions (The strong equality operator is described on page 112).

```
nor:: bool -> bool -> bool
nor a b = ~(a \/ b)
```

Conditional definitions are written by following the function name and formal parameters with a series of alternatives made up from an expression and a Boolean "guard".

```
max:: * -> * -> *
max a b = a, a>b
        = b, a<=b
```

The value of the function is the first (upper most) expression whose guard evaluates to `True`. The final guard may be replaced by the keyword `otherwise`, in which case the last expression is returned if none of the preceding guards evaluate to true.

```
max a b = a, a>b
        = b, otherwise
```

Functions may also be defined using pattern matching:

```
and:: bool -> bool -> bool
and True x = x
and x y = False
```

```
unit_list:: [*] -> bool
unit_list [a] = True
unit_list as = False
```

Note that patterns may overlap, and in this case the first (upper most) definition that matches the arguments is taken as the definition for those particular arguments. Hence the above definitions ensure that if 'and' has `True` as its first argument then the second argument is returned. On the other hand, if `True` is not the value of the first argument the value False is returned. This means that if patterns overlap, the order of the definitions is important.

The right hand side of a definition may introduce local definitions using a where clause.

```
area:: num -> num -> num -> num
area a b c = (s-a)*(s-b)*(s-c)
             where
             s = (a+b+c)/2
```

A conformal definition, whose left hand sides is a pattern without a preceding function identifier, can be used to extract values from a function which returns more than one value:

```
fib 0 = [0,1]
fib (n+1) = [b,a+b] where [a,b] = fib n
```

Within a model, function definitions may appear at top level, or as part of a property statement. The ordering of top level definitions, type declarations and property statements is unconstrained; there is no obligation to define a function before it is used.

To include definitions in a property statement, the list of expressions and type declarations must be followed by the keyword where then the definitions. The scope of the functions thus defined is the whole of the property statement.

```
{}  twice reverse = id
    where
      twice f x = f (f x)
      id x = x
```

4.2.2 Describing Types

The two common approaches to describing data types are the algebraic approach and the representation approach[6]. In the representation approach, the data type is described in terms of an existing (predefined) type - the representation. In the algebraic approach, the data type is described in terms of the properties of a collection of functions over the type.

Both approaches describe the syntax and semantics of the type separately. The syntax of the type is described by listing the functions which take or return values of the type. The type of each listed function is also declared and this information is called the *signature* of the type. Each function in the signature can be classed as a constructor or destructor function. The *constructor* functions return values of the type (they construct values of the type). The *destructor* functions are ones which return values of other types. The semantics of the type are described differently depending on whether the representation approach or the algebraic approach is used.

Representation based type descriptions

The syntax of a representation based type description is almost identical to that of the Miranda abstract type declaration[7] The type is described in terms of a collection of the basic functions by which it is manipulated. Taking the classic example of a stack, these functions might be: `empty` for creating an empty stack, `push` and `pop` for pushing and popping values onto and off the stack, `top` for inspecting the element at the top of the stack and `is_empty` to test if the stack is empty.

[6]The representation approach is more commonly referred to as the model approach. Here the word 'representation' is used to avoid confusion with the concept of a property model.

[7]The only difference between the two is the use of the keyword **abstype** by Miranda and **type** by Miramod, otherwise they are syntactically and semantically identical. The different names stem from the different purposes to which they are put. In Miranda the abstype mechanism is used to hide the details of a type's representation from the users of the type, hence the type is an abstraction of the more detailed representation. On the other hand, when describing data types, the representation is in many senses more abstract than the type being described; in fact it is a model of the type being described.

The first part of the type description consists of a declaration of the type's name and the signature of the type:

```
type stack *
with
    empty    :: stack *
    push     :: * -> stack * -> stack *
    pop      :: stack * -> stack *
    top      :: stack * -> *
    is_empty:: stack * -> bool
```

The type may be polymorphic, in which case the type name is followed by the appropriate number of type variables. The signature of the type is written following the reserved word **with**, and consists of a list of type declarations for the functions which describe the type.

The signature should be followed by a declaration of the type being used as a representation. In this case, the list type will be used to represent the stack:

```
stack * == [*]
```

Finally, property statements can be used to describe the functions of the stack in terms of the representation (: is the list construction operator and [] is the empty list):

```
{e s}
    empty = []              ;;
    push e s = e:s          ;;
    pop (e:s) = s           ;;
    top (e:s) = e           ;;


    is_empty [] = True      ;;
    is_empty (e:s) = False  ;;
```

```
ret_stack [] = empty          ;;
ret_stack (e:s) = push e (ret_stack s)  ;;
valid_stack s = True
```

The functions `ret_stack` and `valid_stack` are needed for the comparison of component models[8] and their description is therefore mandatory. `ret_stack` is a *retrieval function* that converts the representation type to the type being described (the *described type*) and `valid_stack` is a predicate on the representation type that is true for all values that can be expressed by the described type and false otherwise. The function `ret_stack` should be defined for all `s` such that `valid_stack s` is true.

Thus every representation type description has two associated functions whose names are `ret_` and `valid_` followed by the name of the type. The retrieval function has the type t_r `->` t_d where t_r is the representation type and t_d is the described type, and the validity predicate has the type t_r `-> bool`. These types are assumed for the appropriately named functions and must not be included in the signature of the type.

There is an additional type law associated with representation type descriptions. This states that for any function included in the signature of the type, the representation type and the described type are considered as equivalent. In any other context the two types are distinct. In the stack example, this means that expressions such as

```
empty = []
```

are correctly typed despite the fact that

```
empty::stack *
```

and

[8]See chapter 6, section 6.9

```
    []::[*]
```

This type law provides the motivation for disallowing the functions `valid_stack` and `ret_stack` from the type's signature — it forces the writer to produce useful properties for these functions rather than writing

```
{s} ret_stack s = s             ;;
    valid_stack empty = True
```

Algebraic type descriptions

An algebraic type description describes the type in terms of the signature functions and their properties rather than a representation. The syntax of an algebraic type description is identical to that of a representation type description except that there the declaration which links the described type to a representation (`d_type == r_type`) is omitted.

As an example, the type '`set *`' has functions: `empty` for creating empty sets, `add1` for adding an element to a set, `member` for testing set membership and the functions `union`, `intersection` and `diff` for combining sets.

```
type set *
with
    empty   :: set *
    add1    :: * -> set * -> set *
    member  :: * -> set * -> bool
    union, intersect, diff
            :: set * -> set * -> set *

{a b s t}
    member a empty = False                          ;;
    member a (add1 a e) = True                      ;;
    a ~= b |- member a (add1 b e) = member a e      ;;
    member a (union s t) = (member a s \/ member a t)         ;;
    member a (intersect s t) = (member a s & member a t)      ;;
    member a (diff s t) = (member a s & ~member a t)
```

Algebraic type shorthand

The algebraic type notation can be long-winded when used to describe simple type structures such as records (product types), type unions (sum types) and combinations of the two (sum of product types). To overcome this problem a special notation for describing types in terms of *shell* functions is provided. The shell functions are functions which return values of the described type and denote a unique value of the type for each combination of argument values.

A good example is the type `tree *` which is either a leaf containing a value of type * or a node containing two sub trees:

```
tree * ::= Leaf * | Node (tree *) (tree *)
```

The symbol `::=` should be read "has shells" and is followed by a list of shell function names with their parameter types. The vertical bar `|` is used to separate shells and any type variables used by the shells must appear as parameters to the type being described.

The equivalent longhand is:

```
type tree *
with
    leaf:: * -> tree *
    node:: tree * -> tree * -> tree *

{l r l' r' e e'}
    leaf e ~= node l r ;;
    (node l r = node l' r') = (l=l' & r=r')
```

```
(leaf e = leaf e') = (e = e')
```

The two properties above distinguish `leaf` and `node` from other functions which return values of the type `tree *`. The first property expression ensures that each shell produces values that are distinct from the other shell, and the second property ensures that each combination of argument values produces a unique value in the described type. To emphasise the fact that these functions have implicit properties, the constructor names of the short-hand version must start with capital letters.

The notation for short-hand algebraic types is identical to Miranda's algebraic type notation, except that Miranda also allows 'laws' to be attached to algebraic types. Laws are not allowed in Miramod as they are superceded by the longhand algebraic notation.

A Pascal enumerated type can easily be described using the algebraic type shorthand. The elements of the type are represented by shells with no parameters:

```
weekdays ::= Mon | Tue | Wed | Thur | Fri | Sat | Sun
```

On the other hand, a Pascal record type can be described using a single shell function with one or more parameters; for example the type `date` might be the product of the types `day`, `month` and `year`.

```
date ::= Date day month year
```

As with the longhand notation, functions over the type may be described using property statements.

Predefined types and functions

The predefined types and functions provided by Miramod play an important role in assisting the description of other types and functions. In fact all of the built-in types may be considered as type descriptions in their own right. The notation used is an extension of the Miranda notation for built-in types and functions.

There are three primitive types, numbers, characters and Boolean, and these are denoted `num`, `char` and `bool` respectively. The distinction between real numbers and integers is handled internally and all the usual numeric operators are provided. Floating point notation (eg `57.6e-10`) can be used to denote constant numeric values. Character constants appear between single quotes (eg `'a'`) and the usual C language conventions apply for unusual characters. The type bool has two values `True` and `False` and the operators mentioned in section 4.2.1 are available for manipulating Boolean values.

Two compound types, the list and tuple, are available. Although much of the notation for lists has already been covered, it is summarised here along with some previously unmentioned operators.

The list type is denoted $[\alpha]$ where α is the element type. The length of a list is not fixed, but all the elements must be of the same type. The empty list is `[]` and `[a,b,c]` is the list containing three elements `a`, `b` and `c`. Several operators are provided for manipulating lists:

`:` is the list construction operator (e.g. `1:[2,3] = [1,2,3]`);

`++` provides list concatenation;

`--` provides list difference (e.g. `[1,1,2,3,4] -- [1,3] = [1,2,4]`);

`#` is a prefix operator giving the length of a list;

and `!` is the list indexing operator (e.g. `[1,2,3]!0 = 1`).

Tuples are analogous to Pascal records. They have a fixed number of elements, and each may be of a different type. The type of a tuple with n elements of types $t_1, t_2 \ldots t_n$ is denoted $(t_1, t_2, \ \ldots, t_n)$ and an n-tuple of values $e_1, e_2 \ldots e_n$ is denoted $(e_1, e_2, \ \ldots \ , e_n)$.

Several additional notations are available for describing lists of numbers:

`[a..b]`	list of integers from `a` to `b` inclusive
`[a..]`	infinite list of all integers `>=a`
`[a,b..c]`	list of numbers in the arithmetic series a, b ... c with largest member not exceeding c (if b-a positive, smallest member not less than c if b-a negative).
`[a,b..]`	infinite list of numbers in the series starting at a, interval = (b-a)

List comprehensions are a useful notation for defining lists whose elements satisfy some properties. A list comprehension is written:

[e | q_1; q_2; ... ; q_n] (where n>=0)

and denotes the list of all values of the expression e such that the qualifiers q_1 .. q_n hold. Qualifiers are Boolean expressions or generators. A generator is an expression of the form:

```
plist <- list
```

where `plist` is a list of one or more patterns separated by commas. Each pattern contains one or more of the variables introduced by the generator (patterns are expressions that contain no operators or functions other than user defined shells and the built in shells '`:`' and '`+n`'), and variables introduced by generators come into scope from left to right.

For example the following is the (infinite) list of Pythagorean triangles:

```
[(a,b,c)| a,b,c <- [1..]; a^2+b^2=c^2]
```

Lists of patterns on the left hand side of `<-` are shorthand for multiple generators from the same list, e.g. `a,b,c <- [1..]` is shorthand for `a<-[1..]; b<-[1..]; c<-[1..]`.

It is sometimes useful to allow universally and existentially quantified variables to be introduced as part of a property expression. Miramod does not allow true existentially quantified variables and only allows universally quantified variables to be introduced at the start of a property statement. In many situations, it is possible to use the list comprehension notation and the functions **exists** and **forall** to 'fake' universal and existential quantifiers. These functions are defined by the following equations:

```
    exists,forall:: [bool] -> bool

    exists [] = False
    exists (a:x) = a \/ exists x

    forall [] = True
    forall (a:x) = a & forall x
```

As an example, consider the description of a function which returns an unspecified member of a list:

```
{s y}  exists [ element s = y | y <- s]
```

This expression can be read "there exists a y, which is a member of the list s, such that 'element s' is equal to y".

This concludes the description of Miramod. Further details of the language, particularly the predefined functions that are available, will be described as and when they are used. Having described the language Miramod and its use to formalise the properties of functions and data types, the remainder of this chapter describes how this language can be used as a basis for component library retrieval.

4.3 Writing property models

There are two distinct situations in which property models are written. One is when a component is added to the library and the other is when a re-user needs to retrieve a component from the library. In the former case, the details of the component's behaviour may be available in the form a specification of the component, or they may be inferred from the component itself. Even if a formal specification of the component is available, it will generally be too large and complex to be used for retrieval purposes, so a small and simple model containing the most important aspects of the component must be written. The two main techniques that can be used to bridge this gap between the specification and its model are simplification and abstraction. Simplification involves removing parts of the component that are not essential for the description of the remaining parts of the component. This works well for components such as text editors which have many sophisticated operations that are built on top of a few basic ones. The sophisticated operations can be removed without affecting the remaining operations and without destroying the basic capabilities of the component. On the other hand a component such as a compiler would be extremely difficult to model satisfactorily using just simplification - a model which described the compilation of just one or two language features would not be very informative. In such cases the technique of abstraction is more appropriate. In this context abstraction involves replacing several parts of the model with a single part, or at least a smaller number of simpler parts that collectively describe a similar but simpler behaviour.

A re-user wanting to search for a component in the library is faced with a similar problem, though unless he or she has a very precise idea of what is wanted, the gap between the requirements and a reasonable model should be smaller. In this situation simplification and abstraction can again be used to help isolate the key properties of the required component.

As an example, suppose that a parser or compiler generator, with the following features, is needed:

- Parsers or compilers can be generated from a grammar and set of compilation rules.

- The grammar is in BNF (Backus-Naur Form) and is context free. It provides a notation for terminals (these are the basic atoms recognised by the grammar and are also called 'tokens') and non terminals, arbitrary length sequences, arbitrarily large sets of alternatives, optional parts, zero or more repetitions, one or more repetitions and fixed numbers of repetitions.
- The compilation rules are arbitrary pieces of code which can be attached to the desired parts of the grammar and have access to the results of sub compilations as well as the compilation 'state'.
- The compiler generator should be capable of producing compilers for sequences of any type of token specified by the grammar and should be capable of producing any type of output specified by the compilation rules.
- The resulting compiler should return one of a set of 'fail' values specified by the grammar and compilation rules if the input does not conform to the grammar or the compilation rules detect an illegal input.

If these features are included in the model, it will be complex. An alternative is to produce a simpler version by deciding on a few of the more fundamental (or easier to describe) aspects. In this case the ability to decide if an input sequence conforms to the grammar might be considered as the most important aspect of the component. One argument in favour of this view is that the compilation rules would make little sense without a grammar, but a grammar can still function sensibly without a set of compilation rules - in effect the component can be described in terms of its ability to generate a grammar checker from a grammar.

Another source of complexity is the description of the grammar. A powerful abstraction is to replace the conventional representation of a grammar as a set of terminal symbols, a set of non-terminal symbols, a set of grammar rules and a start symbol, with an abstract type which has only three constructor functions. These are: `token` which constructs grammars that recognise single token sequences (i.e. sequences of length one), `then` which takes two grammars as parameters and returns the grammar that recognises sequences of the first parameter's grammar followed by sequences of the second parameter's grammar, and finally `alt` which takes two grammars as parameters and returns the grammar that recognises sequences of either of its parameter's grammars. A very desirable feature of this abstraction is that it retains the ability to describe almost any grammar that can be described with the more complex set of constructions[9]. The only construction that cannot be defined in terms of the abstraction is the 'optional' construction.

A final simplification that can be made is the replacement of the set of 'fail' values with a single fail value.

Calling the component **cgen**, its type can be described as:

```
cgen:: grammar * -> ([*]->bool)
```

The first parameter of **cgen** is the grammar, whose type is parameterised by the type of input tokens it describes. The result returned by **cgen** is a function from list of input tokens to a Boolean value which indicates if the input tokens conform to the grammar or not.

An elegant feature that Miramod inherits from Miranda is that the function **cgen** can actually be described and used both as a single parameter function that returns a parser or as a two parameter function which takes a grammar and some input and returns a Boolean result:

```
parser = cgen gram
result = cgen gram input
```

In fact the expression '`cgen gram input`' is interpreted as '`(cgen gram) input`', and any application of a function to several arguments is considered as a sequence of higher order functions applied to single arguments. As a result the type expressions

t_1 -> t_2 -> t_3 and
t_1 -> (t_2 -> t_3)

[9]A grammar that allows arbitrary length sequences must be recursively defined - in effect, infinite grammars must be allowed. Section 4.6 discusses this in more detail.

are equivalent and the type of cgen can be described as

```
cgen:: grammar * -> [*]->bool
```

As well as describing the behaviour of cgen the signature of grammar must be given.

```
type grammar *
with
    token    :: (*->bool) -> grammar *
    then, alt:: grammar * -> grammar * -> grammar *
```

The constructor token creates a grammar that recognises the single tokens defined by the given function. Using the function digit which is True for all the digit characters and False for any other character, the grammar that recognises all single digit input sequences can be expressed as 'token digit'. Alternatively individual characters can be recognised using the operator section notation 'token (='t')'[10].

One property that could be used to describe token is that when the parser generated by cgen from the grammar token tf is applied to a unit list [t] then the result is 'tf t':

```
    cgen (token tf) [t] = tf t
```

On its own, this property does not completely describe token (it only provides information on the behaviour of 'cgen (token x)' when applied to a unit list). This emphasises the point that it is not necessary to provide a complete description of the model - in fact the use of incomplete descriptions is a further way in which the model may be simplified.

[10](='t') denotes the function that returns True when given the character 't' and returns False otherwise. This notation is valid for any binary operator and produces a partially parameterised version of the operator.

The constructor `then` combines two grammars to produce the grammar which recognises sequences of the first grammar followed by sequences of the second. In other words, if the grammar `x` recognises sequence `p` and the grammar `y` recognises sequence `q` then the grammar '`(then x y)`' recognises the sequence `p++q`.

```
    cgen x p & cgen y q |- cgen (then x y) (p++q);;
```

Finally the constructor `alt`, which recognises sequences of either of its argument grammars, is described in a similar manner to `then`:

```
    cgen x p \/ cgen y p |- cgen (alt x y) p
```

Putting these parts together, the complete property model is written:

```
cgen:: grammar * -> [*]->bool

type grammar *
with
    token     :: (*->bool) -> grammar *
    then, alt:: grammar * -> grammar * -> grammar *

{tf t x y p q}
    cgen (token tf) [t] = tf t;;
    cgen x p & cgen y q |- cgen (then x y) (p++q);;
    cgen x p \/ cgen y p |- cgen (alt x y) p
```

Given that the component described by this model is reasonably large and complex, the size of the model is very modest.

The above example is intended as a demonstration of the techniques that can be used to reduce the model to a size that is sufficiently small for the purposes of component library retrieval[11]. These techniques are:

[11]Chapter 5 attempts to answer the question of what is sufficiently small.

Simplification of the model by the omission of functions and types that are not key parts of the component - for example the omission of an `optional` function from the grammar constructors.

Abstraction of the model by the replacement of collections of functions, collections of types or even collections of parameters with fewer, more abstract functions, types or parameters. An example of abstraction in the compiler model is the replacement of a grammar involving terminals, non-terminals, rules, sequences and alternatives with the three grammar constructors.

Weakening of property statements. One collection of property statements is weaker than another if it can be proved from, but cannot prove the other collection. The weakening of property statements does not involve the omission or replacement of functions and types, it simply involves saying less about them. Given a property statement that gives a precise description of the value of a function for certain forms of argument, the statement can be weakened in two ways. Firstly it can be changed so that it places less constraint on the value of the function (in other words it describes a set of possible values that could be returned). Alternatively it can be changed so that it places no constraint of the value of the function for some forms of argument. In the latter case this might well involve the removal of a complete property expression. In the compiler generator example, the properties of the model are weakened since they place no constraint on the value returned when a parse fails.

4.4 Comparing property models

If property models are to be used as a basis for component library retrieval, it is essential to have a method of comparing models that can be automated. The result of a comparison should be a measure of similarity rather than a straightforward 'match' or 'no match' answer. Since the models will almost certainly use different names for the functions and data types that they describe, two models can only be compared through the use of a mapping between the names of the models.

4.4.1 Model equivalence

A good starting point for model comparison is the test for equality. Two models are considered as equivalent if there is a one to one mapping between the functions and types of each model that maintains the consistency of types and allows the conjunction of all the properties in one model to be shown as logically equivalent to the conjunction of all the properties in the other model. The mapping is itself a logical expression, typically a conjunction of equalities between functions and types of each model.

To introduce some notation for describing model relationships, if `M` is a model or mapping consisting of a conjunction of property expressions then m_i is the i^{th} property expression in `M` and `M` in the context of a logical expression denotes the conjunction of all m_i in `M`. A more formal statement of the equivalence relationship is that two models `A` and `B` are equivalent if there exists a mapping `M` between models such that the conjunction of `A` and `M` is logically equivalent to the conjunction of `B` and `M`:

```
equiv A B = ∃M . (A & M) = (B & M)
```

One method of testing for this relationship is to use the properties of one model and the mapping between models to prove each of the properties in the other model and then repeat this process in the other direction. If all the properties can be proved in both directions then the models are equivalent. If model `A` has properties $a_1 \ldots a_n$, and model `B` has properties $b_1 \ldots b_m$ then the two models are equivalent under mapping `M` if and only if:

$$\begin{array}{c}
(\ a_1 \ \& \cdots \& \ a_n \ \& \ M \ \vdash \ b_1 \ ) \ \& \\
\vdots \\
(\ a_1 \ \& \cdots \& \ a_n \ \& \ M \ \vdash \ b_m \ ) \ \& \\
(\ b_1 \ \& \cdots \& \ b_m \ \& \ M \ \vdash \ a_1 \ ) \ \& \\
\vdots
\end{array}$$

$(b_1 \& \cdots \& b_m \& M \vdash a_n)$

where expression $a \vdash b$ is true if b can be proved from a.

The task of proving the properties of one model by taking the properties of the other model and the mapping equalities as axioms requires a theorem prover. For the remainder of this chapter, the existence of a suitable theorem prover is assumed. Chapter 5 describes the design of an experimental theorem prover which has been used to investigate the feasibility of the model approach to component library retrieval.

The equality relationship, as defined above, is unsatisfactory for the matching of component models because it provides a binary answer rather than the more continuous measure that is required and also because it is far too strict. Since the models being compared will have been produced by different people (a re-user on one hand and a librarian or component author on the other), it is likely that they will be at different levels of abstraction and based on different simplifications. One improvement on the equality relationship is to relax the requirement that properties have to be proved in both directions: if the properties of one model can be proved from the properties of the other and the mapping or vice versa then the models are considered to match. This allows the matching of two similar models where one describes more types and functions in more detail than another. This new matching relation can be defined as:

```
weakequiv A B = ∃M . (A & M ⊢ B) \/ (B & M ⊢ A)
```

Although this method of matching models helps to remove the dependency on models which are at the same level of simplification, it does not provide more than a binary answer. A further improvement therefore, is to count the number of property expressions proved as a percentage of the property statements which would need to be proved to demonstrate equality. This percentage can then be used as a measure of similarity, 100% indicating equality and 0% indicating no similarity. Unfortunately the accuracy of this measure depends heavily on the way in which component models are written. Property expressions vary according to their independence from other property expressions in the model and also according to the amount of information they provide about the model. For example, consider the properties of the `reverse` function:

```
{a x}
    reverse [] = []                  ;;
    reverse (a:x) = reverse x ++ [a]
```

Each of these properties expressions are independent but the second provides a great deal more information about reverse than the first since many equalities about the reverse of lists can be deduced from the second:

```
    reverse [a] = reverse [] ++ [a]
    reverse [a,b] = reverse [] ++ [b,a]
    reverse [a,b,c] = reverse [] ++ [c,b,a]
    ...
```

but no further equalities of the reverse of lists can be deduced from the first property expression. Using the measure of similarity introduced above, a model that implied the first property expression and not the second would produce the same matching value as a model that implied the second and not the first (provided the two models where equivalent in all other respects). Two possible methods of improving this measure would be to allow the model writer to provide information on the relative importance of property expressions, or to design an algorithm for estimating the relative importance. Neither of these alternatives have been investigated as part of the current research.

To continue with the same example, if the previous property statement concerning reverse is extended in the following manner:

```
{a x}
    reverse [] = []                  ;;
    reverse (a:x) = reverse x ++ [a] ;;
    reverse ([]++[]) = []
```

then neither the first nor third property expressions are independent, since the first is implied by the third and the known properties of ++ and the third is implied by the first and the known properties of ++. A consequence of this is that any model that implies the first will also imply the third, and the importance of what is actually a single property, will be magnified relative to the remaining property expressions. Rather than attempting to provide an automated check for independence, the experimental retrieval system described in this thesis relies on the production of independent properties by the model writer.

It is important to note that the matching algorithm distinguishes between the logical conjunction operator `&` and the symbol used to separate property expressions '`;;`'. Each operand of a conjunction is not treated as a separate property expression when calculating the match value, therefore the semantics of '`a1 & ... & an`' differ from those of '`a1 ;; ... ;; an`' when matching models.

4.4.2 Views between models

Although the mapping between models and the method of computing a match value described in the previous section allows models which are derived from different simplifications to be successfully compared, it will not successfully compare models containing different abstractions for the same component. For example, if two compiler generator models are compared and one model describes a grammar with an 'optional' constructor whilst the other does not, then a match can still be detected, even though it will not be a 100% match. On the other hand, if one model includes the compilation rules and the other does not, then no match will be found because the grammar rules will also be different (in the model that includes compilation rules, the grammar rules must allow the compilation rules to be attached).

To take this example further, consider the matching of the previously developed model for a compiler generator and the compiler generator model in figure 4.1. This new model includes compilation rules (the names have been changed so that the two models may be easily distinguished):

```
yacc::  * -> (ygram ** *) -> [**] -> *

type ygram ** *
with
    term:: (** -> *) -> ygram ** *
    sequen:: crule * -> ygram ** * -> ygram ** * -> ygram ** *
    choice:: ygram ** * -> ygram ** * -> ygram ** *

crule * == * -> * -> *

{fail f t}
    yacc fail (term f) [t] = f t
{fail f lg rg i a b}
    ok (yleft a)  & ok (yright b)
    |-
       exists [yacc fail (sequen f lg rg) (a++b)
                    = f (yleft l) (yright r) &
               l++r = a++b & ok (yleft l) & ok (yright r) &
               | (l,r)<-splits (a++b)]
    ;;
    yacc fail (choice lg rg) i
          = if (ok (yleft i)) (yleft i) (yright i)

    where
       yleft = yacc fail lg
       yright = yacc fail rg
       ok = (~=fail)
       splits l = [(take n,drop n)|n<-{0..#l}]
```

To paraphrase the second property expression: For all fail values **fail**, compilation functions f, ygrams lg and rg and input lists a and b. If lg successfully compiles a, and rg successfully compiles b, then there exists input sequences l and r such that: l++r=a++b; lg successfully compiles l; rg successfully compiles r; and the result of compiling the input a++b with sequen f lg rg and fail value fail is the compilation function f applied to the result of compiling l with lg and r with rg.
The expression '(l,r) <- splits (a++b)' is necessary because of the lack of a true existential quantifier. As the expression 'l++r=a++b' appears explicitly the list generator can be read as an existential quantifier.

Figure 4.1: A compiler generator model yacc

The type ygram (previously grammer) is a polymorphic type with two type variables, the first for the type of input tokens that will be accepted and the second for the type of the output produced by the compiler. The **sequen** constructor (previously the **then** constructor) has an additional parameter which is a compilation rule (type crule). If the two sub grammars of the **sequen** function successfully compile the input, then the given compilation rule will be applied to the results of the sub compilations and the resulting value will be returned. The compilation rule is also polymorphically typed, but with only a single variable for the compilers output type. It is described directly as a binary function on the output type. yacc (previously cgen) is also given an extra parameter, called the 'fail' value, which is produced when a grammar or compilation rule fails.

It is not possible to construct a mapping between this new model and the original model since the types of the grammar constructors and the function yacc are different. Despite this, the two models have strong similarities, especially if the type variable representing the compiler's result is replaced with the type bool, fail is replaced with 'False' and the compilation rule with (&)[12].

This correspondence between the two models can be formalised as a 'view'. A view is a collection of property expressions which relate the functions of one model to the functions of the other — it provides a view of one model in terms of the functions and types of another model. The view of cgen from yacc that has just been described could be written:

```
 cgen = yacc False
token = term
 then = sequen (&)
  alt = choice
```

Using this view and the properties of the yacc model, it is possible to prove the properties of the cgen model. Unfortunately this does not work in the other direction. The view actually contains an implicit type equality:

[12]Enclosing an operator in parenthesis without including any arguments converts it to a function.

```
grammar * == ygram * bool
```

which means that in conjunction with the properties of **yacc** only properties involving the type '`ygram * bool`' rather than the type '`ygram * **`' can be proved. Since the properties of **yacc** all involve the type '`ygram * **`', none can be proved from the view and **cgen**.

Implicit type equalities such as the one above are referred to as the the type part of the view, whereas equalities between expressions are referred to as the function part of the view.

This suggests a matching relation based on the existence of a view between the two models, the hypothesis model and the conclusion model, that allows the properties of the conclusion model to be proved from the properties of the hypothesis model. To compute a value for the match, the number of properties proved in a particular direction can be taken as a percentage of the number of properties to be proved in that direction and the higher of the two percentages considered as the match value. In the above example this would be 100% for proving **cgen** from **yacc** but 0% for the other direction, and thus the match value would be 100%.

Unfortunately the previous definition of a view as a collection of property expressions is too general for the matching relation described above. Any model could be proved from any other simply be assuming a sufficiently powerful view. The intention of a view is simply to translate between models and taken in isolation it should not imply or state any properties of the conclusion model. Such a view is referred to as a *permissible* view.

Both models and views can be characterised by the functions which satisfy them. A view is therefore said to be permissible with respect to a conclusion model provided that the functions which satisfy the conclusion model are a proper subset of those which satisfy the view (the formal definition of a permissible view is given in section 4.5).

In the general case, the question of view permissibility is undecidable, so instead, a number of computationally feasible restrictions are placed on views: Firstly they are restricted to property statements which are equalities between expressions containing the functions of one model on one side and the functions of the other model on the other side - functions from different models may not appear on the same side of the equality. Secondly, although a view which instantiates type variables in the hypothesis model is acceptable, one that instantiates type variables in the conclusion is unacceptable since this would weaken the conclusion and restrict the functions which satisfy it. Despite the fact that these restrictions are not sufficient to ensure that a view is permissible, they ensure that any view that is accepted will either be permissible with respect to a model or the model will be trivial.

4.4.3 View synthesis

Models are compared by proving the properties of one model from the properties of the other. If A and B are models and A is being proved from B then B is called the hypothesis model and A is called the conclusion model. Before this comparison can take place, a view between models which can be used to rewrite the conclusion properties in terms of the hypothesis properties must be synthesised.

This view is initially generated on the basis of the type information contained in each model. Functions with 'similar' types are equated in such a way as to take into account any difference between their types and ensure that the resulting view is correctly typed. The notion of similarity used is based on a collection of equalities involving some of the standard Miramod types and the generalisation relation between polymorphic types. The equalities used are those suggested by Mikael Rittri[52]:

$$\begin{aligned} (x,y) &= (y,x) \\ (x,(y,z)) &= ((x,y),z) \\ x->y->z &= (x,y)->z \\ x->(y,z) &= (x->y,x->z) \end{aligned}$$

In addition, any n-tuple (n>2) is considered equivalent to a collection of 2-tuples:

$$\begin{aligned}(a,b,c) &= (a,(b,c)) \\ (a,b,c,d) &= (a,(b,(c,d))) \\ &\vdots\end{aligned}$$

Each of these equalities has an associated pair of functions which can convert between values of the equated types, hence a view can be generated between functions whose types are 'similar' using these functions.

For example, the functions `f::a->b->c` and `g::(a,b)->c` can be equated with the view::

```
f = curry g
    where
        curry h a b = h (a,b)
```

The second part of the similarity relation used concerns the situation where one type is an instance of another. We say that a type `t` is a generalisation of a type `u` iff the type expression for `u` can be obtained from that of `t` by some consistent substitution of type expressions for variables. Since the intention is to prove properties of the conclusion model from those of the hypothesis, it is clear that hypothesis types may be generalisations of conclusion types but not visa-versa. In this case the view between functions `f::t` and `g::u` where u is a generalisation of `t` is simply `f = g`.

Thus we say that a hypothesis type `u` and a conclusion type `t` are 'similar' if there exists a type `v` which is equivalent to `t` under the above equalities, and `u` is a generalisation of `t`. In this case a view can be generated between the functions `f::t` and `g::u` using the conversion functions corresponding to the rules used to equate `v` and `t`.

Since the types being compared may contain names of described types, the type part of the view must be established before the function part. The comparison algorithm does this by first pairing every conclusion model type with a hypothesis model type in as many ways possible. The result of this is a mapping from conclusion types to hypothesis types (some of the hypothesis types may have no counterpart in the model type). Each mapping is then scored on the basis of the number of conclusion functions which have similar hypothesis functions, and the most succesfull mapping is then selected as the type part of the view.

The view between functions is then synthesised in the following manner. First of all, an equality is added to the view for any conclusion functions which are 'similar' to only one function in the hypothesis in only one way (ie. an equality is not included if it is one of many between the same pair of functions). The remaining functions are not included in the view initially but are marked as *free functions* of the appropriate type.

During the process of proving the properties of the conclusion model these functions may be instantiated with expressions containing only functions of the hypothesis model. When this occurs the instantiation is recorded as part of the view, where it has the form of an equality between the function and its instantiation. Since this instantiation may well be incorrect (i.e. it may prevent other properties of the conclusion model from being proved), it must be possible to backtrack and try other alternatives. In effect the free functions are existentially quantified variables over the whole model (ie "there exists functions ... such that, for all variables ... ").

Taking the compiler generator example once again, and trying to prove the `cgen` model from the `yacc` model, the `cgen` function `token` matches the type of `term` provided '`grammar * == cgram * bool`'. This type equality is acceptable because it does not restrict the type of the properties that need to be proved.

```
token :: (*->bool) -> grammar *
term :: (*->**) -> grammar * **
```

Under the same type equality both `then` and `alt` have a type that now matches `choice`:

```
then,alt:: grammar * -> grammar * -> grammar *
choice:: cgram * ** -> cgram * ** -> cgram * **
```

Since it is unlikely that the writer of a model would describe the same function twice, giving it different names each time, only one of `then` or `alt` should be equated with `choice`. Rather than make a choice at this stage, and potentially wasting effort in trying to prove the properties under an incorrect assumption, it is better to leave both functions free, in the hope that the attempt to prove properties about them will suggest a correct instantiation. `cgen` should also be left free, since its type does not directly match with any of the `cgram` functions (although it is close to `yacc`).

Due to the free functions, the order in which properties are proved is important, so the properties which contain the fewest free functions should be proved first. In the example, this heuristic suggests the property:

```
    cgen (token x) [y] = x y;;
```

Using the view developed so far, `token` can be rewritten with `term`. Since `cgen` is free, the first property of the `yacc` model can be used to rewrite the left hand side of this equation, at the same time as recording the instantiation of `cgen` as '`cgen = yacc f`' where `f` is free. The resulting equation is:

```
    x y = x y
```

Hence the first property is proved and the view developed further (although `cgen` is still not completely defined). Of the remaining two properties to be proved, both contain only one free function but the third has a marginally simpler structure. Using the newly acquired `cgen` equality in the view, the property can be rewritten as:

```
    yacc f x p \/ yacc f y p |- yacc f (alt x y) p
```

bearing in mind that `f` is free. The consequence of this implication can be rewritten using the third property of the yacc model and instantiating `alt` to `choice`:

```
yacc f x p \/ yacc f y p |-
      if ((~=f) (yacc f x p)) (yacc f x p) (yacc f y p)
```

Now there are two possible cases, either '`yacc f x p`' returns `True` or it returns `False`. One way of proving the above property is to prove it for both cases (since they are mutually exclusive)[13].

CASE `yacc f x p = True`

Substituting `True` for `yacc f x p` gives

```
True \/ yacc f y p |- if ((~=f) True) True (yacc f y p)
```

which reduces to

```
if (~f) True (yacc f y p)
```

which follows from

```
~f
```

hence the case is proved if `f` is instantiated to `False`.

CASE `yacc f x p = False`

Substituting `False` for `yacc f x p` and `False` for `f` gives

```
False \/ yacc f y p |- if ((~=False) False) False (yacc f y p)
```

which reduces to

```
yacc f y p |- yacc f y p
```

proving the second case.

Thus the third property of the conclusion model `cgen` is proved. The view is now:

```
token = term
cgen = yacc False
alt = choice
```

[13]This ignores the possibility that the expression could be undefined. Such issues are discussed in the following section as well as chapter 5.

Two out of three of the property expressions have now been proved. The final one however, is the most complex. To start with, now that '`yacc fail`' is always referred to as '`yacc False`' in the properties that need to be proved, the hypothesis model could have its properties specialised by replacing `fail` with `False`. This simplifies the definition of the local function `ok` to '`ok x = x`' and thus the second property expression becomes:

```
{f lg rg i a b}
    yacc False lg a  & yacc False rg b
    |- exists [yacc False (sequen f lg rg) (a++b)
                      = f (yleft l) (yright r) &
               l++r = a++b & ok (yleft l) & ok (yright r) &
               | (l,r)<-splits (a++b)]
```

Now since the list comprehension asserts that for all members of the list '`yacc False lg l`' and '`yacc False rg r`' are both true, their occurrences in the following equation can be replaced with the value `True`.

```
    yacc False lg a & yacc False rg b
    |- exists [yacc False (sequen f lg rg) (a++b) = f True True &
              l++r = a++b
              | (l,r) <- splits (a++b)]
```

The resulting equality within the list comprehension no longer contains any of the existentially quantified variables, so it may be 'floated' outside the scope of the (pseudo) existential quantifier. Having done this the remaining `exists` and list comprehension may be removed altogether (`H |- C1 & C2` proves `H |- C1`) giving the following property of `yacc` and `sequen`:

```
    yacc False lg a & yacc False rg b
    |- yacc False (sequen lg rg) (a++b) = f True True
```

The second property of `cgen` can finally be proved:

```
    cgen x p & cgen y q |- cgen (then x y) (p++q)
```

Using the view, this converts to:

```
    yacc False x p & yacc False y q |- yacc False (then x y) (p++q)
```

One way to prove an implication is to assume the hypothesis and then prove the conclusion. In this case, assuming the hypothesis means that the conclusion can be rewritten using the conclusion of the yacc property that has just been derived. This also involves instantiating the remaining free function with '`sequen f`' where `f` is itself free. The resulting expression that must still be proved is:

```
   f True True
```

Since `f` is free, the equation

```
   f True True = True
```

can simply be assumed as part of the definition for `f`, hence the last property is proved and the completed view of `cgen` is:

```
    token = term
    cgen = yacc False
    alt = choice
    then = sequen f
    f True True = True
```

This view is permissible because although it has defined a function local to the view (`f`), the equality that describes the function does not involve any `cgen` functions.

This example demonstrates the method of synthesising views between functions[14]. Firstly the types of functions are used to create the view between similarly typed functions. The remaining functions are left free and the view for these functions is discovered by the theorem prover. The theorem prover achieves this by attempting the proof of properties about functions for which it already has views before the properties of functions for which it has no view. In the case of properties not distinguished by this criterion, the proof of the structurally simplest theorem is attempted first. During the proof, free functions may be instantiated provided the right hand side of the resulting equality does not contain any of the functions of the model being viewed (including recursive references to the function which is being instantiated).

4.5 Property Model and View Semantics

It is beyond the scope of this thesis to provide a full semantics for Miramod and the views between models, not least because this would require a semantics for Miranda, which has not yet been published. Despite this, it is instructive to investigate models for Miramod by considering a more abstract version of the language.

Abstract Miramod

The abstract version of Miramod is referred to as μ. In the language μ, property models are built from property expressions using the '$;;$' constructor and the empty property model E. Each property expression is represented as an n-ary predicate over the names of free functions which occur in that property expression. Since a view is also a collection of property expressions, we use exactly the same abstraction for views as for property models. Our language also contains symbols which can be used to talk about property models and views. If a, b and v are property models, then $a\&v$ is a property model obtained by combining the view v with model a, $a \leadsto b$ is a sentence which is true iff the properties of b can be derived from those

[14]The property prover is not capable of performing this proof directly — the **yacc** model would need to contain appropriate heuristic information before this proof could be performed.

of a, $Q(v, b)$ is true iff the view v is permissible with respect to b and $a > b$ is true iff a 'matches' b.

Syntax

The language μ is described by giving the symbols of the language and the rules with which the symbols may be put together to form well formed property expressions, well formed property models, and well formed formulas:

1. A countable set of symbols:
   (a) Function names, written $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots, \mathsf{z}$.
   (b) A countable set of predicate letters: P^k_n where k and n range over the positive integers.
   (c) Operators $;;$, $\&$, $\leadsto$ and $>$.
   (d) Permissibility relation Q.
   (e) The empty property model E.
   (f) The auxiliary symbols:) and (.
2. The class of well-formed property expressions consists of:
   (a) $\mathsf{P}^k_n(f_1, \ldots, f_n)$ where k and n are positive integers, and $f_1, \ldots, f_n$ are function names.
3. The class of well-formed property models consists of:
   (a) The empty property model E.
   (b) Compound property models $p;;ps$ where p is a well-formed property expression and ps is a well formed property model.
   (c) Conjoined property models $a\&b$ where a and b are well-formed property models.
4. The class of well-formed formulas (wffs) consists of:
   (a) Compound formulas (a), $a \leadsto b$, $a > b$ and $Q(a, b)$ where a and b are well-formed property models.

A model for the language μ

In order to interpret μ property expressions, property models and formulae, it is necessary to fix a model for μ. This model is centered around the representation of property expressions and property models. An assignment of names to continuous functions is represented by a set of tuples. In general, a property expression does not give a complete description of its free functions, and so there will be more than one assignment possible. The model for μ property expressions is therefore a set of assignments.

Since a complete property model also describes its free functions, the model for a property model will also be a set of assignments.

The language μ contains predicates, and so the model contains a set of relations onto which these predicates can be mapped.

The model therefore consists of the following parts:

1. The set $\mathcal{N}$ of objects (names) $\mathcal{A}$,$\mathcal{B}$,$\mathcal{C}$... $\mathcal{Z}$.
2. The set of continuous functions $\mathcal{C}$.
3. A countable set of relations R_n^k on $\mathcal{C}$ where n and k range over the positive integers.
4. A set of assignments where each assignment is a set of name / continuous function pairs.
5. The values T and F.

The free functions of an assignment are the names which appear as part of the assignment. If a and b are assignments then they are compatible ($a \overset{*}{\simeq} b$) iff a and b assign the same continuous functions to their common free functions.

$$a \overset{*}{\simeq} b \overset{def}{\Longleftrightarrow} \forall{<}n,f{>}\in a \;.\; \forall{<}m,g{>}\in b \;.\; n = m \rightarrow f = g$$

The Semantic Function

The values of μ property expressions, property models and formula under the semantic function (written $[\![\ ]\!]$) are now defined recursively:

- Each function name in μ is mapped to the corresponding name in the model:

$$\begin{aligned} [\![a]\!] &= \mathcal{A} \\ [\![b]\!] &= \mathcal{B} \\ &\vdots \\ [\![z]\!] &= \mathcal{Z} \end{aligned}$$

- Each predicate symbol is mapped onto the corresponding relation in the model, so for n and k ranging over the positive integers:

$$[\![\mathrm{P}^k_n]\!] = R^k_n$$

- The semantic function for μ property expressions can be defined in terms of the semantic function for predicate symbols and function names.

  For n and k ranging over the positive integers:

$$\begin{aligned} [\![\mathrm{P}^k_n(f_1, \ldots, f_n)]\!] = \{ & \{<[\![f_1]\!], c_1>, \ldots, <[\![f_n]\!], c_n>\} \mid \\ & <c_1, \ldots, c_n> \in [\![\mathrm{P}^k_n]\!] \} \end{aligned}$$

- The empty property model $\mathtt{E}$ is simply mapped to a set containing only a single empty assignment — it places no restriction on the functions that will satisfy it.

$$[\![\mathtt{E}]\!] = \{\{\}\}$$

- When adding a property expression p to a property model ps, each assignment from p is combined with each assignment from ps provided they are compatible.

  For all μ property expressions p and property models ps:

$$[\![p;;ps]\!] = \{p' \cup ps' \mid p' \in [\![p]\!] \ ; \ ps' \in [\![ps]\!] \ ; \ p' \overset{*}{\simeq} ps'\}$$

- As with the property model constructor ;;, when conjoining property models a and b, each assignment from a is combined with each assignment from b provided they are compatible.

  For all μ property models a and b:

$$[\![a\&b]\!] = \{a' \cup b' \mid a' \in [\![a]\!] \ ; \ b' \in [\![b]\!].; \ a' \overset{*}{\simeq} b'\}$$

- The derives relation $a \leadsto b$ has to be true iff the properties of b can be derived from those of a. If this is the case then each of the assignments in the interpretation of a should be stronger than an assignment in the interpretation of b.

$$[\![a \leadsto b]\!] = \forall a' \in a \ . \ \exists b' \in b \ . \ a \supseteq b$$

- The permissibility relation $Q(v,p)$ ensures that the view is not too powerful with respect to the free functions of *each* property expression in p. It insists that the assignments of p are a restricted version of the assignments for p's free functions allowed by v.

$$\begin{aligned} [\![Q(v,\mathtt{E})]\!] &= T \\ [\![Q(v,p;;ps)]\!] &= v \overset{*}{\supset} p \wedge [\![Q(v,ps)]\!] \end{aligned}$$

The restriction relation $\overset{*}{\supset}$ is defined so that $v \overset{*}{\supset} p$ iff the following two conditions hold. Firstly, for every assignment in p there must be a compatible assignment in v and secondly there must be an assignment in v which is incompatible with or has no free functions in common with any of the assignments from p.

$$\begin{array}{rcl} v \overset{*}{\supset} p & \overset{def}{\Longleftrightarrow} & \forall p' \in p \,.\, \exists v' \in v \,.\, v' \overset{*}{\simeq} p' \wedge \\ & & \exists v' \in v \,.\, \forall p' \in p \,.\, (v' \overset{*}{\not\simeq} p' \vee v' \cap p' = \{\}) \end{array}$$

- The property models a and b match provided there exists a view (which is permissible with respect to b) such that the conjunction of the view and a is sufficient to derive b. If the language μ contained variables and existential quantification, the semantic function for $>$ could simply be defined in terms of $\rightsquigarrow$ and Q:

$$[\![a > b]\!] \;=\; [\![\exists v.\ Q(v, b) \wedge a \& v \rightsquigarrow b]\!]$$

However, since existential quantification is not available in μ, the permissibility and derivability conditions must be described directly in the model. This results in a fairly complex definition.

For all positive integers n:

$$\begin{array}{rcl} [\![a > \mathrm{E}]\!] & = & T \\ [\![a > p_1;;\ldots;;p_n;;\mathrm{E}]\!] & = & \exists v.(\ v \overset{*}{\supset} [\![p_1]\!] \wedge ([\![a]\!] \cap v) \subseteq [\![p_1]\!] \wedge \\ & & \vdots \\ & & v \overset{*}{\supset} [\![p_n]\!] \wedge ([\![a]\!] \cap v) \subseteq [\![p_n]\!]) \end{array}$$

Each line imposes two conditions on one of the property expressions $p_1 \ldots p_n$. The first condition is that the view v must be permissible with respect to the property expression ($v \overset{*}{\supset} [\![p_i]\!]$). The second is that the property expression must follow from the conjunction of property model a and the view v $((\llbracket a \rrbracket \cap v) \subseteq \llbracket p_i \rrbracket)$.

4.6 Infinite Structures and the Undefined Value

It is sometimes necessary to describe the situations in which a function or data type does not denote a well-defined value in the normal mathematical sense. For example, part of the description of the 'divide' function might state that the expression '`1/0`' does not denote a well-defined value. To facilitate this, a special 'undefined' value is added to the language. This value is denoted **`undef`** and belongs to every type. The undefined value can also be used to describe the termination properties of a function; an expression that does not terminate when evaluated has no well defined value and is therefore equivalent to **`undef`**.

The existence of **`undef`** and *partial functions* (functions that sometimes return **`undef`**) add considerably to the complexity of Miramod's semantics. For this reason Miramod is designed so that property models can be written without paying attention to the semantics of **`undef`**, and yet partial functions may be described in detail if necessary. To this end, the distinction between partial functions (partially defined functions) and partially described functions is important. A partial function has some combinations of input values for which the output is the undefined value **`undef`**; on the other hand a partially described function has combinations of input parameters for which the output value has not been described but may be a well-defined value.

4.6.1 The semantics of the undefined value

If partial functions are to be described in property models, the semantics of **`undef`** in terms of the predefined functions and types as well as fundamental property statement constructors (`|-` and `=`) are important. One aspect of **`undef`**'s behaviour is the result of applying a function to **`undef`** (since **`undef`** is a member of any type, any function may be applied to `undef`). In many languages this behaviour is the same for all functions; they all return the undefined value if applied to the undefined value (`f undef` = `undef`). Functions which exhibit this behaviour are called *strict* functions. By contrast, functions which return a well-defined value when applied to `undef` are called *non-strict* functions (`f undef` $\neq$ `undef`).

Functions described and defined by Miramod are not constrained to be strict. For example:

```
False & x = False;;
1 + undef = undef
```

The first property expression states that the result of applying `&` to `False` and anything (including the undefined value) is `False`, so the function `(False &)` is non-strict whereas the function `(1+)` is strict. An alternative way of putting this is to say that the operator `&` is non-strict in its second argument provided its first argument has the value `False` and the operator `+` is strict in its second argument when its first argument is `1`.

The strictness of a defined function follows directly from the function definition and strictness of any other functions occurring in the definition. If the value of a parameter is needed before the function can return even a partial result, then the function is strict in that parameter.

```
(&) False x = False
(&) True x = x

(\/) True x = True
(\/) False x = x

(~) True = False
(~) False = True
```

Given the above definitions, it is clear that all three operators are strict in the first parameter as this must be evaluated before deciding which defining equation applies. The `&` operator is non-strict in the second parameter whenever the first parameter has the value `False` since `False` is returned regardless of the value of `x`. Likewise the second parameter of `\/` is non-strict when the first parameter is `True`.

The existence of an undefined value forces the use of two distinct equality relations; *weak equality* and *strong equality*. These are defined by the following truth tables (T represents true or `True`, F represents false or `False`, E_1 to E_n represent the $n > 0$ distinct values of the type being compared and $\perp$ represents `undef`):

Strong Equality

==	E_1	E_2	...	E_n	$\perp$
E_1	T	F	...	F	F
E_2	F	T		F	F
$\vdots$	$\vdots$		$\ddots$		
E_n	F	F		T	F
$\perp$	F	F		F	T

Weak Equality

=	E_1	E_2	...	E_n	$\perp$
E_1	T	F	...	F	$\perp$
E_2	F	T		F	$\perp$
$\vdots$			$\ddots$		
E_n	$\vdots$	F		T	$\perp$
$\perp$	$\perp$	$\perp$		$\perp$	$\perp$

The weak equality operator is strict in both arguments, so if either argument is undefined then the value `undef` is returned. Furthermore any attempt to compare functions using weak equality results in the undefined value. Due to its executable semantics, Miranda allows only weak equality in expressions. This is because the equality operator can only be implemented for types which have a canonical representation. Since Miramod is descriptive rather than executable, the most common use of equality is in its denotational (or mathematical) sense: i.e. strong equality. Unfortunately it is also desirable to have access to weak equality within property expressions. For example, consider the description of a function that compares three values for equality:

```
eq3:: * -> * -> * -> bool
{x y z}
    eq3 x y z = (x=y & y=z)
```

The leftmost equality should be strong since it asserts a property. The right hand pair of equalities must not be strong otherwise the function eq3 would be un-implementable. The difference between these equalities is that the leftmost forms a property expression whilst the right hand equalities both form Boolean expressions. Generally speaking, equalities which form property expressions should be strong equalities since the property expression is either true or false. On the other hand equalities which form Boolean values should be weak so that un-implementable functions are not accidentally described.

Since property expressions are either true or false but may not be **undef**, they are given a separate type from Boolean expressions which may be **undef**. This type can be thought of as:

```
property ::= T | F
```

The shells T and F are hidden and may not appear in property statements (unless they are describe as a separate algebraic type). The respective types of strong and weak equality and inequality are therefore:

```
(=),(~=):: * -> * -> property
(=),(~=):: * -> * -> bool
```

When they appear in property expressions the result type of either operator is used to determine whether the strong or weak version applies. If there is insufficient type information to make this choice then the weak version is used. An alternative notation for strong equality and inequality can be used to make their distinction from weak equality and inequality clear:

```
(==),(~==):: * -> * -> property
```

The semantics of the 'proves' operator |- (see fig 4.2) do not involve the undefined value since the type property is always well defined and |- has type:

|-	T	F
T	T	F
F	T	T

Figure 4.2: 'proves' truth table

```
(|-):: property -> property -> property
```

Finally, the syntax of a property expression allows a Boolean expression to appear at any point where a property expression is allowed. Since the types are different, a *coercion* from `bool` to `property` must take place. This is achieved by the function `(True==)` so in effect the appearance of a Boolean expression `b` in the place of a property expression is shorthand for '`True == b`'.

This raises a potential ambiguity when an equality occurs as a property expression. One interpretation is that the quality is strong and the type `property` is returned. The other interpretation is that the equality is weak and returns a Boolean value which is then coerced to a property value. This ambiguity is resolved by the rule that Boolean expressions occurring in the place of property statements may not be equalities.

Using the compiler generator `cgen` as an example, the property:

```
cgen x p & cgen y q |- cgen (then x y) (p++q)
```

is shorthand for:

```
(True == cgen x p & cgen y q) |- True == cgen (then x y) (p++q)
```

As it stands this property expression describes very little of the partial behaviour of `cgen`. If either '`cgen x p`' or '`cgen y q`' are undefined then the hypothesis is false and nothing can be concluded about `cgen (then x y) (p++q)`. If they are both `True` then the conclusion that `cgen (then x y) (p++q) ~== undef` is valid. Information about the strictness of `cgen` can easily be added to the description:

```
    cgen undef g = undef
    cgen g undef = undef
```

4.6.2 Infinite and Partial Structures

The property statements of Miramod can be used to describe infinite structures and *partial structures* as well as functions that will operate on these structures (a partial structure is one which is at least partially defined but contains an undefined value). For example the infinite list of ones and the partial list containing one element can be described:

```
{}
    infinite == 1:infinite          ;;
    partial == 1:undef
```

Both `infinite` and `partial` are distinct from the undefined value:

```
    infinite ~== undef;;
    partial ~== undef
```

Any attempt completely to evaluate a partial or infinite structure will result in the undefined value. So for example: '`#infinite == undef`' and '`(partial = partial) == undef`' (The `#` operator gives the length of a list). On the other hand, many functions are well-defined even over partial or infinite parameters. For example the function `hd` (which returns the head of the list) applied to either `partial` or `infinite` returns 1:

```
    hd infinite == 1
    hd partial == 1
```

4.6.3 Property Variables

Since values which are undefined, partial or infinite are permitted it is important to define if the universally quantified variables of a property statement are allowed to be undefined, partial or infinite. A property statement whose variables included the undefined value as well as partial and infinite values is stronger than an identical property statement whose variables do not include these values. In keeping with the aim of allowing property models writers to ignore issues of undefined behaviour and infinite structures, the default case is to assume that variables are both finite and completely defined (unless they have a function type). This means that the weaker statement is assumed by default and the probability of a naive property writer asking for unintended partial behaviours is reduced. For example, the property statement

```
{x}
   x::[*]
   reverse (reverse x) = x
```

is interpreted as "for all `x::[*]` such that `x` is completely defined and finite, ...", and is a correct description of the function that reverses lists. If a variable is intended to cover undefined, partial and infinite lists then it should be followed by the `+` symbol. The model:

```
{a+ x+}
    reverse (a:x) = reverse x ++ [a]
```

is therefore interpreted as "for all `x::[*]` and `a:*`, ...".

The notation of Miramod therefore allows simple models to be written quickly and easily without directly involving the writer in the deeper issues of the partial behaviour of the model. At the same time the model writer may, if he or she chooses, describe the partial behaviour of the model to whatever level of detail they require.

4.7 Summary

The language Miramod is motivated by the need for a formal method of describing components. Its similarity to the notation of Miranda is justified by the fact that the components Miramod is intended to describe are primarily Miranda components. Miramod extends Miranda in two key areas. It provides property statements for describing rather than defining functions, and it provides a more complete notation for describing algebraic types than is provided by Miranda.

As a method of describing components for the purposes of component library retrieval, Miramod meets its objectives in the following ways:

- It gives a formal description of the component and hence it does not rely on names.
- Its lack of dependency on names provides the possibility of re-use across application domains.
- It allows the re-user to describe components at the level of detail they require.
- It is potentially capable of high precision and high levels of completeness; however the achievement of this potential is dependent on the success of the theorem prover in comparing models and the ability of the user to make and formalise 'good' abstractions for components.

Miramod has several shortcomings from the point of view of component description. These shortcomings derive from the need to automate the comparison of Miramod models. They are: the lack of full (arbitrarily positioned) existential and universal quantifiers, the restriction that local functions must be defined and not described and the restriction that types described using the **abstype** notation must be given an associated retrieve function and validity function.

Models may be compared by proving the properties of one model from the properties of the other. This proof is based on a view which consists of a set of equalities which relate the functions of one model to the functions of the other model. To allow the matching of models with similar but not identical functions, these equalities may consist of an arbitrary expression on either side, but one expression must contain only functions of one model and the other expression must contain only functions of the other model.

Finally, this view between models can be discovered using type information to equate some functions and also by leaving other functions 'free' so that relationship with the other model can be established by the theorem prover.

Chapter 5

The property prover

The model-based approach to component library retrieval relies on the existence of a method for comparing models that can be automated and leads to high precision and low loss. This method has been outlined in chapter 4. The current chapter describes the method in greater detail as well as describing the design of a prototype 'property prover' which implements the comparison of property models.

The property prover is intended as a prototype for investigating model based component library retrieval. The principal questions it is intended to help answer are:

- What is the relationship between a model produced by a library user and the models of potentially re-usable components? In other words, what is a good matching relation to use for component library retrieval?
- Assuming that a matching relation will be based on views, what views should be allowed between similar components? More specifically, what defines a view that is too restrictive (i.e. prevents a reasonable match because the view is not flexible enough to reconcile the differences between the models) and what defines a view that is too general (i.e. allows totally different models to match)?

- How feasible is model based component library retrieval?

These questions cannot be answered by a theoretical analysis since they are highly dependent on human factors as well as technical ones. They rely on unpredictable factors such as the contents of the library, the nature of components generally required by the users of the library, the ability of users to write property models for components and the ability of the librarians and component contributors to write property models for the components in the library.

The property prover described in this chapter is part of the prototype component library retrieval system which is intended to answer these questions within a limited context (ie Durham Miranda library and the staff and students of the Computer Science Group at Durham). The theorem prover has the following objectives:

1. To compare models and give an indication of their similarity.
2. To establish a view between models
3. To check if the properties of one model follow from the properties of another model, the view and the axioms of the language.

5.1 Limitations

There are a number of theoretical and practical limitations to the property prover. Firstly the question of whether or not two models are similar is undecidable in the general case. More specifically the question of whether or not a particular property can be proved from a model is undecidable. This follows from the undecidability of the halting problem which states that:

> There is no effective procedure (or executable function that always terminates) that can take any function definition `fd` and any input `i` of the correct type and decide whether or not the function will terminate

(halt) when applied to that input (in other words prove or disprove '`f i == undef`' where `f` is the function defined by `fd`).

For two functions to be equivalent they must both produce the same output values for corresponding inputs, and must also fail to terminate for the same inputs. Since the latter condition is the halting problem, which is undecidable, the question of function equivalence must also be undecidable. Not surprisingly the same result holds for the question of whether a property can be proved from the properties of another model. If the property is in effect a statement of the halting problem '`f i == undef`' then there is no effective procedure for deciding if this property holds. This does not mean that such a property can never be proved automatically, it simply means that for any algorithm which attempts to decide '`f i == undef`' there will be combinations of `f` and `i` for which it will either fail to terminate or produce an incorrect result.

When comparing models for component library retrieval, it is clearly preferable to produce an incorrect result (with a resulting loss in precision or completeness of retrieval) than for the retrieval system not to produce any result (fail to terminate). This contrasts with more conventional applications of theorem provers, such as verification in software engineering and proof checkers for mathematicians, where the possibility of non termination is acceptable but the possibility of an incorrect result is not.

A property prover that always terminates can do one of three things. It can attempt to prove a property and return 'proved' if it succeeds and 'not proved' if it does not. Alternatively it can attempt to disprove the property and return 'disproved' if successful and 'not disproved' otherwise. The third possibility is to attempt to prove and disprove the property; if neither is successful then an 'unsuccessful' value is returned otherwise a 'proved' or 'disproved' value is returned. In terms of measuring the similarity of models the three possible results could each correspond to a score which contributes to the overall matching score of the two models. Experience of the experimental Miranda library at Durham has indicated that is easier and more natural to describe component models in terms of equalities than inequalities[1], it follows that trying to prove properties which are usually equal-

[1]there is also considerable support for this method in the literature [24])

ities, from a model made up of equalities is usually easier than trying to disprove properties under the same circumstances.

If model based component library retrieval is to be feasible, the loss of retrieval precision and completeness resulting from this limitation of the property prover must be acceptable. The issue is complicated by the fact that there is a tradeoff between the maximum time taken to obtain a result and the probability of an incorrect result. A 'fast' property prover might be unacceptable on the grounds of inadequate precision and completeness whereas a 'slow' property prover might be unacceptable on the grounds that it takes too long to search the library. The existence of more than one property in the conclusion model helps overcome this limitation of the property prover — the effect of an incorrect result is reduced by the presence of additional properties (provided these produce correct results) since the overall matching value is based on the results of attempting to prove all the properties and not just one.

A second limitation to the property prover is the lack of any formally defined semantics for Miramod (or Miranda). This means that there is no formal specification for the property prover and therefore the property prover itself cannot be guaranteed correct. As with the decidability limitations of the property prover, this limitation does not greatly affect the property prover's ability to meet its objectives, since these involve detecting similarities rather than proving equivalence.

5.2 Review of Theorem Proving Techniques

The central goal of automated theorem proving and automated deduction is to make it possible for computers to draw conclusions from sets of facts. The need for deduction arises in situations where the problem description is incomplete in the sense that some questions cannot be answered by straightforward evaluation.

Notation

In the remainder of the thesis, logical statements that are not part of the notation accepted or used by the retrieval system are written using the following more elegant mathematical equivalents:

Miranda or Miramod	Equivalent	Meaning
`\/`	$\vee$	logical disjunctions
`|-`	$\vdash$	proves
`Undef`	$\perp$	the undefined value

The symbol $\rightarrow$ is also used to denote logical implication.

5.2.1 Resolution Theorem Proving

One of the most commonly used methods of theorem proving is based on the *resolution* rule of inference. This states that if:

$$Q \vee P \,\&$$
$$\tilde{}Q \vee R$$

then

$$P \vee R$$

Since $\tilde{}A \vee B$ is equivalent to $A \rightarrow B$ the resolution rule is in fact a restatement of the chain rule of inference, which is written:

$$(S \rightarrow Q) \ \& \ (Q \rightarrow R) \vdash S \rightarrow R$$

(having replaced $Q \vee P$ with $P \vee Q$ and P with $\tilde{}S$).

The resolution method introduced by J.A.Robinson [54] proves a theorem from a set of axioms by applying the resolution rule to the negated theorem and the axioms until a contradiction is reached. The axioms and negated theorem are represented in *clause form* which is a conjunction of *clauses*. Each clause is a disjunction of *literals* where a literal is either an atomic formula (proposition in propositional calculus or predicate in predicate calculus) or the negation of an atomic formula.

$$(L_{1,1} \vee L_{1,2} \ldots \vee L_{1,n})\ \&$$
$$(L_{2,1} \vee \ldots)\ \&$$
$$\ldots$$
$$(L_{m,1} \vee L_{m,2} \ldots \vee L_{m,o})\ \&$$

The general resolution rule is used to resolve clauses. It differs from the resolution rule given above in that it can resolve any two clauses provided that one clause contains the negation of an atomic formula from the other clause. The result is a clause containing the literals of both clauses without the matching atom:

$$\frac{\begin{array}{c} A_1 \vee .. \vee A_n \vee Q\ \& \\ B_1 \vee .. \vee B_m \vee {}^{\sim}Q \end{array}}{A_1 \vee .. \vee A_n \vee B_1 \vee .. \vee B_m}$$

The general resolution rule is repeatedly applied to two of the clauses to produce a new clause — the *resolvent*. This is then added to the set of clauses and the process continued recursively on the new set of clauses. If the empty clause can be produced (by resolving A and ~A) then the theorem is proved by contradiction.

For example, supposing the theorem $A \rightarrow D$ is to be proved from the axioms $A \rightarrow (B\ \&\ C)$ and $C \rightarrow D$. Firstly, the axioms are put into clause form (each clause is numbered so that the resolution process can be described):

1) $^{\sim}A \vee B$
2) $^{\sim}A \vee C$
3) $^{\sim}C \vee D$

The theorem to be proved is then negated and converted to clause form, giving the two clauses:

4) A
5) $\tilde{}D$

General resolution is now used to produce an empty clause (a contradiction) by firstly resolving 2 and 4 to give:

6) C

and then resolving 6 and 3 to give:

7) D

Finally 7 and 5 are resolved to give the empty clause, and the theorem $A \rightarrow B$ is proved by contradiction.

The method of resolution theorem proving can be used to prove theorems expressed in *propositional logic* in which the atomic formula are in fact propositions (e.g. "John is a man"). However it also applies to first order predicate calculus which allows predicates, functions, variables and quantifiers. For example the proposition "John is a man" can be expressed as $man(John)$ and the proposition "For all x, if x is a man then x is human" can be expressed as $\forall x.man(x) \rightarrow human(x)$. Given these two predicate calculus clauses as hypotheses the conclusion $human(John)$ can be reached, but in propositional logic "John is a human" does not follow from the above propositions since their content is not relevant.

Two extensions to the resolution method introduced above must be made to apply resolution theorem proving to predicate calculus formulae. Firstly the predicate calculus must be transformed to clause form, and secondly the resolution rule must allow for the *unification* of the arguments of predicates. The conversion of predicate calculus to clause form is formally straightforward (see [13] for details), the most interesting aspect being the replacement of existentially quantified variables with skolem constants and functions. For example the formula $\exists H.\mathit{husband}(\mathit{Jill}, H)$, which states that Jill has a husband, can have the existential quantifier removed and the variable H replaced by a unique skolem constant to give $\mathit{husband}(\mathit{Jill}, k)$. The fact that k is a unique constant indicates that Jill's husband exists without giving any clue as to who he is. If the existential quantifier occurs within a universal quantifier then it must be replaced by a skolem function applied to the universally quantified variable. For example, in the formula:

$$\forall X.\exists Y.\mathit{married}(X) \ \& \ \mathit{female}(X) \rightarrow \mathit{husband}(X, Y)$$

which states that for all X there exists a Y such that if X is married and X is female then Y is X's husband, the existentially quantified variable Y must be replaced by a skolem function applied to the universally quantified variable X.

$$\forall X.\mathit{married}(X) \ \& \ \mathit{female}(X) \rightarrow \mathit{husband}(X, f(X))$$

This ensures that the hypothesis does not insist that all married women have the same husband (the skolem constant).

Once all the existentially quantified variables have been removed (and all universally quantified variables moved to the outermost level of the formula), the remaining universal quantifications can be removed since any remaining variables must be universally quantified. The resulting formula, in clause form, is:

$$\tilde{}\mathit{married}(X) \vee \tilde{}\mathit{female}(X) \vee \mathit{husband}(X, f(X))$$

The second extension needed for resolution to apply to first order predicate calculus is the ability to unify predicate arguments during the application of the resolution rule. Continuing with the previous example, it should be possible to prove that Jill has a husband from the above clause and the assertion that Jill is female and married. In other words the following clauses should lead to a contradiction:

1) $\tilde{}$ $married(X) \vee \tilde{}female(X) \vee husband(X, f(X))$
2) $married(Jill)$
3) $female(Jill)$
4) $\tilde{}$ $husband(Jill, Y)$

To make any progress with this proof it must be possible to substitute variables and values for other variables. For example, clauses 1 and 4 could be resolved if *Jill* was substituted for X and $f(Jill)$ was substituted for Y.

These substitutions are achieved by unification of the two predicates begin resolved. Unification performs two tasks; firstly it decides if predicate arguments are comparable, and secondly if they are comparable it gives the appropriate substitution. This substitution is then applied to the whole of the resolvent (but not to the clauses resolved). In the example this gives:

5) $\tilde{}married(Jill) \vee \tilde{}female(Jill)$

Which gives a contradiction when resolved with clause 2 then 3.

The principal advantages of resolution as a method of automated deduction is that it is *complete* for first order predicate calculus since it can prove all true theorems. It is also *sound* because it can never prove a non theorem true. Another advantage is the simplicity achieved because the method is based on a single rule of inference — the resolution rule.

The main disadvantage is that the search space generated by the resolution method grows exponentially with the number of clauses used to describe the problem, so that proofs of even moderate complexity cannot be found in reasonable time. A further problem is that it does not produce understandable proofs since there is no distinction between goals and antecedents.

There are a number of elaborations to the basic method of resolution theorem proving and these are generally aimed at reducing the combinatorial explosion of the search space. The set-of-support strategy insists that one parent of each resolvent is from the original negated hypothesis or one of the clauses derived from it thus restricting the number of resolutions that are applicable at any one time and reducing the search space [61]. Greater reductions in the search space are achieved by the Linear-input-form strategy. This insists that one resolvent is always from the base set (the original set of clauses). Although this method is more efficient, it is not complete since there are some theorems of first order predicate calculus that cannot be proved using the method. Another elaboration of resolution theorem proving involves the handling of the equality relation [53] which allows theorems such as

$$(a = b \ \& \ P(a)) \rightarrow P(b)$$

to be proved.

5.2.2 Natural Deduction

Natural deduction (or non-resolution) theorem provers attempt to mimic the behaviour of human theorem provers. The method of natural deduction maintains a clear distinction between goals (formula that the theorem prover is attempting to prove) and the antecedents from which it is attempting to prove the goals. There are two modes of theorem proving, *forward chaining* and *backward chaining.* In the forward chaining mode the rules of deduction are applied to the hypothesis to produce results which may then be used recursively to prove the desired conclusion. In the backward chaining mode, the rules of deduction are used to produce subgoals which imply the conclusion. These subgoals are then proved recursively. A natural deduction theorem prover will use one or both of these modes of operation.

Unlike resolution theorem provers, a natural deduction theorem prover uses many rules of inference. Some examples of rules that might be used are:

$$H = C \vdash H \rightarrow C$$

$$H \rightarrow A \ \& \ H \rightarrow B \vdash H \rightarrow A \ \& \ B$$

$$H_1 \rightarrow C \ \& \ H_2 \rightarrow C \vdash H_1 \vee H_2 \rightarrow C$$

$$A \rightarrow C \vee B \rightarrow C \vdash A \ \& \ B \rightarrow C$$

$$H \ \& \ A \rightarrow B \vdash H \rightarrow (A \rightarrow B)$$

and finally

$$B \rightarrow C \ \& \ H \rightarrow A \vdash H \ \& \ (A \rightarrow B) \rightarrow C$$

One of the main advantages of natural deduction theorem proving is that because the theorem proving process is easy for humans to follow, it is possible for humans to guide the theorem proving process, either interactively or by supplying domain dependent heuristics. Some examples of domain dependent heuristics that can be used by a natural deduction theorem prover are:

Rewrite Rules (or reduction rules) These are equalities which can (should) be used as substitutions and have an explicit direction in which they should be used. For example the rules

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

and

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

can be used to help normalise arithmetic expressions. There are two desirable properties of a set of rewrite rules; one is finite termination and the other is unique termination. A set of rewrite rules has the finite termination property if there is no infinite sequence of expressions $e_1, e_2 \ldots$ where e_i rewrites to e_{i+1} for all $i > 0$. A set of rewrite rules has the unique termination property if for every expression e all irreducible forms of e are identical. There are algorithms for deciding the unique termination property of a set of rules that are finitely terminating, as well as algorithms for extending sets of rewrite rules that fail to meet the unique termination property.

Forward chaining Another group of domain dependent heuristics are concerned with forward chaining deductions which produce new hypotheses from the original set of hypotheses. These heuristics are in the form of demons that scan the hypotheses looking for sets of assertions. If they find an appropriate set then they make their own assertions based on the ones they have found. For example:

> **if** $A \subset B$ **and** $C \subset B$ **are hypotheses and if** $A \cup C$ **is**
> mentioned somewhere, then assert $(A \cup C) \subseteq B$.

Decision procedures For certain theorems there are algorithms to decide if a sentence is true or false very quickly. For example, sets of linear inequalities over the real numbers can be decided by the simplex algorithm.

Examples and counterexamples Heuristics used by human theorem provers often involve examples and counterexamples. Given a set of axioms T and a request to prove $H \rightarrow C$ then an example is an interpretation of the symbols of T, H and C that satisfies the axioms T and hypothesis H.

Supposing $A \rightarrow B$ is one of the axioms, and a subgoal that the theorem prover is currently attempting to prove is B. Rather than trying to prove A immediately, a check can be made to see if A is true under the example interpretation. If it is true for the example then it is worth trying to prove A as a subgoal. If A is false for the example, it must also be false in the general case and therefore it is not worth attempting to prove A as a subgoal.

Another type of inference rule used in natural deduction is induction. This is particularly important for dealing with recursively defined objects and is described in the following section.

5.2.3 Proof by Induction

Supposing some property $P(n)$ is to be proved for all natural numbers n. It is sufficient to prove the two properties $P(0)$ and $P(n) \rightarrow P(n+1)$. For example, if $P(n)$ states that $x^{(n+m)} = x^n * x^m$ for all x and natural numbers n and m then $P(n)$ can be proved in the following manner:

Prove $P(0)$.

$$\begin{aligned} x^{(0+m)} &= x^m \\ &= x^0 * x^m \end{aligned}$$

which establishes the *base case.*

Prove $P(n) \rightarrow P(n+1)$ by assuming $P(n)$ as the induction hypothesis and proving $P(n+1)$.

$$\begin{aligned} x^{(n+1)+m} &= x * x^{(n+m)} \\ &= x * x^n * x^m \qquad \text{(from the induction hypothesis)} \\ &= x^{(n+1)} * x^m \end{aligned}$$

which establishes the *induction case* (or *induction step*).

The conclusion that $P(i)$ is true for all integers i is justified by the fact that for any finite i, a finite proof for $P(i)$ can be constructed using the proof of $P(0)$ above followed by proof of $P(n) \rightarrow P(n+1)$ initially with n as 0 to establish P(1), then with n as 1 to establish P(2) and so on until P(i) is proved.

The above proof, based on one or more base cases and one or more recursive (or inductive) cases, is known as a proof by induction. This particular style of induction or induction principle is one of many that are specific to the natural numbers. A similar principle exists for induction over lists, the difference being that `[]` replaces 0 and `(e:)` replaces (+1) (where `e` is a new universally quantified variable).

Rather than relying on a theorem prover to invent an induction principle for every type, a general induction principle for all types can be used to create specific inductive proofs tailored to the type and context of the induction variable.

Such an induction principle is based on the idea of a *well founded relation.* A well founded relation is one which relates two arguments of the same type according to whether or not one is "less" than the other in a specific sense. The crucial property of well foundedness comes from the restriction that there must be no infinitely decreasing sequences in the set of values over which the relation is defined. If R is a well founded relation and $R(x, y)$ is true then x is said to be "R-less" than y. Thus R is well founded provided there is no infinite sequence

$$\ldots\ x_{i+1},\ x_i,\ \ldots, x_1$$

such that x_{i+1} is R-less then x_i for all $i > 0$.

The generalised principle of induction, which states that to prove $P(x)$ it is sufficient to prove the base case:

$$\tilde{}Q(x) \rightarrow P(x)$$

and the induction step:

$$Q(x)\ \&\ P(d(x)) \rightarrow P(x)$$

is only valid provided $Q(x) \rightarrow R(d(x), x)$ and R is a well founded relation. In other words the induction only holds provided $d(x)$ is R-less than x when $Q(x)$ is true.

An informal justification of induction principle is that p can be proved for any finite x provided a finite number of applications of the induction step and base case lead to the conclusion $P(x)$. This is guaranteed by the condition that provided $Q(x)$ is true, $d(x)$ must be R-less than x where R is a well founded relation, so there can only be a finite number of applications of d to x which remain R-less than x. When the R-less relation no longer holds $Q(x)$ must also be false (the induction is only valid provided $Q(x) \rightarrow R(d(x), x)$) and if $\tilde{}Q(x)$ then $P(x)$ (the base case). So the conclusion $P(x)$ for all finite x must follow from the validity of the induction instance and the proof of the induction cases.

An induction scheme similar to the one for natural numbers mentioned above can be derived from the generalised induction principle if the functions Q and d are given the following values:

$$\begin{aligned} Q(x) &= (x \neq 0) \\ d(x) &= x - 1 \end{aligned}$$

giving

$$\begin{aligned} x = 0 &\rightarrow P(x) \ \& \\ x \neq 0 \ \& \ P(x-1) &\rightarrow P(x) \end{aligned}$$

which is a valid induction because the relation R where

$$R(x, y) = x < y$$

is well founded for the natural numbers and

$$p \neq 0 \rightarrow x - 1 < x$$

The generalised induction principle introduced above can be extended to allow an arbitrarily large number of induction variables which are R-less according to some measure (usually lexicographic ordering). There are also $k >= 1$ induction cases each with one or more induction hypotheses. These further generalisations allow inductions on sum of product types, each induction case dealing with one of the recursive shells with a hypothesis for each recursive component and a case condition (Q) which recognises the particular shell.

Due to its close relationship with recursion, induction plays an important role in proving theorems about recursive functions. Boyer and Moore[12] argue that any proof system which is not capable of inductive arguments must, when attempting to prove theorems about recursively defined objects, be doomed to assume what it is trying to prove.

Discovering the appropriate form of induction to use in a given situation is a difficult problem. In particular, the theorem prover must decide which variables to base the induction on, what the inductive cases should be and what substitutions should be used to create the induction hypotheses. A powerful set of heuristics for selecting the best induction scheme to use are employed by the Boyer and Moore theorem prover (BMTP)[12] which is described in the following section.

5.2.4 The Boyer and Moore Theorem Prover (BMTP)

The BMTP is based on an extendible theory of recursively defined functions and data objects. Only provably total functions are admitted to the theory which is not strongly typed. The totality of a function is proved using a well founded relation over a measure of the arguments to the function which ensures that the arguments are 'less' on each recursive call in the definition body the function. This ensures that each definition describes only one function and the theory may not be rendered inconsistent by the addition of a new function definition. This measure and well founded relation also play a key role in the theorem proving process by suggesting the form of an inductive argument for conjectures that involve the function.

BMTP uses recursive functions as an alternative to existential and universal quantification. Axioms and theorems are treated as functions which return either `F` if they are false or `~F` if they are true (since the theory is not strongly typed a theorem may have values other than `T` and `F`). As theorems are proved by the system, they are retained and used in the proof of subsequent theorems. In this way proof guidance can be given by providing a sequence of lemmas each of which is proved and then used by BMTP in the subsequent proofs. Thus given a theorem to prove (what), BMTP attempts to provide the proof (how), but if it is not capable of doing this it can be assisted with the appropriate sequence of lemmas to enable it to prove the theorem.

The fundamental objects of BMTP are `T` and `F` and the fundamental operators are `IF`, `==` and `~==`[2]. The behaviours of these are defined by the following axioms

[2]BMTP is described using Miramod notation rather than the original notation employed by Boyer and Moore

which BMTP assumes.

$$
\begin{array}{lcl}
\mathtt{T} \neq \mathtt{F} & & \\
\mathtt{x} = \mathtt{y} & \rightarrow & (\mathtt{x} == \mathtt{y}) = \mathtt{T} \\
\mathtt{x} \neq \mathtt{y} & \rightarrow & (\mathtt{x}\ \tilde{}{==}\ \mathtt{y}) = \mathtt{F} \\
\mathtt{x} = \mathtt{F} & \rightarrow & (\mathtt{IF\ x\ y\ z}) = \mathtt{z} \\
\mathtt{x} \neq \mathtt{F} & \rightarrow & (\mathtt{IF\ x\ y\ z}) = \mathtt{y}
\end{array}
$$

The logical function are defined as follows:

```
   ~p = IF p F T
 p & q = IF p (IF q T F) F
p \/ q = IF p T (IF q T F)
p |- q = IF P (IF q T F) T
```

BMTP allows recursively (inductively) defined data objects to be added to the theory in such a way that consistency with lemmas already proved is guaranteed. These data objects are based on shell functions[3] and consist of a shell function name, an optional bottom object, a recogniser function name and n selector function names where n is the arity of the shell function. The recogniser function returns T if its argument is the shell function or bottom object and F otherwise. The selector functions each return an argument of the shell. The name of a well founded relation for the shell is also given. An example of a shell definition is:

```
Add the shell ADD1 of one argument
with bottom object (ZERO),
recogniser NUMBERP,
selector SUB1,
and well-founded relation SUB1P.
```

Given a shell definition, and provided the names used do not clash with existing

[3]In this thesis the use of the word 'shell' to describe a class of constructor functions, stems from the shell principle of BMTP.

names BMTP adds a number of axioms concerning the names defined by the shell. These include:

```
r (shell x1 ...xn),
r btm,
(r x & x ~== btm) |- (shell (sel1 x) ...(seln x)) == x
```

where r is the recogniser, *shell* is the shell function, $\mathtt{x}_1 \ldots \mathtt{x}_n$ and x are variables, *btm* is the bottom object and sel_i is the i^{th} selector.

Shell objects are finite and mutually exclusive, which means that no two bottom objects or shells represent the same object. They are also not exhaustive, which means that an object does not have to be one of the currently defined shell objects or T or F. This ensures that adding new types does not alter the truth of existing theorems. Since the BMTP theory is not strongly typed a function can return values of any shell.

The Boyer and Moore theorem prover is based on the generalised principle of induction, or Noetherian Induction[26] and the majority of its heuristics are oriented towards induction proofs.

To prove a conjecture the system attempts to rewrite it to `~F` (since the language is not strongly typed it is quite possible for a conjecture to return a value that is neither `T` or `F`). The various heuristics (such as using lemmas as rewrite rules) are layered; the least risky ones are applied first (the ones that guarantee equivalence) and the most risky (induction) applied last. If any phase actually alters the theorem being proved then all the preceding phases are re-applied. Induction is therefore applied to the simplest and most general form of a conjecture. Many of the earlier heuristics are orientated towards producing a conjecture that is amenable to inductive arguments.

The heuristics, in the order that they are applied, are:

Simplification This involves the use of axioms (including definitions and shell axioms) and previously proved lemmas to simplify the conjecture.

Elimination of undesirable concepts. Lemmas about the equivalence of terms can be used to suggest the elimination of certain functions or operations. For example operations such as / and - might be traded for operations such as * and +.

Using equalities When the conjecture being proved has an equality as one of its hypotheses, the equality is sometimes used to substitute one of its operands for the other in the remainder of the conjecture and then removed from the conjecture. Two distinct heuristics, uniform substitution and cross fertilisation are used. Uniform substitution performs a substitution for all occurrences of the term being substituted for, whereas cross fertilisation performs only a single substitution.

Generalisation The generalisation heuristics are designed to help prepare a conjecture for induction. Conjectures must frequently be generalised before they can proved because without generalisation the induction hypothesis may not be sufficiently strong to prove the theorem. Generalisations are achieved by replacing terms with variables. To prevent over generalisation to a non theorem, shell type information about the new variable is often added to the hypothesis.

Elimination of irrelevant terms This stage cleans up irrelevant terms that have been generated by the previous simplifications.

Induction Inductions are formulated from information collected when definitions are added to the theory and from information available at the time of induction. When a function is defined it must be proved to be total using a measure of its arguments and a well founded relation. An induction is only valid if when the substitution is applied to a measure of the induction variables the result is 'R-less' than the original measure according to some well founded relation 'R'. Thus the same measure and well founded relation used to prove termination of the function may be used to suggest an induction template for conjectures that involve the function.

At Induction time all the templates of each recursive definition in the formula are retrieved as candidate induction templates. The following heuristics are then applied to formulate an induction from these templates:

1. Templates that do not apply to the current conjecture are thrown out

(ie a nonvariable argument appears as an induction variable) and ones that do are instantiated with the actual parameters which appear in the conjecture. This gives a set of induction schemes.

2. Subsumed[4] induction schemes are removed.
3. Schemes are merged. Thus if an induction on x is suggested by one schema and an induction on y by another then an induction on x and y is produced by merging the schemes.
4. Flawed schemes are discarded.
5. Scoring functions are used to pick between any remaining schemes.

Limitations

The BMTP's limitations lie in two areas: firstly its ability to represent facts and theorems in the domain of interest; and secondly its ability to prove theorems efficiently.

One of the main limitations is the fact that the BMTP only allows total functions to be defined. This limitation stems from BMTP's insistence that a definition is only admitted if it has been proved to terminate and there is no mechanism, other than non termination, for describing a functions value for a subset of its possible inputs. This does not mean that theorems about potentially non terminating programs or even theorems about the termination of programs cannot be expressed and proved. The language used to express the conjectures being proved need not be the same as the language in which the program referred to by the conjectures is written. In the general case, an appropriate theory of program semantics must be applied to the source program to give a formal statement in the BMTP notation whose validity implies the desired properties of the program.

In terms of efficiency, the BMTP suffers from what is known as the referencing problem. That is, the more axioms and lemmas that are available to the theorem prover, the worse its performance gets.

[4]The subsumption relation is described fully in [12], however an intuitive description of the relation is that one scheme subsumes another if it covers all the cases and substitutions of the other and can therefore be used in place of the other schemes.

5.3 The Design of a Property Prover

The intention of the property prover is not to investigate novel methods of theorem proving, but to apply existing methods to the task of component library retrieval. Thus the strategy has been to follow an existing method as closely as possible, innovating only when the peculiarities of the application make this essential.

The design of the property prover follows the design of the theorem prover described by Boyer and Moore in [12]. This choice was made for several reasons. Firstly the properties which must be proved are properties of recursive functions and inductive data types, and hence the heuristics concerned with induction which are central to BMTP are also important to the property prover. Secondly Miramod and the BMTP theory (which is based on pure LISP) are very similar. Thirdly the proofs and heuristics of BMTP are 'natural' to humans, but resolution based proofs are not. To allow the comparison of models that are more than just trivially different, it is essential that some domain specific heuristics are available. These heuristics can form part of the model in the library and be used to help match the library model against the required model. However if these heuristics are to be added to the model by librarians or component contributors then the model comparison process must be understandable to the librarians. Since the proofs followed by resolution theorem provers are highly unnatural to humans, a resolution based property prover would provide little opportunity for the addition of heuristics to the model.

Given the decision to follow the BMTP design, an important choice must be made between the alternatives of translating the Miramod notation to the BMTP theorem or adapting the BMTP heuristics to the Miramod notation. Though the underlying semantics of both notations are very similar, Miramod provides a great deal more 'syntactic sugar' since one of its primary goals is to allow properties to be expressed simply, elegantly and with the minimum of effort. The BMTP notation is very sparse by comparison. It is designed for ease of proving theorems rather than ease of describing properties. The solution adopted is therefore to describe property models using the Miramod notation and compile this to BMTP notation for the purposes of comparing models. The design and implementation of the compiler is described separately in chapter 6.

The compiler and property prover are used in two contexts: one as part of the retrieval process, and the other as tools for building and maintaining the component library.

The design of the retrieval system is summarised in figure 5.1. This diagram has three main parts: the re-user who wishes to retrieve a component, the retrieval system which attempts to find appropriate components and the component library which stores components. The retrieval system initially receives a request from the re-user in the form of a property model (the diagram arrows indicate data flow). This property model is then compiled and any errors in the model (including type errors) are reported back to the re-user . The compiled model is then used by the property prover to compare with the compiled models contained in the library. Any models sufficiently similar to the re-users model are collected as part of the component selection, a summary of which is presented to the re-user . If this selection is large then the components are presented in order, with the most similar components first. The re-user may then pick components from the selection and view them in greater detail, choosing to remove them from the selection or retrieve them as appropriate.

The compiler and property prover are also used by the librarians and component contributors to create models that can be used to identify stored components. This process is summarised in figure 5.2.

Each component has one or more property models associated with it, and these models also contain at 'theory' of the model which assists the property prover in proving properties of the model (for example the component which appends lists might be described by a model whose theory includes the fact that 'append' is associative). The compiler is used to produce a compiled version of the model (including the associated theory) and any errors detected are reported to the librarian or component contributor. The model can also be checked to ensure that the theory is consistent with the model by attempting to prove the theory from the model (for example proving the associativity of 'append' from the models description of append).

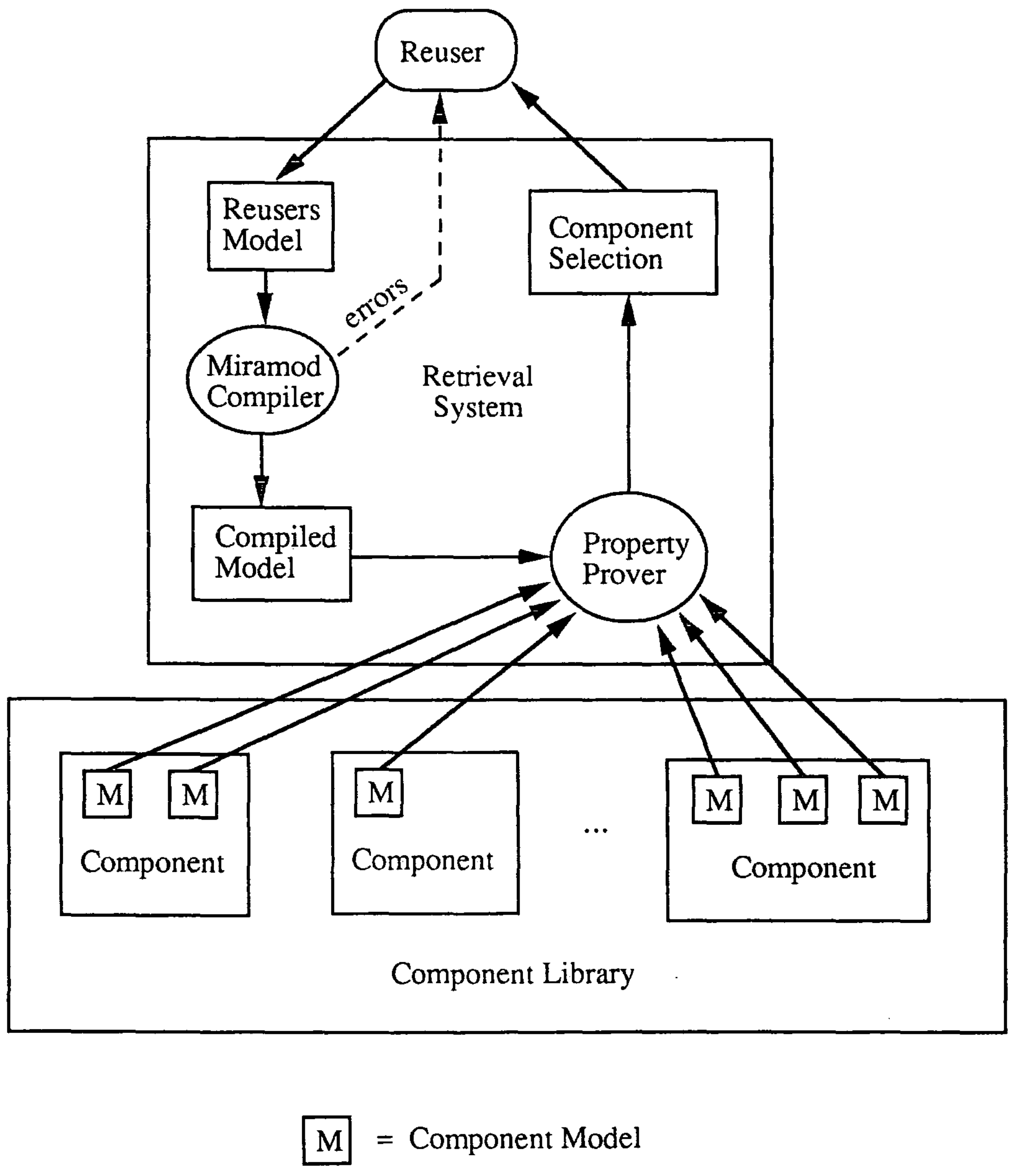

Figure 5.1: The Retrieval System

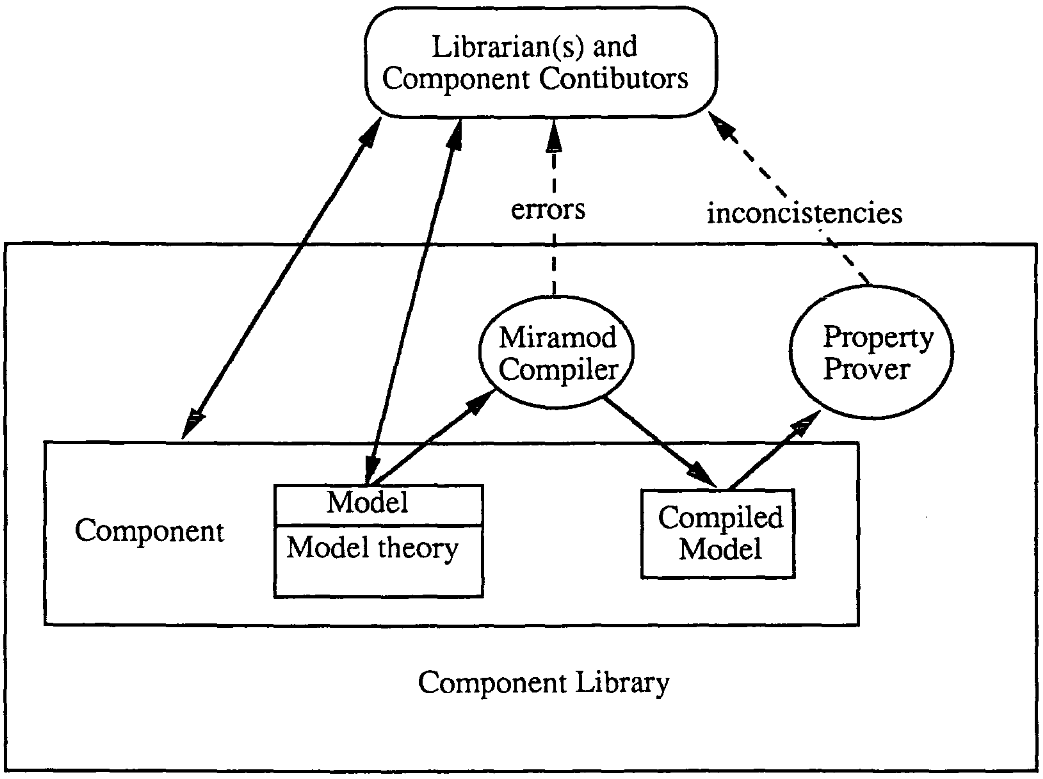

Figure 5.2: Producing Component Models

5.3.1 Differences between BMTP and the property prover

The property prover is based on theorem proving techniques, but diverges from conventional theorem proving in several areas. Rather than producing an answer 'yes' or continuing to attempt a proof indefinitely, the property prover returns a result which is intended to be a similarity measure. This measure is guaranteed to be returned within a certain time limit, even if it is incorrect (in the sense that it indicates its lack of theorem proving ability rather than the similarity of models).

Due to the differences in the objectives of the two systems, the theory on which they are based is not entirely the same:

- Partial functions are allowed by the property prover but not by BMTP.
- The property prover admits arbitrary definitions to be added whereas BMTP insists that a function is proved total before its definition is added.
- Arbitrary axioms may be assumed by the property prover but the BMTP only allows axioms to be admitted via the shell principle, the definition principle or by proving them as lemmas.
- The property prover allows higher order functions, whereas BMTP insists that variables may not appear in the place of function symbols.
- Function definitions may contain local definitions in the property prover but not in the BMTP.
- In addition to the BMTP values `T` and `F` the property prover has special values, `True`, `False` and `Undef` as well as a special weak equality operator `=`.
- The property prover assumes that all expressions and definitions are strongly typed.

The relaxed theory of the property prover provides two serious problems. Firstly since the principle of definition is not used, there is no way to ensure that the assumption of definitions does not introduce inconsistencies. Secondly, the principle of induction needs additional checks to ensure that it remains sound because a value can be infinite, in which case the induction principle only holds for conjectures that are *chain complete* (see 5.4.11).

The introduction of inconsistencies cannot be prevented since Miramod allows functions and types to be described using property statements which are in effect axioms. Despite this it is important to prevent axioms which would render the basic theory of the BMTP inconsistent (for example the axiom `T=F`).

For this reason functions and properties of `T` and `F` are not allowed to be introduced as axioms or definitions. Contradictions may still be introduced accidentally, for example the axiom `True == False` introduces a contradiction, but the way in which axioms are used by the property prover limits the effects of such contradiction to the properties of `True` and `False` along with functions which involve `True` and `False`. BMTP will only use these axioms to rewrite '`True == False`' to `T` thus ensuring that any equality of expressions which always return either `True` or `False` will always be provable. However it will not make the inference that `True` $==$ `False` $\rightarrow$ `F` and hence `T=F` which means that everything is provable. Although the above example is based on an extremely crude and obvious contradiction, the introduction of a contradictions can occur in far more subtle ways. Any contradiction introduced constitutes an error by the writer of the model, and should ideally be detected by the property prover. BMTP achieves this by putting the onus on the user to prove that any definitions they add do not contradict any previous definitions or axioms. Since the property prover must freely admit definitions and axioms, it provides no such safeguard, but instead attempts to limit the implications of a contradiction so that the model may still retain some useful meaning and retrieval may still be possible.

5.3.2 Property Prover Notation

The notation used to describe the property prover is essentially a subset of the Miramod notation, however there are a number of differences, especially concerning the built in values and functions used by the property prover.

Property statements return the BMTP values 'T' or 'F' which are referred to as BMTP Boolean values.

There is a special shell called `Undef` which is produced by the Miramod function `undef` and has the recogniser `(Undef==)`. `Undef` is a member of every type other than the property type which allows only the values `T` and `F`. There is also a special equality operator, called weak equality and denoted `=`, which differs from the normal BMTP equality `==` in that it returns one of `True`, `False` or `Undef` rather than `T` or `F`. The weak equality operator returns `Undef` if either of its arguments are functions or are only partially defined.

Miramod types are represented directly as BMTP shell objects (bottom objects are not used) one BMTP shell per Miramod shell and the shell function is given the same name as its Miramod equivalent. Given a shell function `C`, the recogniser function of each shell is called `REC_C` and the selector functions are called `SEL_C_n`, where `n` is the number of the argument selected by the function (the first argument is number 1). Shells with no arguments (constants) have a special notation for their recogniser function which is written `(C==)` where `C` is the shell name.

Hence for the Miramod list type there are two recogniser functions `([]==)` and `REC-:` as well as two selector functions `SEL-:-1` and `SEL-:-2`. The following shell definitions are used:

```
Add the shell [] of zero arguments.

Add the shell : of two arguments
with
recogniser REC_:
selectors SEL_:_1 and SEL_:_2
```

To make the examples given in this thesis more readable, selector functions are usually given meaningful names such as `hd` for the selector function `SEL-:-1` which selects the head of a list and `tl` for the selector function `SEL-:-2` which selects the tail of a list.

Functions defined in Miramod that use pattern matching in their parameters, are translated into a form which uses the shell recognisers to select the appropriate pattern case and the selector functions extract formal parameters which appear as parts of patterns. For example the list append operator ++ can be defined by the following equations:

```
[]++x = x
(a:x)++y = a:(x++y)
```

Given this definition the translated version is:

```
x ++ y
   = IF (x==Undef) Undef
        IF (x==[]) y
           ((SEL-:-1 x) : ((SEL-:-2 x) ++ y)
```

5.4 Proving the properties

The property prover is closely based on the Boyer and Moore Theorem Prover. The main differences are that the property prover is capable of proving properties of partial functions, whereas the BMTP theory does not admit the definitions of partial functions and the property prover relies on domain dependent information provided by the component library rather than interactive human assistance to help prove properties. Another important difference is that the property prover provides a crude measure of similarity rather than a straight 'yes' or 'no' answer.

A fundamental assumption made by the property prover is that the property models contained in the library are not only written so that it is easy to prove their properties, but that they include domain specific guidance to the property prover on proving properties of that particular model (rather than general heuristics for proving properties). The form of this guidance and its use by the theorem prover is described in the following sections. Since this assumption is fundamental to the property provers ability, there is no attempt to prove the properties of a library model from the properties of a re-user 's model. The re-user cannot be expected to provide information on how to prove theorems about their model because this is a time consuming and highly specialised task.

5.4.1 The proof strategy

The property prover starts with a set of assumptions including the properties and definitions of the hypothesis (library) model and a property expression from the re-users model which is to be proved (this is referred to as the conjecture).

The conjecture is initially represented as a single clause containing only the one disjunct.

The property prover has a number of simplification phases which it applies to the conjecture in order. Each phase takes a clause as input and returns a set of clauses as its output. If the result of a phase is the singleton set of clauses containing the input clause unaltered then the clause is given as input to the next phase, if the clause is altered by a phase in any way then each member of the resulting set of clauses is returned to the first phase. Clauses that manage to pass through all the phases unaltered are collected until there are no further clauses to be simplified. At this stage one of the collected clauses is selected and an appropriate induction scheme constructed. The resulting set of clauses are simplified again from the first phase onwards.

Boyer and Moore describe this with the analogy of a stepped waterfall between two pools. The top pool contains the unsimplified clauses and each step of the waterfall represents a simplification phase. Clauses trickle down from the top pool to the bottom, hopefully evaporating (being simplified to true) on the way down but sometimes being altered and returned to the top pool. Eventually the top pool and waterfall are both empty. At this stage if there are no clauses left in the bottom pool then the property is proved, otherwise a clause is selected, an appropriate induction created and the resulting set of clauses are placed in the top pool for re-simplification.

This proof strategy is exactly the one used by BMTP; however the property prover employs an additional heuristic that is concerned with the proof of *partial properties* which concern the behaviour of the model over partial and infinite values. Clauses are labelled according to whether or not they are partial (this criterion is discussed in section 5.4.2). When a choice of induction candidate is being made, non partial clauses are given priority over partial clauses. If the non partial clauses are all simplified to true, then the property being proved is given a score to indicate that at least its non partial properties can be proved and then an attempt is made to prove its partial properties. If a partial clause is simplified to **F** on the other hand, any other partial clauses generated in the proof are immediately assumed to be **F** but the property prover continues attempting to prove the non partial properties.

This heuristic is justified on two accounts. Firstly the partial properties tend not to be as good a guide to component similarity as the non partial properties and secondly partial properties which survive the simplification stage tend to be harder to prove than non partial properties (they are often undecidable). The belief that partial properties are not a good guide to component similarity stems from the fact that Miramod is designed to allow re-users to write property models without worrying about the details of partial behaviour. It is supported by the experience that ignoring partial behaviour is a good way of simplifying a model and that accurately describing the partial behaviour of a component is difficult. Partial properties can often be proved by simplification because the partial behaviour of many functions is relatively simple and can be explicitly described as part of the library model. For this reason they are allowed to continue through the simplification phase in the hope that they will disappear. If they do not yield to simplification then the partial behaviour is likely to be complex and difficult if not impossible for the property prover to prove, so it is reasonable to leave such proofs aside whilst performing more important proofs and return to the partial proofs "if time allows".

5.4.2 Simplification

The initial simplification phase involves three main heuristics.

- Using 'type' information.
- Using rewrite axioms.
- Opening up definitions.

The word 'type' used above does not refer to the Miramod types as such but to the shells of Miramod types. The property prover assumes that the properties it is trying to prove are correctly typed and hence that an equality can never be false because the two expressions being equated return values of different types. The type information used for simplification concerns which subset of the shells of a type can be returned by an expression rather than its Miramod type. To make this distinction clear, the type information used by the property prover is referred to as *shell type* information.

The strategy for simplifying clauses is centred around the idea of assuming all other literals in the clause false whilst attempting to simplify the current literal (Since `( ~A |- B ) == A \/ B`).

Each literal in the clause is simplified in turn, so at any time there will be a set of literals that have already been simplified and a set that have not.

$$\{new_1 \ldots new_n,\ old_{n+1} \ldots old_m\}$$

Supposing the literal new_n has just been simplified, literals new_1 to new_n and old_{n+2} to old_m are assumed false whilst old_{n+1} is simplified. If the result is `T` then the clause is proved and an an empty list of clauses is returned. If the literal simplifies to `F` then it is removed from the clause and the next literal is simplified. Otherwise any `IF` functions in the literal are distributed over two copies of the clause each dealing with one of the `IF` branches:

$$\{\ldots new,\ (\ S\ (\texttt{IF}\ c\ l\ r)),\ old\ldots\}$$

gives

$$\{\ldots new,\ ({}^{\sim}c),\ (S\ l),\ old\ \ldots\}$$

and

$\{\ldots new,\ c,\ (\mathcal{S}\ r),\ old\ldots\}$

where $\mathcal{S}$ is the schema for a literal and ($\mathcal{S}$ `t`) is the literal obtained by replacing the schema variable by term `t`.

Both of the resulting clauses are recursively checked for further IF functions and split until no further `IF`'s remain. Once an `IF` free literal is obtained the next literal is simplified under the assumption that all the other literals (including the two newly generated ones) are false. The lists of clauses returned by the simplifications of the two clauses generated are combined to form the result of the original simplification.

This removal of `IF` expressions from the literals of the conjecture is important because it allows the left branch to be simplified under the assumption that the condition is true and the right branch to be simplified under the assumption that the condition is false. If both of the resulting literals can be simplified to true then the clause is proved.

The method of simplifying clauses is taken directly from BMTP, however it provides the motivation for compiling to function definitions which base all 'choices' or 'branches' on only a single function, the `IF` function. Since both the pattern matching definitions and guarded definitions are compiled to functions based on the `IF` function, this heuristic allows the property prover to perform case analysis on both forms of definition. The compiler frequently generates code of the form '`IF (x==Undef) y z`', which in turn produces clauses containing literals such as '`x ~==Undef`'. These are the partial property clauses which receive special attention from the property prover: if they cannot be simplified to true then no attempt is made to prove them by induction until all the other clauses have been proved. The matching value resulting from an attempt to prove a property is dependent on both the proof of both the partial and non partial properties.

Using type information

There are two key processes involved in the use of type information. One is the process of assuming that an expression is `T` or `F` and the other is the process of calculating the set of shells that can be returned by an expression. These sets of shell types are referred to as *shell type sets.*

The assumption that an expression is true or false occurs in two situations. Firstly when a literal of a clause is being simplified, the other literals of the clause are all assumed to have the value `F`. Secondly, during the simplification of an expression `IF c a b`, the branch `a` is simplified under the assumption that the condition is `T` and the branch `b` is simplified under the assumption that the condition is `F`.

Each shell recogniser function has an associated shell type which is denoted by underlining the shell name or symbol. For example:

$\underline{\texttt{True}}$

$\underline{\texttt{False}}$

$\underline{\texttt{:}}$

There are also several special shell types:

$\underline{\texttt{T}}$

$\underline{\texttt{F}}$

$\underline{\texttt{Undef}}$

and a special shell type set:

$\underline{\text{UNIVERSE}}$

As an example of assuming an expression true or false, consider the assumption that the expression '`T1 == T2`' is true. Apart from the obvious assumptions that the shell type set of the expressions 'T1 == T2' and 'T2 == T1' is $\{\underline{\mathtt{T}}\}$ it is also possible to calculate the type sets of T1 and T2 and assume that both expression have a type set which is the intersection of their calculated type sets. If T1 has the value Undef and assuming that the typeset of T2 includes the type $\underline{\mathtt{Undef}}$, then the assumption that the typeset of T2 is $\{\underline{\mathtt{Undef}}\}$ is also valid.

To compute the type of an expression, the current set of shell type assumptions are consulted initially and the corresponding shell type set returned if an assumption for the expression is found. Otherwise shells return the set containing only their corresponding type; recognisers return $\{\underline{\mathtt{T}}\}$ if the parameter's type set is the set containing only the recognised shell type, $\{\underline{\mathtt{F}}\}$ if it does not contain the recognised shell type and $\{\underline{\mathtt{T}}\ \underline{\mathtt{F}}\}$ otherwise. A special case is made when calculating the type set of an IF expression. In this case the typeset of the condition is computed initially and if this yields either true or false then the typeset of the appropriate branch of the IF is returned. Otherwise the typeset of the left branch is calculated under the assumption that the condition is true and the typeset of the right branch is calculated under the assumption that the condition is false. The type set returned for the IF expression is then the union of these two type sets.

The property prover uses type sets in the same way as the BMTP, but the fact that Miramod is strongly typed means that several additional axioms about shell type sets are also used.

These are, for all integers n in the range $1 \dots m$:

$$\underline{\mathtt{UNIVERSE}} - \{\underline{\mathtt{C}_n}\} = \{\underline{\mathtt{Undef}}\ \underline{\mathtt{C}_1} \dots \underline{\mathtt{C}_{n-1}} \dots \underline{\mathtt{C}_{n+1}} \dots \underline{\mathtt{C}_m}\}$$

where $\underline{\mathtt{C}_1} \dots \underline{\mathtt{C}_m}$ are all the shell types of a Miramod type. The corresponding properties of T and F are special because BMTP Booleans cannot have the value Undef:

$$\begin{aligned} \underline{\mathtt{UNIVERSE}} - \{\underline{\mathtt{T}}\} &= \{\underline{\mathtt{F}}\} \\ \underline{\mathtt{UNIVERSE}} - \{\underline{\mathtt{F}}\} &= \{\underline{\mathtt{T}}\} \end{aligned}$$

Using rewrite axioms

BMTP refers to these as lemmas since they must be proved when they are introduced. They are referred to as axioms by the property prover since they are simply assumed rather than being proved.

Rewrite axioms are special property statements of the form:

```
H |- L => R
```

and are used to simplify expressions of the form L to the equivalent form R provided the hypothesis H holds. A simple example is based on the associativity of addition. In this case, no hypothesis is needed so the rewrite axiom is written L => R.

```
a+(b+c) => (a+b)+c
```

Rewrite axioms are the first example of domain specific rules that can be added to a model to help prove properties of the model. The presence of a rewrite rule does not simply mean that the theorem prover **may** rewrite the expressions as indicated but that it **should** do so whenever possible. In this way the presence of rewrite axioms provides the property prover with heuristic information on how it should proceed in its attempt to find a proof.

There are several situations in which the indiscriminate use of rewrite axioms can lead to an infinite sequence of rewrites. One example of this is if the rewrite rule is *permutative*, which means that the left and right hand sides are instances of each other. An example is the rewrite rule based on the commutativity of plus 'a+b => b+a'. Although such a rewrite rule is important since the alternative to its use is usually induction, using such a rule indiscriminately would lead to an infinite sequence of rewrites. To prevent this the property prover will only apply a permutative rewrite rule if the terms moving left are moved into positions previously occupied by alphabetically greater terms. In the commutativity of plus example, b is moved left and replaces a, so a must be an alphabetically greater term then

b. This allows rewrite rules to normalise expressions composed of the same terms. For example, using the permutative rewrite rules:

```
b+a => a+b                (1)
b+(a+c) => a+(b+c)        (2)
```

the expression `(i+j)+k` can be rewritten to the normal form `i+(j+k)`, since the term `(i+j)` is considered alphabetically greater than `k` (an arbitrary decision which is applied consistently).

These rewrite rules ensure that any nest of + expressions with the same "bag" of arguments can be rewritten to the same term. The rewrite rules for + can be extended to allow equivalent terms made up of nests of + and - expressions over the same set of arguments to be rewritten to the same term. The following rewrite rules do this by ensuring that all positive terms are placed on the left of a single top level '-' operator and all negative terms are placed on the right.

```
    b+a => a+b
b+(a+c) => a+(b+c)
a-(b-c) => (a+c)-b
(a-b)-c => a-(b+c)
b+(a-c) => (a+b)-c
(a-b)+c => (a+c)-b
```

Permutative rewrite rules are not the only ones that can potentially lead to an infinite sequence of rewrites. Rewrite rules whose hypothesis can be rewritten by its own left hand side can lead to an infinite sequence of attempts to justify the hypothesis. An example of such a rule is:

```
(a+1)<b |- a<b => T
```

An attempt to rewrite `a<b` would lead to an attempt to rewrite `(a+1)<b`, `(a+2)<b` and so on.

To prevent this from occurring a record is kept of all the hypotheses currently being simplified. Prior to the rewriting of a hypothesis, a check is made to ensure that none of the conjuncts of the hypothesis is an elaboration[5] of a hypothesis that is already undergoing simplification. If one of them is, then no attempt is made to rewrite the current hypothesis and the rewrite rule is not used.

The above heuristics are exactly those used by BMTP, however the property provers treatment of rewrite rules differs in two respects. One is that rewrite rules are treated as axioms in that they do not have to be proved before being accepted by the property prover (Although they can be proved if the model writer requires them to be). The other is that rewrite rules may be used to rewrite higher order functions as well as more conventional terms. In effect a rewrite rule can rewrite a function without all of its arguments. An example of such a rule is the associativity of the function composition operator '`.`' (defined by '`. f g x = f (g x)`').

```
(f.g).h => f.(g.h)
```

As a result of the way in which partial properties are admitted to the property prover theory, rewrite rules which apply only to non partial properties can easily be written by including hypotheses such as `x~==Undef`. For example, since the operator `&` is non strict in its second argument, the commutativity of `&` only holds provided its second argument is not undefined:

```
y ~== undef |- x & y => y & x
```

Opening up definitions

As with rewrite rules, the main difficulty with the use of definitions is to ensure that they are not opened up indefinitely. Three heuristics are used to control the opening up of definitions, the first simply opens up non-recursive functions, the second opens up recursive functions provided a *measured subset* of their arguments

[5]See [12] for details.

are *explicit values* and the third 'tentatively' opens up definitions and rewrites the resulting expression to see if the new version is an improvement on the old.

A measured subset of a functions arguments is any subset for which there exists a measure and well founded relation such that for each recursive call of the function in its definition body, the measure applied to the parameters of the recursive call is less than the measure applied to the formal parameters of the function according to some well founded relation.

An explicit value is a term composed of shells and constants (it contains no variables or functions), and so opening up a function which has explicit values in a measured subset of its arguments is safe in that it cannot be opened up indefinitely (since each time it is opened up the measure must decrease according to a well founded relation).

When a recursive definition is tentatively opened up the actual parameters are substituted for the formal parameters in the definition body and the function's name is added to a list of functions being opened up before the definition body is simplified. During this simplification, no attempt is made to open up any of the functions that are already being tentatively opened up. Once the definition body has been simplified, a check is made for each recursive instance of the function which remains. If there are no such instances or if each instance is "good" then the new version of the function application is used to replace the old version, otherwise the old version is retained. A recursive instance of the function being expanded is "good" in comparison to the original function, provided one of the following conditions hold:

No new terms: There are no new terms in any of the functions arguments. That is, if each parameter of the recursive instance is a term that already appears in the conjecture being proved then the instance is good.

More explicit values: There are more arguments to the function which contain explicit values.

Less complex controllers: The symbolic complexity of some measured subset of the arguments is smaller. Symbolic complexity is measured as the number of function symbols in the argument (but the symbolic complexity of an IF function is the maximum of the symbolic complexities of its two arguments).

The motivation for the 'no new terms' heuristic is that it has a normalising effect on the terms that appear within a conjecture; it encourages terms to be expressed in the same way as other terms in the formula. This is particularly relevant to proofs by induction because the induction conclusion usually needs to be simplified to a form similar to that of the induction hypothesis so that the induction hypothesis can be used to prove the conclusion. For example a simple induction step has the form:

$\mathcal{P}$ (d x) |- $\mathcal{P}$ x

If $\mathcal{P}$ contains recursive functions with x as their argument then opening up those functions in the conclusion will hopefully produce the term (d x) which should be retained wherever possible since it appears in the hypothesis instead of x. Opening up the same functions in the hypothesis is not desirable since terms of the form (d x) will be replaced by (d (d x)) which are less likely to be useful for proving the conclusion.

There are two major differences between BMTP's and the theorem prover's treatment of definitions. Firstly the property prover allows local definitions whose scope is limited to the body of the definition in which they appear and secondly the property prover allows mutually recursive definitions.

The property prover's treatment of local definitions is straightforward since local definitions may only appear as part of a function definition and not at arbitrary points in the definition body. The scope of local definitions therefore includes the whole of the body of the definition to which they are attached. When opening up definitions which contain local definitions the actual parameters are substituted for formal parameters in both the definition body and in the local definitions. The local definitions are then assigned new names (ones that do not appear else where in the conjecture or current axioms and definitions); the new names are substituted for all occurrences of the local function names in the definitions body and the local definitions are added to the property provers set of current definitions. If an identical definition already exists then the new name of the local definition is changed to the existing definition. This means that the theorem prover need not waste time proving that two functions with identical definitions but different names are indeed identical; a situation which occurs when a function with local definitions is opened up more than once for the same set of arguments.

In dealing with mutually recursive definitions, the concept of a *recursion group* is useful. This is a group of functions in which each definition ultimately depends upon the definitions of all the other functions in the group. A function definition F immediately depends on definition G if G occurs in the definition body of F. A function definition D_1 ultimately depends on definition D_n if there is any sequence of function definitions

$$D_1, \ldots, D_n$$

such that D_i immediately depends on D_{i+1} for all $1 \leq i < n$. In other words, a mutual recursion group is a set of interdependent functions.

Mutually recursive definitions are opened up using heuristics that are mostly the same as normal recursive definitions. In particular, when simplifying the body of a tentatively opened up function F, other definitions may also be tentatively opened up, including ones in the same recursion group as F, but F itself may not be opened up a second time. Also, when looking for the "good" property that no new terms are introduced when a function is tentatively expanded, the "no new terms" condition must apply to calls of functions in the recursion group as well as to direct recursive calls to the function being expanded.

The main difference between the opening up of mutually recursive and normal recursive definitions is an additional heuristic which adds another "good" property for the recursive calls of a definition that is being tentatively opened up.

This heuristic states that if opening up a mutually recursive definition means that there are more functions in the recursion group of the opened up function that occur at least once in the conjecture then the opened up version should be retained. The motivation for this heuristic is that it helps prepare the conjecture for inductions which involve mutual recursion. The proof of many properties which involve only a subset of a mutual recursion group frequently relies on other properties associated with the remainder of of the recursion group. These properties are usually all interdependent in the same way as the functions are interdependent, and hence must all be proved simultaneously. By opening up definitions in the recursion group until the whole group appears in the conjecture, the conjecture is usually restated in a form that subsumes all the appropriate properties.

This heuristic can potentially lead to a non terminating sequence of simplifications. This occurs when the opening up of a function definition to introduce an additional member of the recursion group, results in another member of the recursion group being removed through simplification. To prevent this occurring indefinitely, a function definition may only be opened up in this way once, without and intervening induction taking place.

A simplification example

Different aspects of the example introduced here are used in several parts of this chapter to demonstrate the heuristics of the property prover. In fact it makes use of the majority of BMTP's heuristics as well as the heuristics that are unique to the property prover.

The example is based on a tree data structure, defined as the algebraic type:

```
tree * ::= Node * [tree *]
```

To improve readability the tree and list selector functions are replaced by the following function names:

```
SEL-Node-1  ==  tval
SEL-Node-2  ==  subts
   SEL-:-1  ==  hd
   SEL-:-2  ==  tl
```

The functions rtree and rlist are defined over this type; rtree reverses a tree and rlist reverses a list of trees.

```
rtree (Node v l) = Node v (rlist l)

rlist [] = []
rlist (a:x) = rlist x ++ [rtree a]
```

The compiled versions of these functions are:

```
rtree a = ifdef a (Node (tval a) (rlist (subts a)))

rlist x
    = ifdef x
        (IF (x==[]) []
            (rlist (tl x) ++ [rtree (hd x)])
```

where ifdef is defined as follows:

```
ifdef x e = IF (x==undef) undef e
```

The built in definition of the list append operator is:

```
a ++ b
    = ifdef a
        (IF (a==[]) b
            (hd a:(tl a ++ b))
```

Finally the conjecture to be proved is:

```
rtree (rtree a) == a
```

It is important that before making an induction on this conjecture, an attempt is made to rearrange it to include both of the functions in rtree's mutual recursion group. Using the additional heuristic specific to the property prover, which allows a function to be opened up if this results in a conjecture containing more functions from the recursion group, the inner application of rtree is opened up and the result simplified to the formula:

```
rlist  (rlist x ++ [rtree a]) == a:x
```

which is ideally suited to a mutual induction on a and x.

This initial part of the proof of rtree (rtree a) == a is used here as an example of the simplification heuristics. The induction argument produced from the above clause and the use of many of the other heuristics to prove this conjecture form the basis of examples in the remainder of this chapter.

Starting with the original conjecture, the inner application of rtree is tentatively opened up giving:

```
rtree (ifdef a (Node (tval a) (rlist (subts a)))) == a
```

The function rlist is then tentatively opened up, however none of the recursive calls introduced are 'good' (both functions in the recursion group are already present), so the opened up definition is abandoned in favour of the original version. The opened up body of (rtree a) can be simplified no further, and is 'better' than (rtree a) because it introduces a function in the recursion group of rtree that was not previously present in the conjecture. The resulting opened up body is therefore kept and the clause split into two cases corresponding to the IF in ifdef. These cases are a==Undef and a~==Undef. The actual implementation opens up ifdef to 'IF (a==Undef) Undef ...' and then splits the conjecture. To make the proofs clearer, ifdef is never explicitly opened up, but is frequently split into the cases a ~==Undef and a==Undef. The resulting two clauses are:

```
{ a ~==Undef, rtree Undef == a} &
{ a == Undef, rtree (Node (tval a) (rlist (subts a))) == a}
```

Rather than writing clauses in the above form, they are written using the 'proves' operator. For example`{A}` is written simply as `A`, `{A,B}` is written `(~A |- B)` and `{A,B,C}` is written `(~A & ~B |- C)`.

Case `a==Undef`

```
a==Undef |- rtree Undef == a
```

Whilst simplifying the conclusion the hypothesis can be assumed true, providing additional type set information. The property prover actually achieves this by assuming the other literals in a clause false whilst simplifying a literal. Since the clause form contains the hypotheses as negated literals, assuming them false is equivalent to assuming the hypothesis true.

In the above example, opening up '`rtree Undef`' produces `Undef` and under the assumption of the hypothesis, the resulting conclusion `(Undef==a)` simplifies to `T`, proving the clause.

Case `a~==Undef`

```
a~==Undef
|-
    rtree (Node (tval a) (rlist (subts a))) == a
```

The outermost `rtree` application is tentatively opened up and simplified, removing the `ifdef` (since the argument is not `Undef`). During this simplification, the function `rtree` (which is being opened up) is still considered part of the conjecture, so any attempt to open up '`rlist (subts a)`' fails because it does not introduce any functions in the recursion group that were not present before the expansion. The opened up and simplified version of `rtree` is retained because the only recursive call is 'better' due to the reduced symbolic complexity (now only two functions compared to five previously).

```
a~==Undef
|-
    Node (tval a) (rlist (rlist (subts a))) == a
```

When a new attempt to simplify the conjecture is made, the innermost `rlist` is successfully opened up because the other function in the recursion group `rtree` is no longer in the clause.

```
a~==Undef
|-
    (Node (tval a)
          (rlist
              (ifdef (subts a)
                     (IF (subts a==[]) []
                         (rlist (tl (subts a)) ++
                         [rtree (hd (subts a))]
                     )))))
    == a
```

The two 'if' conditions are used to split the conjecture into three cases, one for `(subts a == Undef)`, the other for `(subts a ==[])` and finally one for `REC-: a`. The following three cases are produced:

CASE `(subts a == Undef)`

```
a~==Undef &
subts a == Undef
|- Node (tval a) Undef = a
```

CASE `(subts a == [])`

```
a~==Undef &
subts a == []
|- Node (tval a) [] = a
```

CASE `Rec-: (subts a)`

```
a~==Undef
Rec-: (subts a)
|-
    (Node (tval a)
          (rlist  (rlist (tl (subts a)) ++
                   [rtree (hd (subts a))]
                 )))
    == a
```

All three clauses cannot be simplified further, and are therefore passed on to the next heuristic which is the instantiation of the view. Since this conjecture contains no free functions, there is no view to instantiate. However the next heuristic, elimination of destructors, is particularly relevant to this example. It re-expresses clauses in terms of constructors rather than destructors; as well as allowing the first two cases above to be simplified to `T`, it replaces `(hd (subts a))` with the new variable `b`, `(tl (subts a))` with `y`, `(tval a)` with `v` and `a` with `(Node v (b:y))` producing:

```
Node v (rlist (rlist y ++ [rtree b])) == Node v (b:y)
```

5.4.3 Instantiating the View

Before attempting to prove any properties the library retrieval system will attempt to establish a view between the two models which equates every type described by the re-users model with a type in the component's model and every function in the re-users model with an expression made up from functions in the component's model. This view is represented as a rewrite rule of the form:

`f` $\mathtt{p}_1 \ldots \mathtt{p}_n$ `=> rhs` so that if `f` appears in the conjecture applied to n or more arguments then an attempt is made to rewrite `f` and the n arguments to its corresponding value in the library model (**`rhs`**). The definition of a permissible view ensures that the rule does not generate an infinite sequence of rewrites.

Since this view is established using type information (as described in chapter 4, section 4.4.3) there may be re-users model functions which are not included in the view. In this case the property prover is given specially marked *free functions* in place of the names of these functions. The property prover then attempts to instantiate the free functions using an additional simplification phase which applies the normal simplification heuristics described above but also allows the free functions 'to become instantiated'. This instantiation process is achieved simply by adding an appropriate rewrite rule for the free function. This rewrite rule also corresponds to the view for that particular function, and so any instantiation is restricted by the rule that it must produce a syntactically permissible view for the function that is instantiated. A view is guaranteed to be permissible (it is syntactically permissible) provided it has the form `f` $p_1 \ldots p_n$ `=>` **`rhs`** and:

- `f` is the free function
- p_1 to p_n are either variables or explicit value templates
- **`rhs`** contains no functions from the re-users model and no variables that do not occur in at least one of the templates $p_1 \ldots p_n$ (however, free functions other than `f` are allowed provided they currently have no view).
- The types of `f` $p_1 \ldots p_n$ and **`rhs`** match.
- There is no conflicting view for `f` already in existence.

Given two views of the same function:

```
f p1 .. pn => rhs1
f q2 .. qm => rhs2
```

the views conflict unless there are a corresponding pair of parameters **`pi`** and **`qi`** which are both *explicit value templates* and `pi ~== qi`. An explicit value template is a term which contains only constants, shells and variables; it is the property prover's equivalent of Miramod patterns.

The ability to build up the view of a function as a number of non conflicting rules means that a view can be based on information obtained at different stages of the property proving process.

Since the property prover language admits higher order functions, a free function applied to zero or more arguments may be matched against any term (even if it is not a function application) provided the resulting view is syntactically permissible. For example '`f a b c`' matches the term `t` provided the view '`f a b c => t`' is syntactically permissible. The converse also applies, so `f` matches '`t x y z`' provided the view '`f == t x y z`' is syntactically permissible.

The view instantiation phase is a repeat of the simplification phase previously described except that it allows for the instantiation of free functions at the following points:

- Whilst simplifying strong and weak equalities.
- Whilst attempting to find rewrite rules applicable to a term which contains a free function.
- Whilst attempting to open up function definitions.

When these instantiations occur, the current state of the theorem prover is recorded as a choice point. After the clause has passed through the new simplification phase a check is made to see if the resulting clause has been simplified to T. If it has, then the choice points are removed and theorem proving continues with the newly established part of the view 'fixed'. If on the other hand the clause does not simplify to T under the new instantiations then the theorem prover backtracks to the most recent choice point and continues as if the instantiation had failed to produce a permissible view. The heuristic terminates in one of three ways: when it produces the true clause; when there are no more choice points, or when the heuristic 'times out' because there are too many possible instantiations. When the heuristic produces a true clause then the theorem prover returns to the top of the 'waterfall' to find another clause for simplification. In the other two cases, the original clause passed to the heuristic is handed on to the following stage of the theorem prover and any views created whilst trying to prove the clause by instantiating free functions are removed. The 'time out' from this phase is based on the time allowed to prove the current property and the number of clauses remaining at the top or bottom of the waterfall. The reason for insisting that any instantiations result in the current clause being proved true is that this constrains any backtracking behaviour to a small section of the theorem proving process.

The property prover can also backtrack to the most recent instantiation if it simplifies a term to `T` when it is trying to produce `F` or vice versa. Whenever the term being simplified is a property statement (with possible values `T` or `F`), the property prover is either trying to rewrite to `T`, trying to rewrite to `F` or indifferent. For example when simplifying the literal of a clause or the hypothesis of a rewrite rule the property prover is attempting to simplify to `T`, but when simplifying the condition of an `IF` application then the property prover is indifferent. If an instantiation occurs whilst attempting to rewrite to x (where x is `T` or `F`) and the resulting value is ($\tilde{}x$) then the theorem prover backtracks, abandoning the most recent instantiation involved in rewriting the term.

Rewriting Equalities

If the property prover is trying to rewrite the equality `l == r` to T then an attempt is made to match `l` and `r` by instantiating free functions. For example, if `l` is a free function and `r` is a term made up exclusively from component model functions, then `l` and `r` match under the view that `l => r`. If this match is successful then the equality is rewritten to `T`, otherwise `l` and `r` are rewritten using instantiations. If the resulting expressions are definitely unequal then the prover backtracks until there are no more choice points in the simplification of l and r, or until a non `F` result is produced. The equivalent process is carried out when attempting to prove that inequalities are `F`.

Using Rewrite Axioms

If a free function *properly occurs* in the term being rewritten (ie the term being rewritten contains a free function but is not itself a free function) and is not a free function applied to terms which are all variables then an attempt is made to match the term against all the current rewrite rule left hand sides. During this match, any free functions are allowed to match arbitrary terms provided the resulting view is permissible.

These restrictions are necessary because the intention of the heuristics is to instantiate the view according to the form of the conjecture being proved. The instantiation of a free function on the basis that it matches the left hand side of a rewrite rule and that the resulting view is permissible does not involve the context of the free function and therefore does not base the instantiation on the form of the conjecture being proved.

If one of the rewrite rules variables properly occurs in the term matched by a free function then a new free function name is created and substituted for the variable in the rewritten expression as well as the right hand side of the view. On the other hand, if a free function matches a rewrite rule variable then the free function is simply substituted for the variable and no addition to the view is made.

Opening up definitions

Finally, if the term being rewritten is of the form '`f p1` ...pn' where f is a free function then `f` becomes instantiated to `d` provided `d` is defined with `m` parameters where $m \leq n$:

```
d q1 .. qn = body
```

and there is at least one case from the cases of `body` such that all the conditions in the case can be rewritten (without instantiations), to `T`.

The cases of an expression are defined by the following rules:

1. The cases of an if expression '`IF c a b`' are the union of condition `c` added to each of the cases of `a` and `~c` added to each of the cases of `b`.
2. Any other expression has only one case, which is the empty case.

The motivation for these restrictions is the same as for the restrictions associated with rewrite rules; they ensure that the view is only instantiated if there is an indication that the particular instantiation will simplify the resulting clause.

5.4.4 Eliminating destructors

The heuristics covered in this section are exactly the same as the corresponding heuristics of BMTP and therefore they are only outlined in terms of a few examples. The full details can be found in Boyer and Moore's book[12].

The motivation for eliminating destructor terms such as `(SEL-:-1 s)` is that it is easier to prove theorems in terms of constructors than it is to prove them in terms of destructors. For example if a conjecture contains the terms `(SEL-:-1 s)`, `(SEL-:-2 s)` and `s` then the conjecture containing the corresponding terms `h`, `t` and `(h:t)` is easier to prove since facts such as '`REC-: (h:t)`' are immediately apparent whereas '`REC-: s`' must be explicitly recognised in the hypothesis. The same process is used to trade undesirable terms such as `(a div b)`, `(a mod b)` and `a` for more desirable alternatives such as `d`, `m` and `d*b+m` where `m < b` and `b~==0`.

The property prover relies on the existence of elimination lemmas for guidance as to which terms are 'undesirable' and how to eliminate them. As well as this, there is an elimination lemma associated with every shell, which allows the shells selector functions (destructors) to be eliminated in favour of the shell (which is a constructor). The list shell '`:`' is one such example:

```
REC-: s |- (SEL-:-1 s):(SEL-:-2 s) == s
```

Given the above elimination lemma the theorem prover attempts to prove the conjecture '$\mathcal{P}$ `(SEL-:-1 s) (SEL-:-2 s) s`', by splitting it into the following two conjectures:

```
    ~REC-: s |- 𝓟 (SEL-:-1 s) (SEL-:-2 s) s
```
and
```
    𝓟 h t (h:t)
```

where `h` and `t` are new variables that did not previously occur in the conjecture.

An example of an elimination lemma that is not introduced automatically by the shell principle is:

```
b~==0 |- ((a div b)*b + a mod b) == a
```

This can be used by the property prover to replace `(a div b)`, `(a mod b)` and `a` with the terms `d`, `m` and `d*b+m` under the hypothesis `b~==0`. In fact the additional hypothesis `m<b` is also added by the property prover as part of the generalisation process described in section 5.4.6.

Elimination example

To continue the example of the proof of '`rtree (rtree a) == a`', the simplification heuristics leave three clauses:

```
Case (subts a == Undef)

    a~==Undef &
    subts a == Undef
    |- Node (tval a) Undef = a


Case (subts a == [])
```

```
      a~==Undef &
      subts a == []
      |- Node (tval a) [] = a
```

Case Rec-: (subts a)

```
      a~==Undef
      Rec-: (subts a)
      |-
          (Node (tval a)
                (rlist  (rlist (tl (subts a)) ++
                          [rtree (hd (subts a))]
                       ))
          == a
```

All three clauses contain the terms (subts a) and (tval a) so although each would be treated individually by the property prover, the following discussion applies to all three.

When the type 'tree *' is introduced, the following elimination lemma is added to the set of axioms:

```
Rec-Node a |- Node (tval a) (subts a) = a
```

This lemma can be used to replace (tval a) with v, (subts a) with s and a with (Node v s), where v and s are two new variables, provided 'REC-Node a' is true. This usually means generating two new clauses, one for when 'REC-Node a' is true and the other for when it is false (in which case the elimination cannot occur). However, all the above clauses have the hypothesis a~==Undef from which the typeset of a can be deduced as <u>Node</u> and therefore the possibility that 'REC-Node a' is false need not be considered. Also, since the term a~==Undef becomes 'Node v s ~== Undef', it can be simplified to T and removed from the hypothesis. The three resulting clauses are:

Case `(s == Undef)`

```
    s == Undef |- Node v Undef = (Node v s)
```

which simplifies to `T`,

Case `(s == [])`

```
    s == [] |- Node v [] = (Node v s)
```

which also simplifies to `T`, and finally

Case `Rec-: s`

```
    Rec-: s
    |-
        (Node v (rlist  (rlist (tl s) ++ [rtree (hd s)])))
        == (Node v s)
```

which can be simplified to:

```
Rec-: s
|-
    (rlist  (rlist (tl s) ++ [rtree (hd s)] )) == s
```

Destructor elimination can be applied again, using the list construction elimination lemma:

```
Rec-: s |- (hd s):(tl s) == s
```

The substitutions `(hd s)` for `a`, `tl s` for `x` and `s` for `(a:x)` are made; the non elimination case is simplified to `T` and the hypothesis `REC-: (a:x)` in the elimination case is simplified to `T`, giving:

```
rlist  (rlist x ++ [rtree a]) == a:x
```

This clause is now ready for induction on the variables x and y, since it contains both functions in the mutual recursion group.

5.4.5 Using equalities

This heuristic removes equality hypotheses from the conjecture by substituting one side of the equality for the other in the remainder of the conjecture (uniform substitution), or in specific parts of the conjecture (cross fertilisation). The motivation of this heuristic is that if the previous simplification steps have failed to prove the conjecture then induction will be necessary, the resulting induction hypothesis will be drastically weakened by the presence of any equalities. For example, given a conjecture of the form 'p |- q' where p is an equality, the resulting induction step will have the form:

```
   (p' |- q')
|- (p |- q)
```

where '(p |- q)' is the induction hypothesis and p' and q' are the new versions of p and q produced by the induction step. To prove this the property prover must prove two clauses:

```
~p' |- (p |- q) &
 q' |- (p |- q)
```

Since ~p' is usually an inequality, it provides an extremely weak basis from which to prove (p |- q) and therefore this situation should be avoided whenever possible.

Before attempting induction on a clause, the property prover checks the clause for any hypotheses which are equalities of the form s==t where t occurs elsewhere in the clause and is not an explicit value. If the current clause is the result of an induction step and t also occurs on the right hand side of another equality literal in the clause then cross fertilisation of s==t is carried out, otherwise s is uniformly substituted for t in the whole clause. When cross fertilising, equality literals have s substituted for t in the right hand side only. Other literals have s uniformly substituted for t.

Once the substitutions have taken place, the equality s==t is removed from the clause and the new clause is returned to the top pool of the waterfall to be re-simplified. This is important as it ensures that only one equality is used and thrown away at a time, and since the heuristic is a risky one (it may produce a non theorem from a theorem) it is best to attempt to prove the result of the use of one equality with the lower risk heuristics before trying the higher risk heuristic again.

Equality hypotheses can also be used in the other direction, provided the appropriate conditions are met. In this case, cross fertilisation takes place on the left hand side of equalities rather than the right hand side.

Cross fertilisation example

Continuing with the 'rtree (rtree a) == a' example, the induction heuristics are applied to the previously produced conjecture, and the result of the induction is simplified and the appropriate destructors eliminated. This produces two clauses which must both be proved by a further appeal to induction. One of these is:

```
rlist (rlist y ++ [rtree b]) == b:y &
rlist (rlist z ++ [rtree c]) == c:z
|-
   rlist ((rlist y ++ [rtree b]) ++ [Node v (rlist z ++ [rtree c])])
   == (Node v (c:z)):(b:y)
```

There are now two cross fertilisations that can take place, one based on (b:y) and the other on (c:z). Cross fertilisation is chosen because both occur on the right hand side of another equality literal. The left hand side of the two hypotheses' equalities are substituted for their corresponding right hand sides in the right hand side of the conclusion equality, and the hypotheses are thrown away, giving:

```
rlist ((rlist y ++ [rtree b]) ++
         [Node v (rlist z ++ [rtree c])])
== (Node v (rlist (rlist z ++ [rtree c]))):
   (rlist (rlist y ++ [rtree b]))
```

This example provides the motivation for the next heuristic to be applied. The terms `(rlist y ++ [rtree b])` and `(rlist z ++ [rtree c])` have now served their purpose and are no longer relevant to the proof other than as 'place holders'. Thus they can be replaced by new variable names, giving a much simplified conjecture on which to base the induction which will follow.

```
rlist (x ++ [Node v w]) == (Node v (rlist w)): (rlist x)
```

5.4.6 Generalisation

The power of induction as a method of proof lies in the assumption of instances of the conjecture to be proved, thus the more general the conjecture to be proved, the more general the assumption on which its proof can be based.

To prove a conjecture by induction, it is often necessary to prove instead a more general conjecture from which the truth of the original conjecture follows. For the most part, discovering such generalisations is a task that requires creativity; in other words it is a task that cannot currently be automated. The property prover therefore relies on the library model writer to provide as axioms (or prove as lemmas) such generalisations. Despite this there are some common generalisations that can be produced automatically. These generalisations replace occurrences of terms such as `(f p1 ... pn)` with a new variable and add any appropriate hypotheses that can be derived from `(f p1 ... pn)` but not from the new variable.

The generalisation heuristic looks for common subterms of two or more literals in clauses which are produced by a previous induction step. Terms which are variables, explicit value templates, destructor applications or equalities are not generalised. Since a previous cross fertilisation may have resulted in one such subterm being moved into the only other literal that contains the subterm, common subterms on either side of an equality literal are also accepted. Each occurrence of the common subterm is then replaced by a new variable that does not appear in the clause before generalisation took place.

There are many situations in which this heuristic can produce a non theorem from a theorem because there are properties that can be derived from the term being generalised that are not explicitly mentioned in the hypothesis of the theorem. In this case, when the term is replaced by a variable in the generalised theorem, these properties are neither part of the hypothesis or derivable from the term (which is now a variable), and the theorem may well have become a non theorem. An example of this is the term '`x mod y`'. A lemma that can be proved from the definition of `mod` and can also be used as a rewrite rule is:

```
(x mod y < y) == (y~==0)
```

If '`x mod y`' is generalised to a variable, say `v` then the fact that '`(v < y) == (y ~==0)`' is no longer apparent, unless it is explicitly recorded as part of the hypothesis.

To prevent this type of over generalisation occurring, the property prover relies on generalisation axioms supplied by the library model writer to provide hypotheses concerning the terms being generalised. These generalisation axioms are simply property expressions containing one or more subterms that can be generalised (under the rules given above). For example the following generalisation axiom can be used in the generalisation of '`x mod y`' as well as '`x mod y < y`':

```
(x mod y < y) == (y~==0)
```

When the property prover generalises a term by replacing it with a new variable, it first checks to see if there are any generalisation axioms that contain terms that are the same as the term being generalised under some substitution instance. If any are found the substitution s associated with each match between a term and a generalisation axiom is applied to the whole generalisation axiom and the resulting term is added as a hypothesis to the generalised clause. Once all such hypotheses have been added, the new variable is substituted for occurrences of the term being generalised and the resulting clause is returned to the top pool of the waterfall.

This restriction of generalisations using generalisation lemmas to suggest additional hypotheses is also used by the destructor elimination heuristic described previously.

5.4.7 Eliminating irrelevance

The final heuristic used before induction is one which eliminates irrelevant literals from a clause. This is done for two reasons, firstly it removes terms that would otherwise be in contention for the forthcoming induction and would at best simply slow down the selection of an appropriate induction whilst in the worst case such terms could become the basis of the next induction. Secondly the elimination of irrelevant terms allows the property prover to conclude (or at least guess) that a property is not true, by eliminating all the literals in a clause.

To decide which literals to eliminate, all the literals in the clause are partitioned into sets of literals which have common variables. If one of two conditions holds for a partition then the literals in the partition are removed from the clause.

1. If a partition contains no recursive functions then it is eliminated.
2. If a partition contains only a term of the form `(f` $v_1 \ldots v_n$`)` or `~(f` $v_1 \ldots v_n$`)` where '$v_1 \ldots v_n$' are distinct variables then it is eliminated.

The motivation for these elimination conditions is as follows: If a partition contains no recursive functions then it can add no information to any of the literals in other partitions and since it has not been proved by simplification it is probably false (and certainly can never be proved true by the property prover). If a partition contains only a term of the form (f v_1 ...v_n) or ~(f v_1 ...v_n) then since it has not been proved true it is most probably false for some instances of 'v_1 ...v_n'.

5.4.8 Induction templates

The induction heuristics of BMTP (summarised in section 5.2.4) are in two main parts, one which occurs at function definition time and the other at the time of induction. The current implementation of the property prover does not automate the definition time heuristics and relies on the library model writer to supply the appropriate information in the form of induction templates. The induction time heuristics employed by the property prover are for the most part similar to the BMTP heuristics, the major difference being the property provers ability to invent appropriate inductions for conjectures which contain mutually recursive functions.

Before describing the heuristics for inventing inductions appropriate to a particular clause and for choosing between a collection of such functions, the form of the induction templates which are provided by the component model writer and used by these heuristics is described. Miramod does not provide a syntax for these templates since the intention is ultimately to derive them from function definitions and induction axioms in the same way as BMTP. Induction axioms are not described here because the current implementation of the property prover uses the ready made induction templates instead.

Each induction template is associated with a recursive function. It suggests a possible induction schema for clauses which contain one or more applications of the function to variables that will be used as induction variables. In this thesis, induction templates are written in the following manner:

Function and formals: f x_1 ...x_n
Recursion group: { ...}
Measured subset: { ...}
Changeables: { ...}

Unchangeables: $\{ \ldots \}$

Case 1:

Condition: c_1

Substitutions: $\{<x_{i_{1,1}}, t_{1,1}>, <x_{i_{1,2}}, t_{1,2}> \ldots \}$

$\{<x_{i_{2,1}}, t_{2,1}>, <x_{i_{2,2}}, t_{2,2}> \ldots \}$

...

Case 2:

...

The first line of the template gives the name of the function to which the template applies and the names of the formal parameters of the function. The second line gives the names of the functions in the same recursion group as the template function (including the template function itself). For straightforward recursive functions (as opposed to mutually recursive functions) this set contains only the template function. The "Measured subset" is a set of formal parameters that decrease according to the same measure and well founded relation for every substitution in the case analysis, under the assumption that the corresponding case condition is true. The *changeables* are those members of the measured subset that are changed by at least one substitution in the case analysis and the *unchangeables* are those members of the measured subset that are not. The significance of the changeables and unchangeables is that the induction template only applies to a term if all of the changeables are distinct variables and none of them occurs among the unchangeables (which need not be variables).

For example, consider the function `(range a b)` which produces a list of integers in the range `a` to `b` inclusive. The compiled definition of such a function might be:

```
range a b = ifdef (a<b)
                  IF (a<b ==True) (a:range (a+1) b) []
```

A measure over which the recursion can be shown to terminate is `(b-a)`, which includes both variables despite the fact that only one of them changes in the recursive call.

Finally the case analysis, consisting of one or more cases, is given. Each case corresponds to one induction step in the induction suggested by the template, and is made up from a condition under which the induction step must be proved and a set of one or more substitutions. Each substitution suggests the induction hypothesis obtained by applying the substitution to the conjecture being proved, and the induction step produced by the case assumes each hypothesis produced by a substitution belonging to the case.

5.4.9 Creating Induction Schemes

When all the clauses have been simplified as much as possible by the preceding heuristics, one of the remaining clauses is picked, an induction scheme is invented for it and the resulting clauses are simplified. The 'invention' of an induction scheme is achieved in the following stages:

1. All of the induction templates in the component's model are used in conjunction with the clause to produce a set of plausible induction schemes.
2. Subsumed induction schemes are removed.
3. Wherever possible, schemes are merged.
4. Flawed schemes are discarded.
5. Scoring functions are used to pick between the remaining schemes.

For each function application that appears in the clause, the corresponding induction templates are retrieved and a check is made to see if the template applies to the actual parameters of the function. Firstly the template is instantiated with the actual parameters to which the function is applied. Provided all of the changeables are distinct variables which do not occur in the unchangeables, the changeables and unchangeables fields are replaced with two fields, one for the *changing variables* and the other for the *unchanging variables*. These are the set of variables that are substituted for, or not substituted for respectively in the induction scheme and include unmeasured as well as measured variables. Finally the instantiated template is then added to the list of schemes .

The list of induction schemes is then shortened by removing any that are subsumed[6] by other schemes in the list.

Merging Schemes

The remaining induction schemes are then checked to see if any can be merged. Boyer and Moore use the following example to demonstrate the merging heuristics (some of the details have been modified to suit the property prover notation):

The function `lessp` is defined as:

```
lessp x y == IF (y==0) F
             IF (x==0) T
                       lessp (x-1) (y-1)
```

and it has two induction templates:

Function and formals: `lessp x y`
Recursion group: `lessp`
Measured subset: {`x`}
Changeables: {`x,y`}
Unchangeables: {}
Case 1:
    Condition: `x~==Undef & x~==0`
    Substitutions: {`<x,x-1>,<y,y-1>`}

and

Function and formals: `lessp x y`
Recursion group: `lessp`

[6]The subsumed relation is described in detail by Boyer and Moore.

Measured subset: {y}
Changeables: {x,y}
Unchangeables: {}
Case 1:

    Condition: `y~==Undef & y~==0`
    Substitutions: `{<x,x-1>,<y,y-1>}`

Suppose we are trying to prove the transitivity of lessp:

```
lessp i j & lessp j k |- lessp i k
```

both induction templates apply to all three occurrences of lessp, giving six candidate induction schemes. All six schemes are different either because they result from a different function application and therefore substitute for different variables or because they are based on a different template and therefore have different conditions. Taking as an example the scheme which assumes the conjecture with `i` replaced by `(i-1)` and `j` replaced by `(j-1)` along with the condition `i~==Undef & i~==0`. If this scheme is used for the induction then the term `(lessp i k)` will appear in the conclusion and its corresponding term in the hypothesis will be `(lessp (i-1) k)`. Unfortunately there is then no way of opening up the definitions in the conclusion so that the hypothesis term `(lessp (i-1) k)` is produced since opening up `(lessp i k )` will produce `(lessp (i-1) (k-1))`. In effect the induction has thrown the lessp terms involving `k` "out of sync" by not substituting for `k` as well as `x` and `y`.

The answer is to merge all six induction schemes into one:

```
Accounts for: {lessp}
Changing vars: {i,j,k}
Unchanging vars: {}
Case 1:
      Condition: i~==Undef & i~==0 &
                 j~==Undef & j~==0 &
                 k~==Undef & k~==0
      Substitutions: {<i,i-1>,<j,j-1>,<k,k-1>}
```

BMTP insists that (among other things): the intersection of the changing variables of the two merged schemes must be non empty and the intersection of the changing variables of one scheme with the unchanging variables of the other must be empty.

Flawed Induction Schemes

An induction scheme is said to be flawed if any of its induction variables are changing or unchanging variables of one of the other schemes that are candidates for induction. An induction variable of a scheme $\mathcal{S}$ is one of the changeables of any of the templates which account for one of the terms accounted for by $\mathcal{S}$.

The property prover uses this definition of a flawed scheme to remove all such schemes from the set of candidates provided there is at least one scheme remaining. The reason for this is that since all the remaining schemes have failed to merge, a scheme whose induction variables are changing or unchanging variables of another scheme must disagree on those variables. For example:

```
(a++b)++c == a++(b++c)
```

Since the list append operator recursively changes its first argument and leaves the other argument fixed, three induction schemes are suggested, each accounting for one of the terms `a++b`, `a++(b++c)` and `b++c`. The first two merge giving an induction which replaces `a` with `(tl a)` in the conjecture and leaves `b` and `c` unchanged. The third gives an induction which replaces `b` with `(tl b)` in the conjecture and leaves `c` unchanged. Using the latter induction would be a heuristic mistake because the term `(tl b)` occurs as both the first parameter (a changeable) and the second parameter (an unchangeable) of the append operator.

```
(a ++ tl b) ++ c == a ++ (tl b ++ c)
|- (a ++ b) ++ c == a ++ (b ++ c)
```

Since opening up append will not change the value of its second parameter the term a++b in the conclusion cannot be opened up to produce the hypothesis term (a ++ tl b) and so the induction is flawed. The merged induction:

```
(tl a ++ b) ++ c == tl a ++ (b ++ c)
|- (a ++ b) ++ c == a ++ (b ++ c)
```

is perfect since the induction variable a does not appear as a changing or unchanging variable in the alternative induction scheme and so must always be opened up by ++ in the same way.

5.4.10 Completing the induction

If the result of all the above heuristics still leaves a choice of induction schemes, the 'best' one is selected according to a scoring function which is based on the number of terms to which the schemes applies and the ratio of formals substituted for against the number of formal parameters to the template.

The induction principle is then used to create a set of clauses from the induction scheme. If the scheme has n cases with conditions Q_i, number of substitutions h_i and substitutions $S_{i,1} \ldots S_{i,h_i}$ (where $1 \leq \text{i} \leq \text{n}$):

Conditions	Substitutions
Q_1	$S_{1,1}, S_{1,2} \ldots S_{1,h_1}$
$\ldots$	
Q_n	$S_{n,1}, S_{n,2} \ldots S_{n,h_n}$

the resulting set of clauses consists of a base case:

$\tilde{}Q_1$ & $\tilde{}Q_2$ & ...& $\tilde{}Q_n$ |- C

and the $i = 1..n$ induction cases:

Q_i & $C_{1,1}$ & $C_{1,2}$ ... C_{i,h_i} |- C

where $C_{x,y}$ is the result of applying substitution $S_{x,y}$ to the conjecture C and h_i is the number of substitutions in case i.

The logical operators &, \/, ~ and |- are then expanded to IF expressions which are in turn expanded to clauses before being returned to the 'waterfall' for simplification.

5.4.11 Induction on Partial and Infinite values

The heuristics for generating and selecting induction schemes described above do not specifically recognise the possibility of partial or infinite values.

Partial values are dealt with implicitly by the case conditions which usually exclude the undefined value from the induction cases — thus the truth of the conjecture when the induction variables are undefined is proved (or at least set aside to be proved with any other partial properties) as part of the base case.

If one of the induction variables includes infinite values in its quantification, then the induction principle may be unsound (may conclude that a non-theorem is true) if the conjecture on which the induction is taking place is not chain complete for that variable. Consider an induction argument of the form:

```
(~𝒬 x |- 𝒫 x) &
(𝒬 x & 𝒫 (d x) |- 𝒫 x)
```

If this argument has been proved true then it follows that $\mathcal{P}$ `x` holds for all finite and partial `x` since a proof can be constructed for any `x` using a finite sequence of induction and base cases (assuming the measure and well founded relation associated with `x` and `(d x)`). $\mathcal{P}$ `x` is proved for partial but finite `x` because for any partial `x`, $\mathcal{Q}$ `x` is either `T` or `F` and so either the base case or the induction case must hold. If x is infinite, $\mathcal{P}$ x does not necessarily follow from the proof of the induction argument, because the number of steps needed to construct a non inductive proof from the induction argument is not necessarily finite. If $\mathcal{P}$ x for all infinite x follows from $\mathcal{P}$ x for all finite and partial x then $\mathcal{P}$ x is chain complete. An example of a property that is not chain complete is:

```
{y+} y ~== ones
     where
          ones = 1:ones
```

which can be 'proved' using the induction:

```
(~REC: y |- y ~== ones) &
(REC: y & tl y ~== ones |- y~==ones)
```

but is clearly not true for all y including infinite y.

The property prover uses a number of syntactic conditions which are sufficient to ensure that a conjecture is chain complete with respect to the induction variable(s) [51]. If these conditions do not apply, the property prover continues with the induction, making a note of the fact that the property is only proved for finite cases. This will then be used to reduce the matching score for the model under consideration.

The property prover considers a clause to be chain complete in x provided each litteral is one of the following:

1. An equality litteral between Miranda expressions: `a == b`.
2. An equality litteral between litterals `a` and `b` where `a`, `~a`, `b` and `~b` are themselves chain complete.
3. An inequality of the form: `a ~== Undef` or `Undef ~== a`.

4. A negated shell recogniser for a shell in which none of the other shells in the type have arguments. For example: `~(REC-: t)`

5. A litteral in which `x` does not occur.

5.5 Summary

The property prover is based on natural deduction and the Boyer and Moore theorem prover. Natural deduction is used because the proving process must be easy for component model writers to follow so that they may incorporate in the library model dependent heuristics for proving theorems about the model. The choice of BMTP as a basis for the property prover is guided by the need to prove theorems about inductively defined data types.

The property prover stores the conjecture which it is trying to prove in clause form so that whilst simplifying one of the literals from which the clause is made up, the remaining literals in the clause may be assumed false. Proofs that require a non inductive case analysis are therefore achieved by splitting the conjecture into a number of clauses each of which represents one of the cases.

To prove a clause, it is passed through a series of heuristics which are ordered so that the lowest risk heuristics are applied first (the 'risk' of a heuristic is that it might produce a theorem from a non theorem). If any of these heuristics alter the clause then it is returned to the first heuristics, thus ensuring that the lowest risk heuristics are always applied whenever possible. The last of the heuristics is induction. The induction heuristics are based on the induction templates associated with particular functions. BMTP can automatically derive these templates from function definitions (provided the definition conforms to BMTP's principle of definition). This part of BMTP is currently not implemented in the property prover, which relies on the component model writer to supply appropriate induction templates. The most interesting heuristics of BMTP are the ones for producing an induction scheme for a particular clause by instantiating appropriate induction templates to produce a collection of plausible induction schemes, and then merging the schemes where ever possible before selecting the best one.

The property prover has several novel differences from BMTP which relate to its use as a model comparison tool rather than a theorem proving tool. These differences are centred on the following areas:

- Proving the properties of partial functions (partial properties).
- Proving the properties of functions defined using mutual recursion.
- The presence of local definitions in functions.
- View discovery.

The proof of partial properties is dealt with by associating a special value called `Undef` with each Miramod type. Since pattern matching definitions treat parameters with the value `Undef` as a special case[7] the property prover will separate the clauses which deal with the `Undef` cases from the other cases. These clauses are called partial clauses (since they represent partial properties) and are distinguished by the fact that they have at least one literal with an equality of the form `X==Undef` where `X` is an arbitrary term. When a partial clause is generated by one of the property provers heuristics it is added to a collection of partial clauses. The property prover makes no attempt to prove any of these by induction until all of the non partial clauses have been proved. The result returned by the property prover from its attempt to prove a property is a score which is composed of two parts, one for the partial properties and the other for the non-partial properties. In effect this system ensures that if a property can be proved under the assumption that all variables are completely defined and finite and that all functions terminate for finite input, then it returns a higher matching score than if it cannot, irrespective of the proof of the property without these assumptions.

Another novel feature of the property prover is its ability to generate views for free functions. This is achieved using the BMTP heuristics for rewriting terms and simplifying clauses, but with special versions of the heuristics for simplifying equalities, applying rewrite rules and opening up definitions. Since the view generation heuristics are risky in the sense that they may prevent properties from being

[7]Although this is not immediately apparent from the Miramod definition of the function, the compilation process converts definitions to a form that explicitly states the functions behaviour for undefined arguments if this differs from its behaviour for defined arguments.

proved by creating the wrong view, they are applied after the normal simplification heuristics. Restricted backtracking may occur if the view generated does not help to prove the clause in which the view was generated. View synthesis is therefore based on the properties the view should help prove rather than just the types of the functions that must be related by the view.

Chapter 6

Miramod compilation

The compilation of Miramod closely follows the accepted methods of compiling modern functional languages[35]. These involve translation to an intermediate notation, enriched lambda calculus, which is then transformed to progressively simpler forms of lambda calculus. This final form of lambda calculus (usually supercombinators) is then either interpreted or compiled to an appropriate (possibly abstract) machine language.

Since the objective of the Miramod compiler is to produce a property prover representation of the model suitable for theorem proving rather than execution, the compilation follows the standard methods only as far as the supercombinator form. The resulting supercombinators are then used directly by the property prover.

6.1 The Lambda Calculus

The Lambda Calculus is syntactically and semantically simple whilst being sufficient to express all computable functions. A lambda calculus expression is written using function application, denoted by juxtaposition, variables and lambda abstractions which are denoted $\lambda x.E$ where x is a variable and E is a lambda calculus

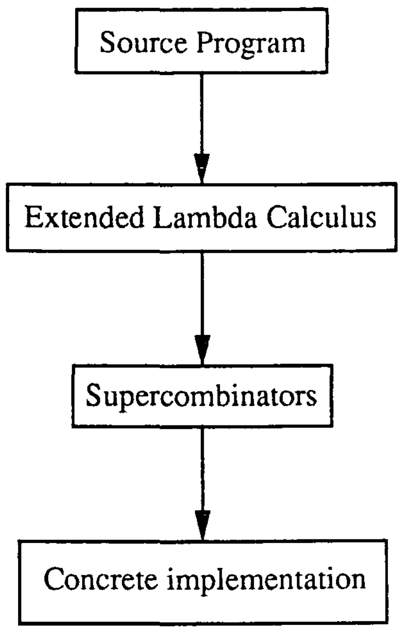

Figure 6.1: Functional Language Compilation

expression containing zero or more occurrences of the variable x.

A lambda abstraction $\lambda x.E$ denotes the function which takes one argument and returns the expression E with the actual parameter substituted for free occurrences of x in E. A variable is said to occur free in a lambda calculus expression provided there are no enclosing lambda abstractions which name the variable.

The semantics of lambda calculus can be summarised with the following three laws:

Alpha conversion: $\lambda x.E \overset{\alpha}{=} \lambda y.E_{[y/x]}$
provided y does not occur free in E.

Beta conversion: $(\lambda x.E)M \overset{\beta}{=} E_{[M/x]}$

Eta conversion: $(\lambda x.F\ x) \overset{\eta}{=} F$
provided F is a function and x does not occur free in F.

The expression $E_{[M/x]}$ is used to denote the expression formed by substituting M

for all free occurrences of x in E [1]

Two enrichments to lambda calculus are made so that the original translation into lambda calculus is straightforward. One is the *letrec* construction and the other is the pattern matching lambda abstraction.

A *letrec* expression is written:

```
letrec
    y = B
in E
```

and introduces a variable y which is bound to the value B in E and also in B itself. Letrec expressions also allow multiple definitions and hence mutually recursive definitions:

```
letrec
    y = ...x...
    x = ...y...
in E
```

The second enrichment to lambda calculus is the pattern matching lambda abstraction. This allows patterns, which may contain variables, in the place of the single variable usually associated with a lambda abstraction. For example the function `tl` which returns the tail of a list, can be written using the pattern matching lambda abstraction $\lambda(x : xs).xs$. If the argument to this lambda abstraction does not match then the special value `Fail` is returned, otherwise the variables in the pattern are bound to their corresponding values in the argument and the abstraction body is returned with the appropriate substitutions made:

$$(\lambda(\mathtt{x} : \mathtt{xs}).\mathtt{xs})\ [\,] \quad = \quad \mathtt{Fail}$$

[1]For the sake of simplicity, this definition does not mention the fact that if M contains a free variable and there is a lambda abstraction in E which uses the same name then the lambda abstraction variable must be replaced with a new variable name which does not occur free in E or M before M is substituted for x in E. i.e. $(\lambda y.E)_{[M/x]} = \lambda z.E_{[z/y][M/x]}$

$$(\lambda(\mathtt{x} : \mathtt{xs}).\mathtt{xs})\mathtt{1} : \mathtt{2} : \mathtt{3} : [\,] \quad = \quad \mathtt{2} : \mathtt{3} : [\,]$$

Finally, built in functions such as '*' are allowed in lambda calculus expressions. The names (or symbols) used for these functions are the same as for the equivalent function or operator in Miramod.

6.2 Organisation of the Compilation Process

The compilation of Miramod to the BMTP notation involves the compilation of property statements, function definitions and type descriptions. The main complexity is in the translation of the property statements and functions which is described in the following sections. The compilation of type descriptions is detailed separately in section 6.9.

The major features that must be removed during the compilation of property statements and definitions are:

- pattern matching definitions, such as

```
and False x = False
and True x = x
```

- guarded definitions

```
max a b = a, a>b
        = b, otherwise
```

- conformal definitions

```
(fst,snd) = ...
```

- local definitions ('`where x = ..`').

- list comprehensions.

```
primes = [n | n<-[1..]; is_prime n]
```

- notation specific to the built in types (eg the operators)

As well as this the model must be checked for type consistency.

The compilation process is organised in the following manner. Firstly the model is compiled to extended lambda calculus which includes pattern matching lambda calculus and local definitions. This involves converting conformal definitions, and guarded definitions to lambda calculus form. Pattern matching definitions can simply be transformed to the corresponding pattern matching lambda abstractions. List comprehensions and the notation for built in types are converted to lambda calculus. The pattern matching lambda abstractions are then compiled to plain lambda calculus using BMTP `IF` expressions. Lambda lifting is then performed to convert the definitions to supercombinators of the form '`f` p_1 `..` p_n `= body`' which are equivalent to the definitions permitted by the property prover. At the same time as the lambda lifting, dependency analysis is performed so that the information about the recursive nature of functions is available to the property prover and local definitions are floated to the top level (where possible).

Finally the resulting functions and property statements are checked for type consistency.

6.3 Translation to Enriched Lambda Calculus

The translation to enriched lambda calculus involves two translation schemes, one for functions and the other for expressions. The notation $\mathrm{TD}[\![d]\!]$ is used to describe the effect of translating a definition d to enriched lambda calculus and the notation $\mathrm{TE}[\![e]\!]$ is used for the translation of expressions. The TD scheme is applied to global definitions and the definitions local to property statements, whereas the TE scheme is used to translate property statements and is also used by the TD scheme.

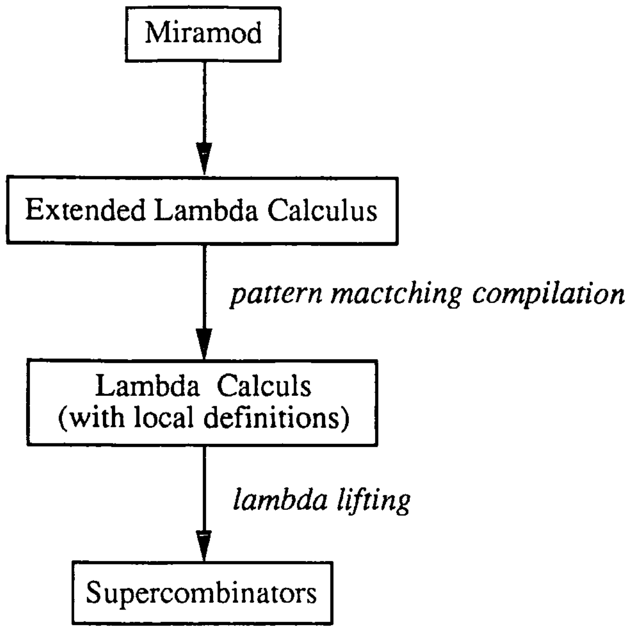

Figure 6.2: Miramod Compilation Overview

6.3.1 Expression Translation

The expression translation scheme is for the most part simple. Much of it concerns the translation of the symbols of Miramod into the corresponding BMTP notation. For example:

$$\begin{aligned} \mathrm{TE}[\![\, \mathtt{a+b} \,]\!] &= + \ \mathrm{TE}[\![\, \mathtt{a} \,]\!] \ \mathrm{TE}[\![\, \mathtt{b} \,]\!] \\ \mathrm{TE}[\![\, \mathtt{(a,b)} \,]\!] &= \mathtt{PAIR} \ \mathrm{TE}[\![\, \mathtt{a} \,]\!] \ \mathrm{TE}[\![\, \mathtt{b} \,]\!] \\ \mathrm{TE}[\![\, \mathtt{(a,b,c)} \,]\!] &= \mathtt{TRIPLE} \ \mathrm{TE}[\![\, \mathtt{a} \,]\!] \ \mathrm{TE}[\![\, \mathtt{b} \,]\!] \ \mathrm{TE}[\![\, \mathtt{c} \,]\!] \end{aligned}$$

Rather than introduce a new name for every Miramod operator or function, many are simply left unchanged and are distinguished entirely by context. Common operators such as + are therefore simply translated from infix to prefix notation.

The translation of list comprehensions (ZF expressions) is more complex and is therefore described separately in section 6.8.

6.3.2 Definition Translation

The translation scheme for definitions must take into account both the pattern matching and the guarded equation styles of definition. Given a pattern matching definition of the form:

$$
\begin{aligned}
&\texttt{f}\ p_{11}\ \ldots p_{1n}\ =\ \texttt{rhs}_1\\
&\texttt{f}\ p_{21}\ \ldots p_{2n}\ =\ \texttt{rhs}_2\\
&\ldots\\
&\texttt{f}\ p_{m1}\ \ldots p_{mn}\ =\ \texttt{rhs}_m
\end{aligned}
$$

the compiler produces the following definition:

$$
\begin{aligned}
\texttt{f}\ =\ \lambda v_1\ \ldots \lambda v_n.\ & f_1\ v_1\ \ldots v_n\\
\llbracket\ \ & f_2\ v_1\ \ldots v_n\\
\ldots\ \ & \\
\llbracket\ \ & f_m\ v_1\ \ldots v_n\\
\llbracket\ \ & \texttt{undef}
\end{aligned}
$$

the functions $f_1 \ldots f_n$ are the compiled versions of each of the original pattern matching equations. Since each of these must be applied to the functions arguments, a lambda abstraction with one new variable for every parameter of the function is created ($\lambda v_1 \ldots \lambda v_n$) and each pattern matching function is applied to the new variables. These function applications are joined using the *fatbar* operator (written $\llbracket$) which returns its first argument if the pattern match does not fail and returns the second argument if it does:

```
a ‖ b = IF (a == Fail) b a
```

The value `Fail` is a special value returned when a pattern matching lambda abstraction (contained in the functions $f_1 \ldots f_n$) fails to match its argument. In this way, the value of the function becomes the value of the first pattern match that does not return `Fail`. The final **undef** is the value returned if none of the patterns match. For many definitions the patterns are exhaustive and the compiler detects this and removes the final ▯ **undef**. For example:

```
and False x = False
and True x = x
```

Is compiled to:

```
and = λuλv.and1 u v
        ▯  and2 u v
```

where **and1** and **and2** are the compiled versions of the two pattern matching definitions.

Each pattern matching definition is translated to a pattern matching lambda abstraction, with one abstraction for each argument. The two pattern matching definitions of the function **and** above are compiled to λ`False`$\lambda x.$`False` and λ`True`$\lambda x.x$ respectively, giving the lambda calculus definition of **and** as:

```
and = λuλv.(λFalse λx.False) u v
        ▯  (λ True λx.x) u v
        ▯  undef
```

The translation scheme TD for a sequence of pattern matching definitions is therefore:

$$\mathrm{TD}[\![\ \begin{array}{lcl} f\ p_{1,1}..p_{1,n} & = & rhs_1 \\ f\ p_{2,1}..p_{2,n} & = & rhs_2 \\ \quad\dots & & \\ f\ p_{m,1}..p_{m,n} & = & rhs_m\,]\!] \end{array} = \begin{array}{ll} \multicolumn{2}{l}{\lambda v_1..\lambda v_n.} \\ & (\lambda\,\mathrm{TE}[\![\,p_{1,1}\,]\!]..\lambda\,\mathrm{TE}[\![\,p_{1,n}\,]\!]\,\mathrm{TR}[\![\,rhs_1\,]\!])v_1..v_n \\ & \dots \\ \talloblong & (\lambda\,\mathrm{TE}[\![\,p_{m,1}\,]\!]..\lambda\,\mathrm{TE}[\![\,p_{m,n}\,]\!]\,\mathrm{TR}[\![\,rhs_m\,]\!])v_1..v_n \\ \talloblong & \mathbf{undef} \end{array}$$

This scheme uses the translation TR for the right hand sides of definitions. In the simple case this is equivalent to TE but in the general case a series of expressions and guards are allowed as well as an optional set of **where** definitions. To compile the definitions containing guards, a built in function '`cond`' is used. It can be defined in terms of the BMTP `IF` as follows:

```
cond c a b = IF (c==True) a
                IF (c==False) b
                   undef
```

The distinction between `cond` and `IF` is important. The condition of `cond` is a Miramod boolean value which may be `True`, `False` or `undef`, but the condition of the property prover `IF` is a property value and may only be `T` or `F`.

Using `cond` the right hand side of a definition containing a sequence of guards is compiled in the following manner:

$$\mathrm{TR}[\![\ \begin{array}{ll} & e1, g1 \\ = & e2, g2 \\ = & .. \\ = & en, gn\ ]\!] \end{array} = \begin{array}{l} \texttt{cond}\ \ \mathrm{TE}[\![g1]\!]\,\mathrm{TE}[\![e1]\!] \\ \quad (\texttt{cond}\ \ \mathrm{TE}[\![g2]\!]\,\mathrm{TE}[\![e2]\!] \\ \qquad \ldots \\ \qquad (\texttt{cond}\ \ \mathrm{TE}[\![gn]\!]\,\mathrm{TE}[\![en]\!] \\ \qquad\quad \texttt{Fail})\ldots) \end{array}$$

The final Fail value is returned if all the guards return false, thus ensuring that the next pattern matching definition will be applied.

If the final guard is the reserved word 'otherwise' then the final `cond` is simply replaced with the last expression. So

$$\mathrm{TR}[\![\ \begin{array}{lll} & \ldots & \\ = & e_{n-1}, & g_{n-1} \\ = & e_n, & \texttt{otherwise}\ ]\!] \end{array} = \begin{array}{ll} \ldots & \\ \texttt{cond} & \mathrm{TE}[\![g_{n-1}]\!] \\ & \mathrm{TE}[\![e_{n-1}]\!] \\ & \mathrm{TE}[\![e_n]\!] \end{array}$$

Finally, if a definition contains some local definitions using the `where` notation, the local definitions are themselves translated and the resulting definitions are used to form a letrec expression:

$$\mathrm{TR}[\![\, rhs \quad \texttt{where} \; d1 \; d2 \; \ldots \; dn \,]\!] = \begin{array}{ll} \mathit{letrec} & \\ & \mathrm{TD}[\![\, d1 \,]\!] \\ & \mathrm{TD}[\![\, d2 \,]\!] \\ & \ldots \\ & \mathrm{TD}[\![\, dn \,]\!] \\ \mathit{in} & \mathrm{TR}[\![\, rhs \,]\!] \end{array}$$

6.4 Pattern Matching Compilation

The next phase is the removal of pattern matching lambda abstractions through a series of transformations.

At this point the Miramod compiler departs from the conventional approach of functional language compilation. The reason for this is that the objectives of the Miramod compiler are different from those of conventional functional language compilers. The Miramod compiler must produce function definitions that are suitable for proving theorems and are made up from `IF` expressions, recogniser functions and selector functions. On the other hand, conventional functional language compilers usually produce a more efficient form[1] by replacing repeated tests for a complete pattern with case statements which deal with all the possible values of one shell immediately.

The translation scheme for pattern matching lambda abstractions is named TP and as well as having a lambda expression as a parameter it is given a list of the pattern matching variables that have already been bound and a list of the variables the pattern is applied to. The notation $\mathrm{TP}(\mathit{bounds})(\mathit{args})[\![\, \mathit{exp} \,]\!]$ is used for the result of translating the pattern matching lambda calculus expression *exp*, applied to parameters *args* and enclosed by the pattern matching variables *bounds*.

Patterns are made up from:

- Constants (eg `True`, `'a'`, `10.5`)
- Variables — which may be repeated within the pattern
- Shells which have patterns in their argument positions[2].

The simplest case is a constant pattern, for example

```
(λFalse.True) x
```

which is translated to the expression:

```
ifdef x (IF (x==False) True Fail)
```

The function `ifdef` ensures that if the parameter is undefined then the result of the pattern match is also undefined:

```
ifdef v e = IF (v==Undef) Undef e
```

The compiled version of the pattern matching lambda expression will therefore return `Undef` if its argument is `Undef`, `True` if its argument is `False` and `Fail` otherwise. In the general case this translation is performed by the following rule:

$$\begin{aligned}\mathrm{TP}(bs)(a : args)[\![\,\lambda k.E\,]\!] = \texttt{ifdef a } (\mathrm{IF}(a == k) \quad & \mathrm{TP}(bs)(args)[\![\,E\,]\!] \\ & \texttt{Fail})\end{aligned}$$

The translation of a variable pattern is more interesting. If the variable is not in the set of variables already bound by the pattern match, then the lambda abstraction is left unaltered but the abstraction body (`E`) is translated with a bound variable set which includes the abstraction variable.

[2]There are in fact two types of shell, sum shells and product shells, however only the details of sum shell translation are given here.

$$\mathrm{TP}(bs)(a : args)[\![\lambda v.E]\!] \;=\; \lambda v.\,\mathrm{TP}(bs \cup \{v\})(args)[\![E]\!]\; a \quad \text{if } v \notin bs$$

If the variable is already in the bound set then a check must be made to see if the value given to the variable in this part of the pattern is the same as the value it has been given previously. For example, a definition of the form '`f y y = body`' will generate the lambda calculus expression '`(λyλy.body) u v`'. Strictly speaking this expression is incorrect, since the two abstractions over the variable `y` are intended to refer to the same variable, however the previously introduced semantics of the lambda calculus mean that the outermost lambda abstraction could be renamed to any variable that does not occur free in the body (alpha conversion). The pattern matching translation takes this into account by keeping a record of the variables that have already been encountered in the current pattern (the *bounds*), and generating the appropriate code if a repeated variable is found. For the example above, the code generated is '`(λy. cond (u=y) body Fail) v`' where `cond` is the conditional function previously introduced. The use of weak equality is important since it ensures that the pattern match behaves correctly for partial values. In the general case, the translation performed for a lambda abstraction over a variable that has already appeared in the pattern is as follows:

$$\begin{array}{lcll}\mathrm{TP}(bs)(a : args)[\![\lambda v.E]\!] & = & \texttt{cond}\ (a = v) & \mathrm{TP}(bs)(args)[\![E]\!] \\ & & & \texttt{Fail} \qquad\qquad \text{if } v \in bs\end{array}$$

The next translation rule deals with shell patterns. Taking the expression

```
(λ(a:x).body) u
```

as an example, the translated code should have the form:

```
ifdef u (IF (REC-: u) body' Fail)
```

where `body'` is `body` with `(SEL-:-1 u)` substituted for free occurrences of `a` and `(SEL-:-2 u)` substituted for free occurrences of `x`. This ensures that if the argument is undefined then the value `Undef` is returned and if the argument is a list construction then `body'` is returned. Otherwise the pattern match fails, returning the value `Fail`.

The general case is more complex, since the pattern arguments may not be simple variables. To cater for this, the translation algorithm generates the appropriate conditions and a `body'` that has the argument patterns as a sequence of pattern matching lambda abstractions applied to the components of original lambda abstraction argument selected using the selector functions:

```
(λ(C p₁ ...pₙ).body) u
```

gives

```
body' = (λp₁...λpₙ.body) (SEL-C-1 u) ...(SEL-C-n u)
```

The new body then has its pattern matching lambda abstractions removed by recursively applying the translation to it. The translation scheme for shell patterns is therefore:

```
TP$(bs)(a : args)[\![ \lambda(c\ p_1\ ..\ p_n).E ]\!]$ =
   ifdef $a$ IF  (REC-c $a$)
                 TP$(bs)($(SEL-c-1 $a$) : ...(SEL-c-n $a$) : $args)[\![ \lambda p_1 \ldots \lambda p_n.E ]\!]$
                 Fail
```

The final translation rule ensures that the recursion terminates when the expression being translated is not a lambda abstraction:

$$\mathrm{TP}(bs)(args)[\![E]\!] = E\ args \text{ if } E \text{ is not a lambda abstraction.}$$

Rather than translating pattern matching lambda abstractions in an entirely separate phase from the translation from Miramod to lambda calculus, the TD is translation schemes calls on the TP translation scheme to compile pattern matching lambda abstractions as they are produced. Hence the TD translation scheme is actually:

$$
\begin{aligned}
\mathrm{TD}[\![\quad & f\ p_{1,1}..p_{1,n} && = rhs_1 \\
& f\ p_{2,1}..p_{2,n} && = rhs_2 \\
& \quad \ldots \\
& f\ p_{m,1}..p_{m,n} && = rhs_m\ ]\!]
\end{aligned}
$$

$$=$$

$$
\begin{array}{l}
\lambda v_1..\lambda v_n. \\
\quad \mathrm{TP}(\{\})(v_1..v_n)[\![\ \lambda\,\mathrm{TE}[\![p_{1,1}]\!]..\lambda\,\mathrm{TE}[\![p_{1,n}]\!]\,\mathrm{TR}[\![rhs_1]\!]\,]\!] \\
\quad \ldots \\
[\!] \ \ \mathrm{TP}(\{\})(v_1..v_n)[\![\ \lambda\,\mathrm{TE}[\![p_{m,1}]\!]..\lambda\,\mathrm{TE}[\![p_{m,n}]\!]\,\mathrm{TR}[\![rhs_m]\!]\,]\!] \\
[\!] \ \ \texttt{Fail}
\end{array}
$$

In this way, the overall translation ensures that the repeated variables in lambda abstractions are interpreted as pattern matching lambda abstractions and replaced with the correct equality conditions only when they are indeed in the position of pattern matching lambda abstractions.

6.5 Compilation to Supercombinators

The result of the preceding compilation phases is a set of property expressions and function definitions in enriched lambda calculus notation. The main objective of this final phase is to replace the lambda abstractions with the simple function definition notation used by the property prover. Many of the lambda calculus definitions can be translated to the property prover notation very simply. For example the function

```
max a b = a, a>b
        = b, otherwise
```

would be compiled to:

```
max = λaλb.cond (> a b) a b
```

which can be converted to the property prover notation simply by removing the lambda abstraction from the right hand side and adding their variables to the left hand side of the definition:

```
max a b = cond (> a b) a b
```

In many cases (particularly compiled list comprehensions) the lambda abstractions are not all at the top level. If this is the case, a new function name can be created and defined as the inner lambda abstraction. The lambda abstraction can then be replaced by this new function. For example the inner lambda abstraction $\lambda a.E$ in

$$f = \lambda x ... (\lambda a.E) ...$$

can be replaced by $f1$ and the definition of $f1$ as $\lambda a.E$ added:

$$\begin{aligned} f &= \lambda x ... f1 ... \\ f1 &= \lambda a.E \end{aligned}$$

These definitions can then be written to the correct form for the property prover by removing the lambda abstractions and placing the abstraction variables on the left hand side of the definition as formal parameters:

```
f x = ... f1 ...
f1 a = E
```

Unfortunately this technique does not work if the body of the lambda abstraction contains any free variables, since moving the body of the abstraction to another definition may remove these variables from the scope in which they were defined. For example:

$$f = \lambda x.g\ (\lambda y.h\ x\ y)$$

In this case the inner lambda abstraction $(\lambda y.h\ x\ y)$ contains a free occurrence of the variable x which prevents it from being extracted as a separate function. Ignoring the free occurrence of x would give the following definitions:

```
f x = g $y
$y y = h x y
```

which is incorrect because in these definitions the two occurrences of **x** are unrelated.

There is one circumstance in which a free variable is acceptable, and this is when the free variable is bound in one of the immediately enclosing lambda abstractions. For example x occurs free in the body of the λy abstraction in

$$\lambda x \lambda y.(\ldots x \ldots y \ldots)$$

This is acceptable, because the λx immediately encloses the λy and the corresponding property prover function can be generated as:

```
f x y = ...x ...y ...
```

The class of lambda calculus expressions that can easily be converted to the property prover notation are known as the *supercombinators.*

A supercombinator is a lambda calculus expression of the form

$$\lambda a_1.\lambda a_2 \ldots \lambda a_n.E$$

where $n >= 0$ and E is not a lambda abstraction, contains no free variables other than a_1 to a_n and any lambda abstractions contained in E are themselves supercombinators.

Any supercombinator can be transformed to the property prover notation simply by replacing any occurrences of the supercombinator with a unique function name and adding a function definition with the supercombinator variables as formal parameters and the supercombinator body with any supercombinators removed as the function definition body.

The Miramod compiler's strategy is therefore to transform all expressions to supercombinator form and then convert these definitions to the property prover notation. The process of converting lambda abstractions to supercombinators is known as 'lambda lifting' and is a standard technique in the compilation of functional languages.

6.5.1 Lambda Lifting

If a lambda abstraction contains a free variable then the free variable can be bound simply by adding a new lambda abstraction for the free variable and applying this to the free variable itself. This process is sometimes referred to as 'abstracting a free variable'.

$$\lambda y.x\ y$$

is transformed (using the reverse of the beta reduction rule for lambda calculus) to:

$$(\lambda x \lambda y.x\ y)\ x$$

The lambda abstraction is now a supercombinator.

The lambda lifting algorithm is as follows:

1. descend into the constituent parts of the expression, recording as free any variables found.

2. for each lambda abstraction:
    (a) lambda lift the abstraction body returning a list of the free variables found.
    (b) remove the abstraction variables from this list and call the result 'Frees'
    (c) give a unique name to the lambda abstraction
    (d) replace the occurrence of lambda abstraction with the new name applied to 'Frees'
    (e) add the new name as a function definition with the original lambda abstractions and 'Frees' as parameters and the definition body as the lambda lifted abstraction body

Taking the right hand side of the following function definition as an example:

$$f = \lambda x.g\ (\lambda y.h\ y\ x)$$

This expression itself a lambda abstraction, so the first step is to lambda lift the abstraction body

$$g\ (\lambda y.h\ y\ x)$$

The function 'g' is defined globally, so it is not recorded as a free variable. The remaining part of the expression is another lambda abstraction, so its body is compiled. In this case there are two free variables x and y so these are added to the free list. Since it contains no lambda abstractions, no change need be made to the λy body and the free list containing x and y is returned. Since y is the λy abstraction variable it is removed from the free list giving 'Frees' as the list containing just x. A unique name is now given to the lambda abstraction — say $\$y$ and the λy abstraction is replaced with $\$y$ applied to the free variable x:

$$f = \lambda x.g\ (\$y\ x)$$

$y is then added as a new function definition with the variables x and y as formal parameters and the lambda lifted abstraction body as the function definition body:

$$\$y\ x\ y = h\ y\ x$$

Since both g and $y are defined globally, the λx abstraction is already a supercombinator and is therefore converted without the addition of any parameters:

$$\begin{aligned} f &= \$f \\ \$f\ x &= g\ (\$y\ x) \end{aligned}$$

As this example demonstrates, the lambda lifting process has a tendency to create trivial and unnecessary supercombinators. These can be eliminated by removing definitions of the form f=$f, renaming the supercombinator $f as f in the rest of the model. Another simplification is to perform η-reduction on definitions of the form:

$$f...w\ x = E\ x$$

where x does not occur free in E

Applying η-reduction removes the extra parameter giving:

$$f...w = E$$

If all of the parameters can be removed using η-reduction then the definition can be eliminated provided the remaining right hand side of the definition is simply a supercombinator name.

6.6 Letrec Compilation

Although local definitions are not part of the BMTP theory, they are permitted in the property prover provided they only occur immediately in the body of a function definition. In other words a function of the form:

$$\begin{aligned} f\ x_1..x_n &= \textit{letrec} \\ &\quad \dots \\ &\quad \textit{in}\ E \end{aligned}$$

is acceptable to the property prover because the letrec is at the top level of the definition body. On the other hand, definitions of the form

$$\begin{aligned} f\ x_1..x_n &= g\ \ \textit{letrec} \\ &\qquad \dots \\ &\qquad \textit{in}\ E \end{aligned}$$

are not acceptable because the letrec is not at top level. The reason for this is that it means only the heuristics for opening up definitions need cater for the possibility of local definitions. When a function definition containing a letrec is opened up, the function's actual parameters are substituted for the formal parameters in the letrec definitions as well as the letrec body. The variables defined by the letrec are assigned unique names and added to the global function definitions, and the letrec body is returned as the value of the function. Since the letrecs must be at top level in the function body, there is no need for the property prover to search for enclosed letrecs before opening up the definition — it can tell immediately if a letrec is present or not.

6.7 Type Checking

An important phase of the compilation process is the check for type consistency within the model. Since the type checking algorithm requires dependency information, it is performed after the lambda lifting and dependency analysis phase. There is a strong case for checking types at an early stage of the translation process since any errors found can then be reported with reference to the source code in which they occur. Rather than having to perform dependency analysis twice, the Miramod compiler records information about the position of definitions and property statements in the source code and reports the error in terms of the property or function name and line number.

The definitions of the model are checked first since this process not only checks for consistency but also establishes the types of the functions if their type is not explicitly declared. The order in which definitions are checked is from the function that is least dependent on other functions in the model to the function that is most dependent on the other functions in the model. This means that unless a definition is mutually recursive the type checking algorithm can depend on the existence of information on the type of every function occurring within the definition.

The type checking algorithm uses two tables which give the types associated with particular identifiers. One table, called the fixed table, contains the type information for identifiers whose type is already known, the other contains information about the current type of identifiers for which a type is being inferred and is called the unfixed table. The type checking algorithm also has a type which it expects to be the type of the current expression.

There are two main cases that must be dealt with by the type checking algorithm, one is when the current expression is a function application and the other is when the expression is an identifier. In the latter case, the unfixed table is checked initially, and if it contains a type for the identifier the expected type and the unfixed types are unified; that is they are checked to see if there is a substitution for type variables in each of the type expressions that make the two expressions equivalent. If this is the case then the substitution is applied to both the unfixed table and the currently expected type. If the expected type and the unfixed type will not unify then this constitutes an inconsistency in the type structure of the model and a type error is reported.

If the identifier is not listed in the unfixed table then its type is obtained from the fixed table and unified with the expected type. The resulting substitutions are only applied to the unfixed table and the currently expected type — not the fixed table. This is because the fixed table contains the types of defined functions whose definitions have already been checked and the current use of the function can be a specialisation of the type inferred for the function from its definition.

The other main case in type checking is when the expression being checked is a function application, for example '`f x`'. In this case a new type variable $\mathtt{T}_x$ is obtained and the type checking algorithm is recursively invoked for `f` with expected type '$\mathtt{T}_x$ `->` $\mathtt{T}_e$' and for `x` with expected type $\mathtt{T}_x$ where $\mathtt{T}_e$ is the expected type of the expression '`f x`'.

6.8 List Comprehensions

The translation of list comprehensions relies on a built in function **`flatmap`** which is defined as follows:

```
flatmap f [] = []
flatmap f (x:xs) = f x ++ flatmap f xs
```

Thus `flatmap` takes a function and a list as parameters and applies the function to each element of the list, concatenating the resulting lists. Note that the function `f` must return a list.

Given a list comprehension of the form

```
   [E|v<-l]
```

where `E` is an expression (possibly containing `v`), `v` is a variable and `l` is a list then the corresponding lambda calculus equivalent is

```
flatmap (λv.[E]) l
```

In the general case the list comprehension is made up from an arbitrary number of qualifiers on the right of '`|`'. So the translation scheme is:

$$\mathrm{TE}[\![\,\texttt{[E|v<-l; Q]}\,]\!] = \texttt{flatmap}\ (\lambda v.\,\mathrm{TE}[\![\,\texttt{[E|Q]}\,]\!])\,\mathrm{TE}[\![\,\texttt{l}\,]\!]$$

The recursive translation ends when there are no further qualifiers left:

$$\mathrm{TE}[\![\,\texttt{[E|]}\,]\!] = \texttt{[E]}$$

Qualifiers may also be boolean expressions, in which case the following translation applies:

$$\mathrm{TE}[\![\,\texttt{[E|B;Q]}\,]\!] = \texttt{if}\ \mathrm{TE}[\![\,\texttt{B}\,]\!]\ \mathrm{TE}[\![\,\texttt{[E|Q]}\,]\!]\ \texttt{[]}$$

where `if` is defined as:

```
if c a b = a, c
         = b, otherwise
```

This translation means that if the boolean expression evaluates to true then the list comprehension made up from the expression `E` and the remainder of the qualifiers `Q` is returned, otherwise the empty list is returned. If the boolean expression is preceded by a generator then the empty list will disappear due to the application of flatmap. It is important to note that none of the variables introduced by `Q` (if there are any) can occur in `B` since the scope of variables introduced as qualifiers extends to the qualifiers on the right but not those on the left.

There are a number of other forms of qualifiers whose translation is not listed here. Details of these can be found in the book by S.L.Peyton Jones [35].

6.9 Type Compilation

As well as compiling property statements and function definitions, the Miramod compiler must convert the type descriptions of a model to a corresponding form in the property prover notation so that types as well as functions can be checked for similarity. Miramod provides three mechanisms for describing types: algebraic type descriptions, algebraic type shorthand and representation based types (the built in types are automatically made available to the property prover).

The compilation of Miramod algebraic type descriptions is trivial because the constructors and destructors of the type are simply treated as functions without definitions by the property prover. All that is required is for the property statements associated with the type to be compiled in the same way as other property statements are compiled.

The compilation of algebraic type shorthand and representation based type descriptions is more complex and these are described in the following sections.

6.9.1 Algebraic Type Shorthand

A shorthand algebraic type description is written:

t ::= C_1 ... | C_2 ... | ... | C_n ...

where t is the type name and C_i are the shell names which are followed by the types of their arguments. These shells have a special status, since they can be written on the left hand side of pattern matching function definitions. The compiled version of such a definition assumes that each shell has a recogniser function and that each argument of a shell has a selector function. It is therefore important that the appropriate function definitions for the recogniser functions and selector functions are generated for the algebraic type.

Given a shell C, the recogniser function is named REC-C and the selector functions are named SEL-C-n where n is the number of the argument selected by the function. These functions are defined as follows:

```
REC-C (C p1 .. pn) => T
REC-C undef        => F
S~==C |- REC-C (S p1 .. pn) => F

REC-C x |- (C (sel-C-1 x) (sel-C-2 x) ... (sel-C-n x)) = x
```

and for all n where 1≤n≤m

```
SEL-C-n (C p1 .. pn .. pm) => pn
```

The recogniser functions return a property (T or F) rather than a Miramod boolean value, hence they return F if their argument is undefined and they can never return the undefined value. The semantics of the selection functions applied to the wrong argument are not defined because the compiler ensures that any use of a selector function only occurs after its argument has been checked with a recogniser function.

6.9.2 Representation Based Type Descriptions

When a representation based type description occurs in a re-user's model, the properties describing the type will contain functions and values of the representation type as well as the described type. To prove these properties in terms of a component model it is necessary to have some means of mapping the re-user's model properties into a form in which they can be proved from the component model. For algebraic type descriptions this is achieved through the view of the described functions, however for the properties that include values and operators of the representation type, there must also be a method of mapping representation values and functions into component model values and functions. Taking the example of a stack type description from chapter 4 (page 76), the properties associated with the description are:

```
{e s}
    empty = []                  ;;
    push e s = e:s              ;;
    pop (e:s) = s               ;;
    top (e:s) = e               ;;

    is_empty [] = True          ;;
    is_empty (e:s) = False      ;;
```

Rather than leaving it to the property prover to try and invent a view from the representation to the component model as well as the view from the function of the described type to the component model, the compiler insists that the re-user supplies an abstraction function with each representation based type description. This abstraction function can then be used to convert all the properties to a form in which a view of only the signature functions is needed to prove the properties. In the above example, the retrieval function given with the model is:

```
{e s}
    ret_stack [] = empty         ;;
    ret_stack (e:s) = push e (ret_stack s)  ;;
```

Using `ret_stack`, the property of pop in the above example can be converted to:

```
pop (ret_stack (e:s)) = ret_stack s
```

and hence proved from properties such as '`pop (push k l) = l`'.

The conversion of properties of representation described types is complicated by the fact that not every value in the representation need correspond to a value of the type it describes. This situation occurs when the constructors of the type never produce some of the possible values in the representation. For example a set is often represented as an ordered list — in which case the constructors should never produce an out of order list[3]. To cater for this possibility a re-user is expected to supply a validity function as well as an abstraction function. The validity function gives a boolean result depending on whether or not a representation value has an equivalent value in the described type (in other words it is true for a value if that value can be created by the constructors of the described type). The converted properties are then given an additional hypothesis for each retrieval function application they contain. This hypothesis uses the validity function for the type to ensure that the property only need hold for values that can be created from the constructors of the described type. Given the validity function `valid_stack` the `pop` property from the example is therefore converted to:

```
valid_stack s & valid_stack (e:s)
 |- pop (ret_stack (e:s)) = ret_stack s
```

To perform this translation, the described and representation types are considered as distinct types, and the retrieval function is used as a *coercion* function at any points in the property where an expression which gives values of the representation type appears in a position where a value of the described type is expected. The resulting expression is correctly typed even when considering the representation and implementation types as distinct. A coercion function is one that converts from values of one data type to values of another, in this case between representation and implementation types[28].

Deciding where these coercions are needed in a given property expression is not entirely straightforward since the types of some of the functions or variables in the statement may not be explicitly mentioned. This results in a situation where a variable or function must have both the representation type and the implementation type at different points in the expression and it is therefore unclear what the type

[3]In the context of data type specifications this situation is referred to as implementation bias[32].

of the function or expression should be considered as when adding coercions. Since the only coercion available is from representation to described type, all expressions that are expected to be of both types are assumed to be of the representation type and are coerced to the described types in all the contexts where a described type is expected. Taking the property expression for **pop** as an example:

```
pop (e:s) = s
```

the variable s appears on the left hand side in the context of a list. On the right hand side its context is as a stack (since the opposite side of the equality returns a stack). The type of s is therefore taken as a list, and the occurrence of s on the right hand side is coerced to a stack. If the alternative choice of taking s as a stack is made, then the resulting property statement is badly typed in the component model and the property cannot be proved.

6.10 Summary

The aim of the Miramod compiler is to establish the correctness of a model with respect to the language syntax and type laws, and to translate the Miramod notation to a notation that is acceptable to the property prover. It achieves this using for the most part conventional functional language compilation techniques: translating Miramod first to enriched lambda calculus, then translating enriched lambda abstractions to normal lambda abstraction and finally performing lambda lifting to produce supercombinators.

The major difference between Miramod compilation and conventional compilation to supercombinators is in the translation of pattern matching, where the property prover translates pattern matching lambda abstractions to a form based on the property prover's IF expressions rather than to a form that can be efficiently executed.

Chapter 7

Implementation

This chapter describes the current state of the retrieval system's implementation. The main components of the system are the property prover, Miramod compiler, and model comparison algorithms.

In the remainder of this chapter, all timings given are in real time, and represent typical values when running the software as the only user of a Sun 3/50 workstation.

7.1 Miramod Compiler

The compiler has been written using the Unix tools 'lex' [34] and 'yacc'[31] as well as the logic programming language Prolog. It consists of a total of around 3,000 lines of source code. The initial lexical analysis and parsing is handled by lex and yacc, which produce output in the form of Prolog clauses. These clauses are then translated by Prolog, first to lambda calculus then into supercombinator form. Finally dependency analysis is performed to establish the recursion groups of recursive functions and prepare for the final phase, which is type checking.

The choice of Prolog to implement the main transformations performed by the compiler has been justified by the transparent nature of much of the code. The implementation of many of the translations corresponds closely to the transformation schemes detailed in chapter 6.

Compilation times are typically in the order of two or three seconds for small models (approximately 20 lines of code), but these deteriorate for large 'models' — the Miranda standard environment, which is 700 lines, takes four minutes to compile. This performance is acceptable since the models should always be small so that they can be matched by the property prover. During a particular retrieval, the compiler need only be used to compile the re-user's model, since the compiled versions of the component models are stored along with the source.

7.2 Property Prover

The property prover is implemented in Prolog and consists of approximately 5,000 lines of source code. The language Prolog was chosen as a compromise between the need to produce a prototype property prover as quickly as possible whilst at the same time making it reasonably efficient.

The obvious alternative of using Miranda to implement the property prover was rejected because of the current version's inability to perform list indexing operations in constant time. Indexing operations are fundamental to the efficiency of the property prover because they form the bases of lookup tables for many crucial items such as function definitions, constructor type information, rewrite rules generalisation and elimination axioms, and induction templates.

Prolog is itself a crude theorem prover based on the resolution of a special class of clauses called Horn clauses. It is frequently criticised for its depth first search strategy, but in the property prover implementation this is generally used as the underlying mechanism for executing the property provers searching heuristics. Prolog's search strategy is used for the synthesis of views (instantiating free functions), however in this case the backtracking behaviour is tightly constrained by the implementation. Another criticism of Prolog is its lack of an 'occurs check' in its unification algorithm. This means that goals such as '`X = cons(X,Y)`' succeed despite the fact that `X` properly occurs in `cons(X,Y)`. For the vast majority of unifications in the theorem prover this is not a problem, but in the few cases where a variable may potentially occur within the term with which it is being unified an additional predicate which performs an occurs check is used rather than the conventional unification algorithm. The fact that this additional unification predicate can be written in a few lines of Prolog is an indication of the expressive power of Prolog.

The property prover makes maximum use of the hashed lookup provided by Prolog. In particular the version of Prolog used implements hashing on the first parameter of a predicate as well as the predicate name. This means that if there are many facts associated with a particular predicate, for example the predicate used to store function definitions, each can be accessed in constant time provided the top level function symbol (functor in Prolog terminology) is different for each fact (and the hash table is not full). To take maximum advantage of this, the representations of facts such as function definitions are arranged so that the first parameter of the fact will have a different symbol wherever possible, and any attempts to look up such facts can specify a constant symbol (as opposed to a variable) in the first argument position. In the case of function definitions, the first argument is always the function name. An alternative representation which used the first argument to store a list of the function name and formal parameters would produce a linear lookup time because each definition would have the list construction function symbol as its first top level function symbol, and therefore the hash value for all definitions would be the same. The same technique is used to implement constant order lookup for constructor type set information, the current terms in the conjecture, rewrite rules, elimination and generalisation axioms and induction templates.

7.3 Model Comparison

The model comparison algorithm attempts to pair each type described in the reuser's model against a type in the component model. To do this it uses the type signature as guidance. Having selected a mapping of the re-user's model types onto component model types, a view from the re-user's model functions to the component model functions is established. Any functions for which a view cannot be established are marked as free functions.

The property which contains the fewest free functions is then selected and given to the theorem prover which returns two results, one for the proof of the property when ignoring partial and undefined values and the other for the proof that takes these into account. Both results are then scored and the scores summed to give the matching score for that particular property.

A score for each property is established in this way, properties with the fewest free functions first, and the resulting scores are summed and divided by the number of property expressions to give the overall score for the match between components.

The proofs are performed in this order to try and ensure that the best possible view between models is the one synthesised. The best view is likely to be achieved by ensuring that for each property proved, the number of free functions for which a view is synthesised during the proof of that property is as small as possible. The ideal case is achieved when each property with more than one free variable synthesises the view for only one of its free variables. During the proof of subsequent properties, the variable is no longer free.

Both parts of the result returned by the property prover have one of three values: 'T' if the property was proved, 'F' if the attempted proof resulted a clause of the form {F} being generated, and finally '?' if the attempted proof failed to produce any answer (ie. timed out or produced a clause for which no induction could be constructed).

These values are scored by the following table which attempts to reflect the relative importance of total and partial properties as well as the relative importance of 'T', 'F' or '?' in determining the similarity of models. The current values are based on the limited experience gained in matching models from the Miranda library.

	T	?	F
Total properties	80	20	0
Partial properties	20	5	0

So for example, if a model has only one property, and this is proved for total values but is disproved for partial values, then the matching score returned is 80.

7.3.1 Efficiency

The efficiency of the matching algorithm is crucial to the feasibility of model based component library retrieval. Unfortunately neither the maximum acceptable matching time or the time taken to match components is constant.

The property prover's performance is highly dependent on the relationship between the property being proved and the properties of the library model, in particular the heuristics supplied with the components model. For a property that just requires simplification, the property prover typically takes less than a second to complete the proof. For example, the proof of all three compiler properties given in chapter 4 requires only two seconds since none of them involve induction.

Proofs that require induction tend to take longer, and there is frequently a close relationship between the number of inductions required and the time taken to complete the proof. For example the proof of the property '`reverse (reverse x) = x`', assuming only the definition of `reverse` and `++` (which is used in the definition of `reverse`) and the induction templates for `reverse` and `++`, uses two inductions and requires 8.5 seconds to complete. This time would be shortened considerably if the library model contained more information on proving theorems about reverse, for example the rewrite axiom:

```
{y x+}
    reverse ( y ++ [x] ) => x:y
```

or even

```
{x} reverse (reverse x) => x
```

If a property is not provable from the component model then one of three situations occurs: The theorem prover may produces a clause in which each literal is irrelevant, and the result 'F' is assumed. This occurs when the property is not true for base cases and is common when attempting to prove properties of arbitrary models from the library. Alternatively, the property is simplified to a state where no more simplifications can take place and no appropriate induction schemes can be found. This usually takes place immediately (ie without any inductions occurring) or does not occur at all because inductions schemes can be applied ad infinitum without ever simplifying the resulting clauses to T. This case is relatively rare - if any recursive functions are contained in the simplified clause, there should be an appropriate induction template. The third alternative is that the attempted proof terminates because the time allocated for the proof expires (ie the proof times out).

The timeout value is essentially what determines the matching time for models where the required property is true in the base cases but not in the inductive ones. Since the majority of property proofs should fail (most of the components in the library will not be suitable unless an extremely general model is given), choosing a timeout value is crucial; if it is too long then the retrieval time will be unacceptable, if it too short then the retrieval completenes will be low because insufficient time was available for the proof of properties.

The current implementation indicates that a retrieval time of around ten seconds per property is the minimum required to allow for matching on the basis of anything other than trivially similar properties or type information alone. Given a re-user's model with three properties and a timeout value of ten seconds the time taken in matching one model is at most thirty seconds, allowing 120 comparisons per hour. This figure must be considered in relation to the percentage of properties which can be rejected immediately on the basis that no view between the models can be established.

In the retrieval experiments described in chapter 8 on average a view can be synthesised for only 1.6% of the models in the library. If this figure is correct for full scale component libraries, the property prover need only be applied to the

1.6% of the library models for which a view can be found. Since the initial view discovery (as opposed to synthesis) is based on the comparison of known types, it is extremely efficient relative to the property prover and therefore can be used as an initial filter for appropriate models.

Assuming this figure of 1.6%, a library of 10,000 components could be searched, on the basis of non trivial model comparisons, in at most three hours, and since the majority of models fail on base cases the actual time should be a great deal less. Whether or not these retrieval times are acceptable depends on the circumstances of the re-user as well as the computing resources available. If a large amount of work is saved by re-using a retrieved component, then long retrieval times may be acceptable. On the other hand, for smaller components a greater retrieval loss may be acceptable and therefore a faster retrieval time can be achieved by reducing the time allowed to prove each property.

Chapter 8

Experimental library and Retrieval System

To investigate the levels of precision and completeness that can be achieved through model based retrieval, an experimental library and retrieval system have been set up. This chapter describes the experimental library and retrieval system and the results obtained from their use.

8.1 Library

The experimental library consists of two main parts, the components themselves and the models used to describe and retrieve the components. These parts are described separately in the following sections.

8.1.1 Components

The experimental library consists of approximately 600 Miranda components (types and functions), each with one or more models. Coherent sets of types and functions are held in scripts (modules), so that only one '`%include`' statement is needed to bring the whole set into scope. The size of components varies greatly, and although some larger components do exist, the majority are small, consisting of only a few lines of code. Examples of the larger components are: a parser generator which provides a type for constructing parsers; components for manipulating abstract expressions including substitutions, unification, and term re-writing; and a type for writing interactive programs.

Examples of small components are functions such as '`perms`' which gives the permutations of a list and the type '`exception`' which provides for simple exception handling. In the latter case, it is interesting to note that although the basic component itself is straightforward, there are many useful functions associated with the type (currently 15).

The small size of many of these components means that the effort of retrieving them may be greater than the effort of rewriting them. Nevertheless, it is often still beneficial to retrieve rather than rebuild because this means that the same function (with the same name) will be used by all users of the library, and hence the code involving such reused functions will be more readily understood by a larger collection of people.

This is particularly important in the case of the standard environment functions. A script written using unfamiliar functions instead of the standard environment functions is a great deal harder to understand than one which uses the standard functions. For this reason the library also contains components which are part of the standard Miranda and Miramod environments. This allows users who are not completely familiar with these environments to retrieve and use the standard functions rather than building their own.

8.1.2 Models

The models for components are stored separately from the components themselves. Each model consists of the Miramod description along with the name of the component it models and the name of the script which contains the component. If a successful match is made, this information can then be used to locate the component itself.

Although models may consist purely of property statements and definitions, the majority contain additional information in the form of heuristics for the theorem prover. The syntax of Miramod is extended for library models, so that this information can be presented as part of the Miramod description of the components. Each piece of heuristic information is stored as a property expression, using the additional property expression notation described below.

Rewrite Rules are used to indicate that expressions of one form should always be replaced by expressions of another form (with appropriate substitutions made) provided that the hypothesis can be established. They are written:

```
H |- L => R
```

where H is the hypothesis expression, L is the expression to be replaced and R is the replacing expression. If no hypothesis is necessary then the rewrite rule may be written:

```
L => R
```

Induction Lemmas state that one expression (a) is smaller than another (b) according to the given measure (m), provided the hypothesis can be established.

```
H |- m a << m b
```

The hypothesis can be omitted from induction lemmas giving the following form:

```
 m a << m b
```

Elimination Lemmas suggest that a collection of destructor terms should be replaced by the corresponding constructor term.

```
H |- L == vj  (elim)
```

The expression L must contain at least one destructor term of the form `(d v1 ... vn)` where `v1 ... vn` are variables and one of them is in fact `vj`. Again the hypothesis may be omitted.

Since many elimination lemmas have corresponding rewrite rules, they may be written with `=>` appearing instead of `==`, in which case they are interpreted as both an elimination lemma and a rewrite rule.

Generalisation Lemmas are property expressions which should be added as a hypothesis to the conjecture being proved whenever one of their proper subterms is generalised.

```
A  (gen)
```

The symbols `(elim)` and `(gen)` can also serve as property expression separators, and so they do not need to be followed by `;;`.

These extensions do not apply to the models written by a reuser, since no attempt is made to prove anything from the reusers model.

8.2 Retrieval System

To retrieve a component the reuser must supply a model for the component written in Miramod. The retrieval system then selects candidate models from the library by comparing the types of the reusers model with the types of the library models. This comparison is similar to that performed during the initial view synthesis (page 97) and is based on the same definition of 'similar' types. Each possible mapping of users model types to library model types is scored on the basis of the number of users model functions which are similar to one or more functions from the library model and the highest of these is taken as the score for that particular library model. This process is performed for every model in the library, and the models which score 50% or more are then selected for comparison by the property prover.

8.3 Retrieval Experiments

The aim of these experiments is to establish the levels of precision and completeness that can be obtained using model based retrieval. This has been done by formulating retrieval requests for components that are known to be in the library and then analysing the results to obtain estimates of precision and completeness.

One of the major difficulties in obtaining these estimates is that both precision and completeness are based on the notion of the suitability of a component for a particular task. Should components which could be used for the required task with slight, minor or major modifications be judged as suitable? Also if a retrieved component could be combined with other components to achieve the desired result, is this component suitable?

Since the components in the experimental library are generally small, the effort of modifying them would often be greater than that of building a new component from scratch, so we only consider components that can be reused without modification as suitable. On the other hand, since one of the main advantages of functional components is that they can easily be combined, we consider as suitable any components which could perform the required task when combined with other relatively trivial functions.

For each retrieval request, we therefore use these guidelines to judge which of the retrieved components should have been retrieved and which should not have been retrieved, thus obtaining a measure of precision. We then use our knowledge of the components in the library to identify the number of suitable components that where not retrieved and thereby obtain a measure of completeness.

These experiments are based on several important assumptions. Firstly, we have assumed that we are interested in assessing the retrieval method itself, rather than the ability of the library to supply the components we require. As a result all retrieval requests are based on components actually contained in the library. Secondly, the performance of model based retrieval is critically dependent on the time allowed for proofs of properties and the adequacy of the heuristics contained in the library model. As our principal aim is to investigate the potential of the method, we have assumed that the necessary heuristics are present in the model and that the property prover is given sufficient time to make the proof.

The significance of the results obtained is limited by the fact that both the library models and the re-user's models are written by the same person, and so do not reflect the ability of a reuser to produce models similar to those in the library. One further limitation of the experiments is that all our retrieval requests are for functions rather than types. This is due to the fact that the current experimental library contains only a small number of types which are easily distinguished by their level of polymorphism and signatures.

8.4 Case Studies

Before examining the results obtained from many retrieval requests, a few specific cases are described. Some of these are included to demonstrate extremes of performance rather than average performance.

8.4.1 The function 'reverse'

Since various models for the reverse function have already been presented, the results of using these models on the component library are presented here. The first model states that reverse is its own inverse, and that it acts on polymorphic lists.

```
{x}  reverse (reverse x) == x;;
```

```
x::[*]
```

This model for reverse recalls two components, both reverse itself and the function `id x = x`, which is definitely not what we required. The precision is therefore 50%. In this case, the majority of components from the library are rejected because their types are not similar, and only ten components are checked by the property prover. If retrieval of `reverse` was performed on the basis of type alone, a precision of only 10% would have been achieved.

It is tempting to conclude that reverse is the only suitable function, since no other function has the same type and behaviour; however the function 'foldl', though not exactly what is needed, can be used to generate a reverse function by instantiating its first two parameters: `(foldl (converse(:)) [])`. Two factors prevent the retrieval of `foldl`. The first is the definition of similar types we employ — '`[*]->[*]`' is not similar to '`(*->**->*)->*->[**]->[*]`' and secondly the view synthesis heuristics are not powerful enough to discover the appropriate instantiations for `foldl`'s arguments[1].

Runciman and Toyn [55] describe a relation between types which accounts for the intuition that $t_1 -> t_2$ is a generalisation of t_2 because it can be used to create t_2 values. Unfortunately this cannot be used directly to extend the generalisation relation used here without creating a pre-order rather than a partial-order (consider the types of `apply` and `id` where '`apply f x = f x`' and '`id x = x`').

The view synthesis heuristics cannot discover the appropriate view for `reverse` in terms of `foldl`, because they rely on being able to prove the property true as a direct result of the synthesised view, without any further induction phases. Assuming that the model for `foldl` does not contain rewrite rules for '`foldl (converse(:)) []`', two induction steps would be necessary to prove the property, and hence the desired view would be rejected.

[1]In general, a view between functions with non similar types may be created provided that the functions belong to a type description. If they are simply described functions with non similar types, then in the current implementation, no attempt will be made to synthesise a view between them.

Although we have successfully found a suitable function, it is clear that we have missed a component that would have been useful if reverse was not present in the library. The recall figure for this example is therefore 50%.

An alternative model for reverse is:

```
{n}  reverse [1..n] == [n,n-1..1]
```

This stands the risk of retrieving functions that only behave correctly when their arguments have the form `[1..n]`, for example a function:

```
last2first l = [last l,last l-1..hd l]
```

would be sufficient to prove the above property.

In the experimental library there are no such functions, and since `id [1..n] ~== [n,n-1..1]`, only `reverse` is retrieved, giving a recall of 1, 100% precision and 50% completeness.

It is interesting to note that the more specialised type inferred for reverse in the second model (`[num]->[num]`) gives a larger number of candidate models which have a similar type — in this case 22 of them. These are made up from twelve components of type `[num]->[num]`, and the ten components similar to the type `[*]->[*]`. The precision that would be achieved through the retrieval of all similarly typed functions is therefore less than 5%.

8.4.2 The function 'interleave'

In the previous cases, the proof of the models properties ensures a much greater precision than retrieval based on similar types alone. The model for **interleave** provides an example in which the proof of the property does not have such a dramatic effect on precision. The expression `interleave x xs` denotes the list of all ways of inserting `x` into the list `xs`.

```
{a as xs ys} xs++ys=as |- member (interleave a as) (xs++[a]++ys)
```

This model retrieves only the function `interleave` and this is the only suitable function, so the recall is 1, precision 100% and completeness 100%. The only similarly typed component is `undef::*` and hence the type based precision would be 50%.

8.4.3 The function 'justline'

Each of the previous examples have produced high levels of precision. The retrieval of a line justification function provides an example in which only a low level of precision is achieved. `justline` extends a list of characters to a given length by adding spaces alongside existing spaces in the line as evenly as possible. Three obvious properties that we could describe are: the length of the result list, the fact that the result list may only contain additional spaces and the distribution of the additional elements within the list. The last of these properties is by far the hardest to describe, and so to keep our model simple, we include only the first two properties:

```
{n cs}  #cs<=n |- #justline n cs == n ;;
        is_each (justline n cs--cs) (=' ')
```

Unfortunately these properties are also true of several other justification functions in the library, including the left, centered and right justification functions along with their truncating counterparts (which always produce lists of length n). The total number of retrieved components is eight, only one of which is suitable, therefore the precision is 12.5%, the completeness 100% and the similar type precision is 5%.

8.4.4 The selector function 'fst4'

Imprecise retrieval not only stems from the use of a property which is true of many library components; in the case of simple components it can also be a result of our view synthesis being too flexible. This is especially true of the collection of tuple selector functions contained in the experimental library. An example of these is the function `fst4` which selects the first element from a 4-tuple.

```
>{a b c d}  fst4 (a,b,c,d) == a
```

Since the view synthesis allows re-ordering of tuple elements, this model retrieves four components (the four selector functions). In all but one case, the view which gives the correct behaviour of the component is as complex as the component itself, and hence only one component is actually suitable. The vast majority of components in the library are considerably more complex than the views which we can generate and so this example of imprecise retrieval is completely atypical. It might also be possible to improve precision in cases such as these by including some form of viewing restriction in such simple library models.

Another point raised by this example is the large number of functions which have similar types to the tuple selector functions. Due to the type equivalence of curried and uncurried functions, the tuple selectors of the appropriate arity match any function which returns one of its argument types.

8.5 Results

The above examples demonstrate some of the extremes of retrieval rather than typical cases. A more balanced view of the performance of model based retrieval can be obtained from the retrieval results of 30 requests for components in the library. The recall, precision and completeness results for these components are summarised in figures 8.1, 8.2 and 8.3 (Appendix C contains specific examples of these requests).

Figure 8.1 shows the distribution of retrieval requests between various levels of precision, the majority of requests resulting in 100% precision and the mean precision being 79%.

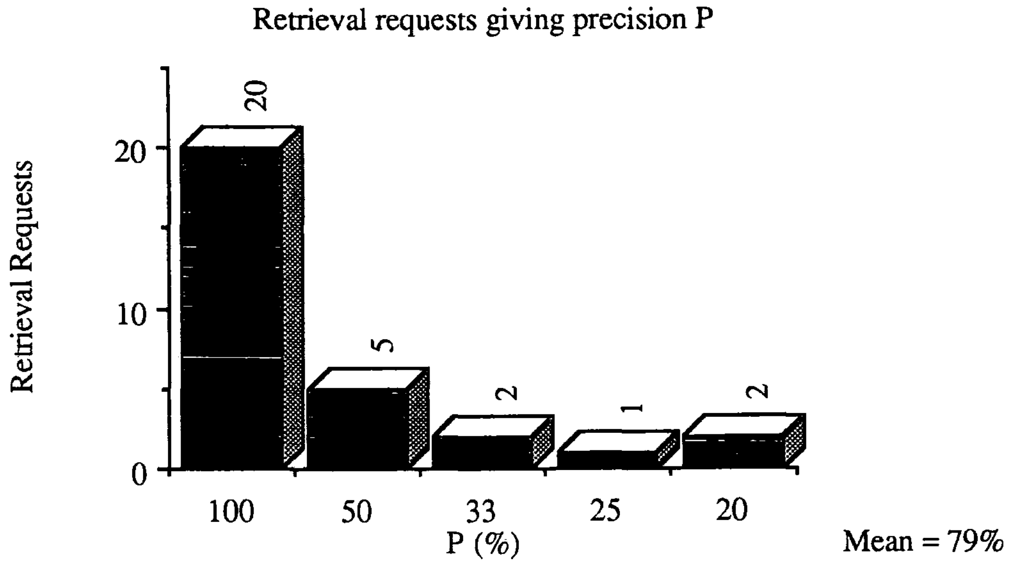

Figure 8.1: Retrieval requests against level of precision

Figure 8.2 shows the distribution of retrieval requests between various levels of completeness. Again the majority of requests achieve 100% with five requests only retrieving one out of two possible components and one request retrieving one out of three. The mean completeness is 89%.

The distribution of retrieval requests between recall sizes given in figure 8.3 is directly related to the distribution for precision. This is because all the retrieval requests only produced one suitable component. The mean number of components recalled is 1.6.

It is interesting to compare this figure with level of recall that would be obtained if all components with similar types were retrieved. Rather than using the limited set of 30 retrieval requests, the level of recall has been established for requests based on the type of each component in the library. Figure 8.4 summarises this information and makes it clear that some components have large numbers of

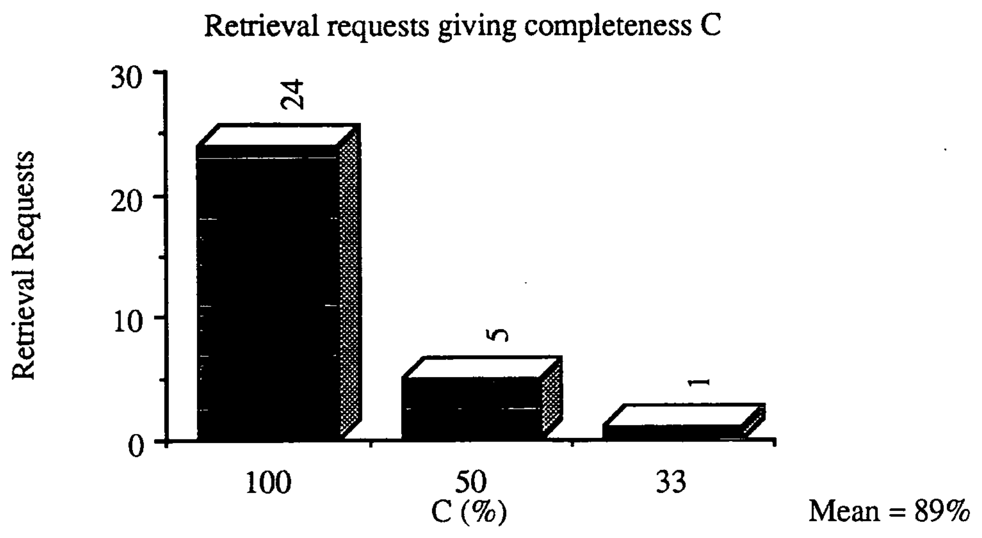

Figure 8.2: Retrieval requests against level of completeness

similarly typed components. The mean figure is 11, which would give a recall of 11 in the case of retrieval based on similar types alone.

These figures demonstrate that the method of model based retrieval can produce high levels of precision and completeness, even when important properties of the model are not described. However, there are several factors that suggest that these figures might be lower in practice. Firstly, the experimental models have been formulated by a person who is already well aware of the library's contents. This means that the experimental requests have tended to correspond closely to components in the library, whereas in a more realistic situation the correspondence may not be as close. It has also been assumed that the library models contain sufficient heuristic information to prove the requested properties and that sufficient time is allowed for the property prover to perform the proof.

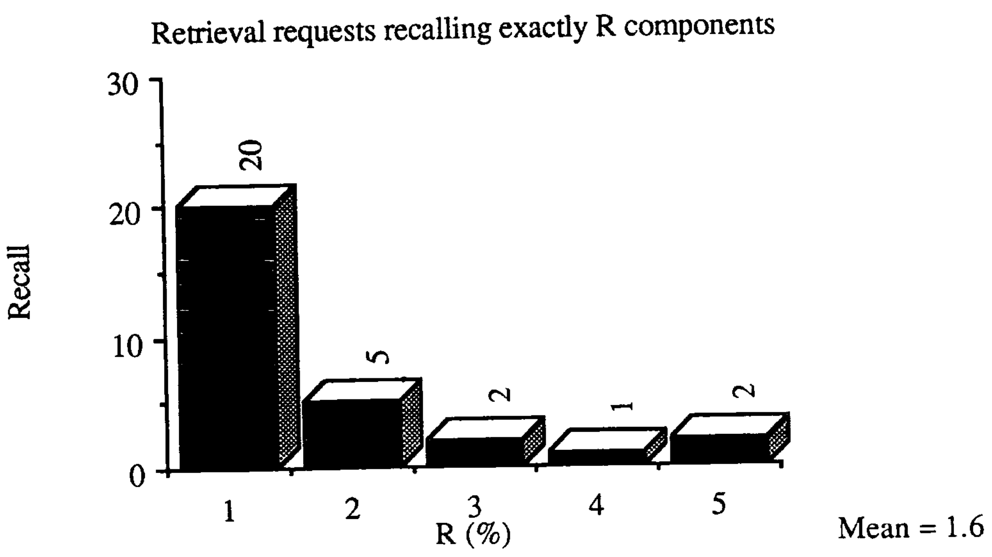

Figure 8.3: Retrieval requests against size of recall

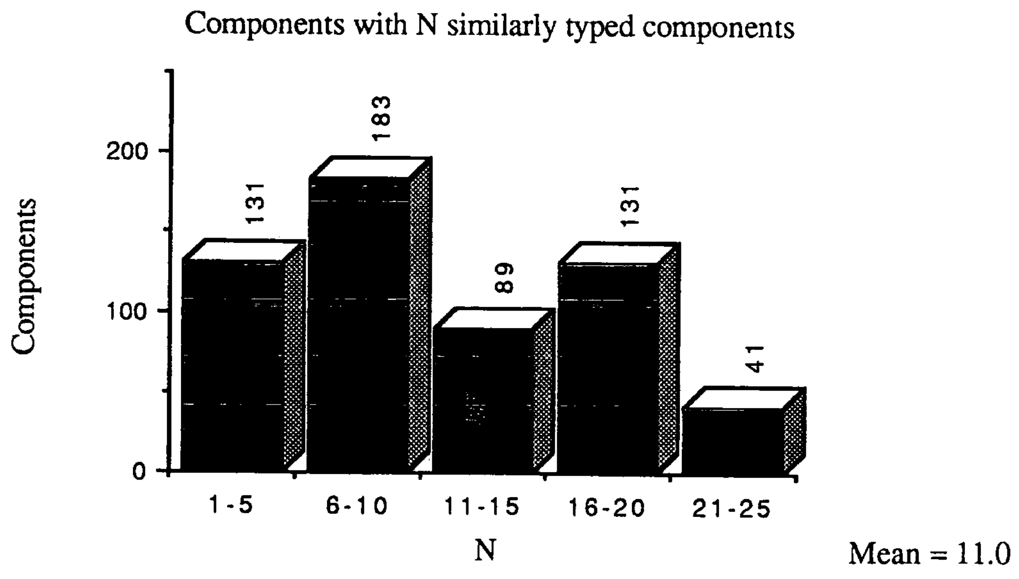

Figure 8.4: Library components against number of similarly typed components

Chapter 9

Conclusions

9.1 Summary of Thesis

This thesis has investigated some of the problems of component libraries and component library retrieval in particular. It has identified the use of names to describe the function of components as a major deficiency of current approaches to component library retrieval. In particular, limiting the possibility of re-use across application domains and leading to low retrieval precision.

A new method of component library retrieval, which uses formal property models rather than names to describe the function of components, has been suggested and developed. This method uses small 'models' to describe only the key functions of components.

The functional programming language Miranda has been extended to provide a notation suitable for describing components. This language is called Miramod. The most important extension provided by Miramod is the ability to describe functions implicitly using equalities as well as explicitly using definitions.

A crucial part of this new method is the comparison of component models.

Models can be compared not just for equivalence but for their similarity to one another. The ability to conclude that two models are 'similar' in some sense is crucial to the method, not only because it should be capable of retrieving components that are similar to the desired component but also because two similar models may be used to describe the same component.

Models can be compared using views. These relate the types and functions of one model to the types and functions of another. Given a view between models it is possible to compare them by attempting to prove the properties of one model from the properties of another. This comparison involves two processes: discovering an appropriate view between models and then proving the properties of one model from another.

Two approaches to view discovery have been investigated. One uses type information to relate similarly typed functions to one another. This method is crude but relatively simple and efficient to implement. The other more interesting approach is to create the view as the properties of the model are proved. This process is called view synthesis, and is capable of creating more complex views than a method based on types alone since it can use information available during the theorem proving process to decide on views that will assist the proof of properties.

The design of a prototype property prover based on work by Boyer and Moore is described. Particular detail is given to the heuristics which are not part of Boyer and Moore's theorem prover and are related to the task of model comparison rather than theorem proving. The property prover does not work directly on Miramod models but relies on a compiler to convert the Miramod notation into a few basic constructs. This compiler follows conventional compilation techniques for functional languages closely, however there are some parts of the design that are specifically aimed at producing component descriptions suitable for theorem proving rather than execution.

9.2 Critical Assessment

The model based approach to component library retrieval meets its principal objective of not relying on names to describe the function of components. The matching process on which retrieval is based treats the names used within models simply as place holders and does not interpret the model in any way other than by its form.

A surprising result of the research is the scale down in size of a component description that can be achieved by using a model which only includes the key properties or functions of the component. If they are well chosen, these key properties can provide a very informative description of the component. Given the limitations of the property prover (particularly in terms of the limited complexity of the proofs that it can achieve) the use of small models as component descriptions is essential.

The prototype Miramod compiler and property prover have demonstrated that models can be automatically compared provided:

- Sufficient information is provided by the library model on how to prove properties about itself.
- The models are at similar levels of abstraction, so that the view between models is not over complex.
- The size of the models is sufficiently small.

To a limited extent the views between models can be synthesised by the property prover as well as being generated on the basis of type information. Although the definition of a syntactically permissible view given in this thesis and the method of view synthesis allow components with similar but not identical functions to be compared, both need to be extended to allow larger discrepancies between components.

The experimental library and retrieval system have demonstrated a high degree of precission, improving on type based retrieval by an order of magnitude. These experiments have indicated that the definition of similar type used is not sufficiently flexible to retrieve some components. It is clear that added flexibility in the type similarity relation would reduce precission of retrieval on the basis of type alone, thus increasing the need for property statements to achieve precise retrieval.

A number of shortcomings have come to light during the course of this project, both in the method and its implementation.

The efficiency with which components can be matched is a major problem. The maximum acceptable time for retrieval is dependent on the size of the component being retrieved and the potential savings made by retrieving a component. For this reason a model based retrieval system must allow the user to control the maximum retrieval time. If a large component, whose re-use would produce substantial savings, is required then a retrieval time in the order of several hours might be acceptable, but for a small component retrieval times should be in the order of minutes or seconds rather than hours.

Automated theorem proving is a well researched problem, and yet it is only through the storage of domain specific theorem proving knowledge along with component models that the model comparison is feasible. A considerable amount of expertise and effort are needed to add the appropriate theorem proving guidance into component models. This effort is essentially directed at developing a theory for the component and then identifying suitable elements of this theory as heuristic theorems suitable for guiding the theorem proving process. Thus a 'librarian' needs considerable mathematical skill as well as an understanding of formal methods and the property proving process.

The method also relies on a certain amount of user expertise. In particular a re-user must be capable of writing small formal specifications, and also of producing suitable abstractions for the components they require. In the present implementation, the models must be written in Miramod, although in principle other languages could be used provided they can be compiled to the property prover notation. It can be argued that both forms of expertise are required of a software engineer in many parts of the software engineering process, and therefore the need for these skills to retrieve components is not a disadvantage.

A further criticism of the work is that it does not cater for components in which timing considerations are among the key properties of the component. It is therefore unsuitable for libraries of real time and embedded system components.

9.3 Future Directions of Research

The experiments described in this thesis have been based on property models written by the author. Further experiments, based on a collection of re-users, would help to determine the ability of re-users in general to produce abstract property models that are similar to models in the library as well as the ability of librarians to produce suitable models and theories for the components in the library.

An important area for further research is an investigation into the feasibility of model based component library retrieval in the context of larger libraries, larger components and more diverse component types.

One interesting possibility which demands further investigation is the use of property models in other parts of the software engineering process. An obvious application of high level models is in software maintenance, where property models of system components could be used during the maintenance of the system to help understand the role of the component and also to help maintainers isolate components of interest. In this context a view between the component and its model over which the component implies the properties of the model might also be useful as a method of explicitly relating the component to the model.

Finally, the role of component library retrieval within the IPSE and ISF frameworks needs to be investigated. Should component library retrieval be seen as just another tool or method to be integrated within the support environment / factory or does the fundamental nature of the need to store and retrieve objects used and created within the environment mean that a retrieval system also plays a fundamental role?

Appendix A

Glossary

backward chaining Backward chaining is a method of proving a conjecture that starts from the conjecture and works 'back' towards the axioms from which the proof is to be made. The rules of inference are used to derive sub-goals, the proof of which is sufficient to prove the original conjecture. See also *forward chaining.*

base case The base case is the part of an induction scheme which must be proved without assuming an instance of the conjecture true.

chain complete An assertion P is said to be chain complete if whenever

$$ys_0, ys_1, ys_2, \ldots$$

is an infinite sequence (or chain) with limit ys, and

$$P(ys_0), P(ys_1), P(ys_2), \ldots$$

are all true, then $P(ys)$ is true also.

changeables The changeables are the formal parameters of an induction template that are in the measured subset and are changed by at least one substitution in the case analysis.

changing variables The changing variables of an induction scheme are those variables changed by at least one substitution in the case analysis. The changing variables include both measured and unmeasured variables.

clause A clause consists of a set of literals, and is true if and only if one or more of the literals is true.

clause form A standard form in which logical formulae can be expressed consisting of a set of clauses which are all true if and only if the formula which the clauses represent is true.

coercion The conversion of values of one type to corresponding values of another type is referred to as coercion.

complete A proof method is complete provided all true theorems are provable.

completeness In the context of component library retrieval, completeness is a measure of suitable components retrieved relative to the number of suitable components in the library.

constructor A function is a constructor of a type if it returns a value of the type.

conformal definition A conformal definition is one which defines the value of a pattern rather than the value of a function. The pattern contains identifiers whose values are defined as the corresponding part of the pattern produced by the definition body.

defined functions Functions that appear on the left hand side of a definition are called defined functions.

described functions The described functions of a Miramod model are those included in property statements but not defined (ie do not appear in the left hand side of a function definition).

described type In a Miramod model, a described type is either an algebraic type, and algebraic shorthand type or a representation based type. The only other types recognised by Miramod are the built in types.

destructor A function is a destructor of a type if it takes a value of the type but does not return a value of the type.

explicit value A term is an explicit value provided it is one of `T`, `F`, `Undef`, or a shell function applied to explicit values.

explicit value templates A term is an explicit value template if it is an explicit value, or an explicit value which contains one or more variables.

fatbar The fatbar operator is an operator used during the compilation of Miramod. It is written ‖ and defined as `a ‖b = IF (a ==Fail) a b`.

final interpretation In the final interpretation of a set of equalities, two values are considered equivalent unless they can be proved unequal.

forward chaining Forward chaining is a method of proving a conjecture that starts with the axioms from which the proof is to be made and works 'forwards' towards the conjecture. The rules of inference are used to derive new theorems from axioms and theorems that have already been established. See also *backward chaining.*

free functions When comparing property models, free functions are the functions for which there is currently no view.

free variables In an expression, the free variables are the ones not bound by a local definition or lambda abstraction which occurs within the expression itself.

generalisation The type `A` is a generalisation of the type `B` if there is a substitution for variables in the type `A` which gives the type `B`.

higher order functions Higher order functions take functions as parameters and / or return functions as results.

inconsistent types If type `A` is neither a generalisation or specialisation of `B` then `A` and `B` are inconsistent types.

induction case An induction case is a part of an induction scheme which assumes one or more substitution instances of the conjecture being proved whilst attempting to prove the conjecture.

induction step See **induction case**.

initial interpretation In the initial interpretation of a set of equalities, two values are considered unequal unless they can be proved equal.

literal A literal is either an atomic formula or the negation of an atomic formula. The definition of atomic formula depends on the context of a literal. In propositional calculus the formulae are propositions. In predicate calculus the atomic formulae are predicates. In a property prover or BMTP clause the atomic formulae are terms.

measured subset A measured subset of a functions arguments is any subset for which there exists a measure and well founded relation such that for each recursive call of the function in its definition body, the measure applied to the parameters of the recursive call is less than the measure applied to the formal parameters of the function according to some well founded relation. A measured subset of the variables in an induction scheme is similarly defined accept that the measure must decrease for each substitution in the scheme.

non-strict A function is non-strict if its value is not always undefined for arguments that are undefined (eg if `f Undef ==undef` then f is non-strict).

off-side rule The off-side rule is used by Miramod as an alternative to explicit statement terminators or separators. It applies to a variety of language constructs and states that the tokens from which the construct is composed must not appear to the left of (or above) the first token in the construct.

partial functions A partial function is a function which has no defined value for some argument values.

partial properties Partial properties are property expressions which describe the properties of functions over undefined values.

partial structures Partial structures are explicit values which contain the undefined value.

permutative Two expressions are permutative if each is an instance of the other.

precision In the context of component library retrieval, precision is a measure of the number of suitable components retrieved relative to the recall

pretty printer A pretty printer is a tool used to re-format code according to standard layout convention.

properly occurs X properly occurs in Y if X occurs in Y but is not equal to Y.

property expressions An expression which denotes a value `T` or `F` and occurs within a property statement. A property expression always denotes a value (ie it cannot be undefined).

property statement A construct provided by Miramod for describing the properties of functions. It consists of a list of variables enclosed in curly braces, followed by a list of property expressions separated by the symbol ;; and optionally ended by the key-word `with` and a list of local definitions.

recall In the context of component library retrieval, recall is the overall number of components retrieved (both suitable and unsuitable).

recursion group This is a group of functions in which each definition ultimately depends upon the definitions of all the other functions in the group.

resolution A rule of inference or method of theorem proving based on the rule of inference, which is:

$$Q \vee P \,\&$$
$$\tilde{}Q \vee R$$

proves

$$P \vee R$$

resolvent The resolvent is the clause produced by an application of the resolution rule of inference.

retrieval function A retrieve function is a function which converts values of a representation type to values of the type which is described using the representation.

re-use In the context of this thesis re-use means the use of the same software engineering knowledge, methods or products in more than one project or organisation.

re-user A person who attempts to perform some re-use – in particular the user of a component library retrieval system.

shell A function which constructs a unique value for each combination of argument values. It is distinguished by the property that if two expressions are made up from different shells, they can never denote equivalent values. ie if S≠T then

$$(\mathtt{S}\ \mathtt{p}_1 \ldots \mathtt{p}_n) \ \ \mathtt{==}\ \ (\mathtt{T}\ \mathtt{q}_1 \ldots \mathtt{q}_n)$$

All shell functions are constructors.

shell type sets A shell type set for a property prover expression is a set of shell types whose union includes the expression. In other words, it is a set of possible shells returned by the property prover expression. These type sets play a fundamental role in many of the property prover's heuristics.

shell type The class of expressions whose normal forms are based on the same shell (ie the class of expressions whose normal form is (S ...) where S is the shell name). The shell type associated with a particular constructor is denoted by underlining the constructor name.

signature The signature of a type describes its syntax. It consists of a set of functions over the type and the respective types of each of these functions.

software engineering The use of sound enginering principles, science and mathematics to economically produce software systems that are reliable, function on real machines and are useful to man.

sound A rule of inference, decision procedure or theorem prover is sound provided that it can never prove a non theorem true.

specialisation The type A is a specialisation of the type B if there is a substitution s for variables in the type B which gives the type A.

strict A function is strict if its value is undefined whenever any of its arguments are undefined.

strong equality Strong equality returns a value which is either true or false but never undefined. If only one of its arguments is undefined then strong equality is false but if both arguments are undefined then strong equality is true.

supercombinators A supercombinator is a lambda calculus expression of the form

$$\lambda a_1.\lambda a_2 ... \lambda a_n.E$$

where $n >= 0$ and E is not a lambda abstraction, contains no free variables other than a_1 to a_n and any lambda abstraction contained in E are themselves supercombinators.

unchangeables The unchangeables associated with an induction template are those variables that are not changed by any substitutions in the case analysis.

unchanging variables The unchanging variables of an induction scheme are those variables which occur in arguments to the term on which the induction scheme is based and are changed by no substitution in the case analysis. This includes both measured and unmeasured variables.

unification Unification is the process of checking that two terms are the same under some substitution instance and also of discovering the minimal substitution for which they are the same.

validation Validation is the process of attempting to establish the fitness or worth of a product for its original mission. Typically validation is carried out on a set of requirements to ensure that they represent a true picture of what the users of a system have requested and need.

verification Verification is the process of attempting to prove that the product of a phase meets its specification. Typically the code of a system is verified with respect to the specification of the system.

weak equality Weak equality is an executable equality operator. Its value is undefined if either of the terms being comparing are undefined.

well founded relation A well founded relation is one which relates two arguments of the same type according to whether or not one is "less" than the other in a specific sense. The crucial property of well foundedness comes from the restriction that there must be no infinite sequences which decrease according to the well founded relation.

Appendix B

Standard Functions

A number of standard functions are available when writing Miramod descriptions. These include all the functions from the Miranda standard environment and a collection of additional functions. The first section of this appendix gives the definitions of the Miranda standard envirionment functions used in this thesis and the second section gives the definition of additional functions which are made available in Miramod.

B.1 Miranda standard environment

```
>|| Some functions from the Miranda Standard Environment
```

'and' gives the logical conjunction of a list of truth values.

```
>and = foldr (&) True
```

'concat' is the function for concatenating lists.

```
>concat = foldr (++) []
```

'converse' reverses the arguments of a function.

```
>converse f x y = f y x
```

'filter p as' gives the list of elements of 'as' for which the predicate 'p' is true.

```
>filter p [] = []
>filter p (a:as) = a:filter p as, p a
>                = filter p as, otherwise
```

'foldl' converts a list into some other value by placing a given binary operator (function) between elements of the list in a left associative fashion.

foldl o i [x,y,z] == ((i $o x) $o y) $o z

```
>foldl o i [] = i
>foldl o i (a:as) = foldl o (i $o a) as
```

'foldr' converts a list into some other value by placing a given binary operator (function) between elements of the list in a right associative fashion.

foldr o i [x,y,z] == x $o (y $o (z $o i))

```
>foldr o i [] = i
>foldr o i (a:as) = a $o (foldr o i as)
```

'id' is the identity function.

```
>id x = x
```

'iterate f x' gives the infinite list [x, f x, f (f x), ...]

```
>iterate f x = x: iterate f (f x)
```

'last' returns the last element of a list.

```
> last x = x!(#x-1)
```

'limit' takes elements from the front of a list until it finds two consecutive elements that are equal.

```
> limit (a:b:x) = a, a=b
>               = limit (b:x), otherwise
```

'map' applies a function to every element in a list.

```
>map f [] = []
>map f (a:as) = f a:map f as
```

'member' tests for list membership.

```
>member as a = or (map (=a) as)
```

'or' returns the disjunction of a list of boolean values.

```
>or = foldr (\/) False
```

'undef' gives the undefined value.

```
>undef = error "undefined"
```

B.2 Miramod standard environment

```
>|| Standard functions available in Miramod (in addition to the Miranda
>|| standard environment).
```

'alternate' joins two lists together putting elements from one list between elements from the other.

```
>alternate [] ys = ys
>alternate xs [] = xs
>alternate (x:xs) (y:ys) = x:y:alternate xs ys
```

'exists' is true if any elements of the given list are true.

```
>exists  = or
```

'forall' is true if all elements of the given list are true.

```
>forall = and
```

'is_any' is true if the given predicate is true for at least one element of the given list:

```
>is_any l p = or (map p l)
```

'is_each' is true if the given predicate is true for all elements of the given list (see 'is_any').

```
>is_each l p = and (map p l)
```

'subseqs' returns a list of all subsequences of the given list. It is defined so that any finite subsequence contained in a finite initial segment will be produced in a finite time.

```
>subseqs [] = [[]]
>subseqs (x:xs) = []:alternate (tl sxs) (map (x:) sxs)
>                 where
>                    sxs=subseqs xs
```

Appendix C

Example Models

This appendix contains a number of example models. The first section contains property models for substantial software components from either the Durham Miranda library or the standard Unix toolset. Several of them make use of functions from the Miramod standard environment (see appendix B). Since these models are either based on type descriptions or are models of components not in the experimental library, no retrieval statistics are presented with the models.

The second section contains examples of models used to retrieve components from the experimental library, as well as their associated retrieval statistics.

C.1 Models for substantial software components

C.1.1 Make: Maintain and update a context

Two models for make are given here, the first is the simplest, since it concentrates on describing make's ability to check that the current context is consistent (up to date).

```
>up2date:: context -> rules -> bool


>type context
>with
>   newobj:: object -> context -> context
>   incontext:: object -> context -> bool
>   after:: object -> context -> object -> bool
>   emptycontext:: context


>{incontext: o o' c}
>   incontext o emptycontext = False;;
>   incontext o (newobj o c) = True;;
>   o~=o' |- incontext o (newobj o' c) = incontext o c


>{after: o o' c p}
>   after o emptycontext o' = False;;
>   after o (newobj o' c) o' = False;;
>   o~=o' |- after o (newobj o c) o' = incontext c o';;
>   o~=p |- after o (newobj p c) o' = after o c o'


>type rules
>with
>   norules:: rules
>   addrule:: object -> [object] -> rules ->rules


>{up2date: o ds rs c}
>   up2date norules c = True;;
>   up2date (addrule o ds rs) c = is_each ds (after o c) & up2date rs c
```

The second model for make is more complex, because it describes make's ability to update a context so that it becomes consistent.

```
>make:: context -> rules -> context



>type context
>with
>   addobj:: name -> object -> context -> context
>   validname:: name -> context -> bool
>   getobj:: name -> context -> object
>   newcontext:: context



>type rules
>with
>   norules:: rules
>   addrule:: name -> [name] -> ([object]->object) -> rules ->rules
>   getname:: rules -> name
>   getdeps:: rules -> [name]
>   getmkfn:: rules -> ([object]->object)
>   rule_exists:: name -> rules -> bool
>   rest_rules:: rules -> rules



>{context: n m o c}
>   getobj n (addobj n o c) = o;;
>   n~==m |- getobj n (addobj m o c) = getobj n c;;



>   validname n newcontext = False;;
>   validname n (addobj n o c) = True;;
>   n~==m |- validname n (addobj m o c) = validname n c
```

```
>{rules: n ds f rs m}
>   getname (addrule n ds f rs) == n;;
>   getdeps (addrule n ds f rs) == ds;;
>   getmkfn (addrule n ds f rs) == f;;
>   rule_exists n norules == False;;
>   rule_exists n (addrule n ds f rs) == True;;
>   n~=m |- rule_exists n (addrule m ds f rs) == rule_exists n rs
```

```
>{make: rs c}
>   undef ~== #limit (iterate (applyrules rs) c)
>   |-  make rs c == last (limit (iterate (applyrules rs) c))
```

```
>   where
>       applyrules rs c
>           = c, norules = rs \/ ~validname (getname rs) c \/
>                ~is_each (getdeps rs) (converse validname c)
>           = (applyrules (rest_rules rs) .
>               addobj (getname rs) .
>               getmkfn rs (map (getobj c) (getdeps rs))
>              ) c,
>                   otherwise
```

C.1.2 Grep: Search for patterns

The UNIX utility grep searches for lines which match a given pattern. The following model describes both the basic operators for forming patterns and the behavior of grep when given a pattern.

```
>grep:: pattern -> [[char]] -> [[char]]
```

```
>type pattern
>with
>   constpat:: char -> pattern
>   appendpat:: pattern -> pattern -> pattern
>   altpat:: pattern -> pattern -> pattern
>   wildcard::pattern
>   match:: pattern -> [char] -> bool
```

```
>{match: c c' a b s}
>   match (constpat c) [c] == True;;
>   c ~= c' |- match (constpat c) [c'] == False;;
>   match (altpat a b) s == match a s \/ match b s;;
>   match (appendpat a b) s
>       == or [match a (take n s) & match b (drop n s)|n<-[0..#s-1]];;
>   match wildcard [c] == True
```

```
>{grep: pat ls}
>   grep pat ls == filter (is_any (match pat) . subseqs) ls
```

C.1.3 Diff: List the differences between two files

An extremely simple model for diff might be based on the fact that its output can be used to reconstruct one of the compared files from the other.

```
>{f g}
>   edit (diff f g) f = g;;
```

Unforutunately this model is too simple; it is satisfied by trivial functions such as:

```
diff f g = g
edit g f = g
```

The important point that our model has missed is that we want the result of diff to be minimal in some sense. To capture this, we must describe some of the properties of the result of diff.

```
>type edcom, com
>with
>   mked:: com -> num -> num -> [line] -> edcom
>   edit:: [edcom] -> [line] -> [line]
>   size:: [edcom] -> num
```

```
>{c a b ls ecs}
>   size [] == 0;;
>   size (mked c a b ls:ecs) == 1+#ls+size ecs
```

```
>{e f g}
>   edit (diff f g) f == g;;
>   edit e f == g |- size e <= size (diff f g)
```

C.1.4 Lex: Build lexical analysers

The following model of lex considers a lex program as a list of pattern/result pairs. Each pattern can be matched against an input sequence to produce a matching initial segment of the sequence and the corresponding result can be used in conjunction with the initial segment to produce an appropriate output token. As it processes a list of input tokens, lex will always pick the pattern which matches the largest input sequence.

```
>type pattern,in,out
>with
>   lex:: [(pattern,res)] -> [in] -> [out]
>   match:: pattern -> [in] -> [in]
>   resout:: res -> [in] -> out


>{rules i i' p r}
>   lex rules [] == [];;


>   (i',r) == largest fst_less [(match p i,r)|(p,r)<-rules] &&
>   i~=[]
>       |- lex rules i == resout r i':lex rules (i--i')
>
>fst_less (a,b) (c,d) = a<d


>largest l [a] = a
>largest l (a:b:c) = largest l (b:c), a $l b
>                  = largest l (a:c), otherwise
>
```

C.1.5 Join: The relational database operator

The following model is for the UNIX join program, and so the hypothesis that the two input lists are already ordered is included.

```
>{a b as bs n}
>   ordered as & ordered bs
>       |- join n as bs == [a++b|a<-as;b<-bs; #a>n & #b>n & a!n=b!n]
```

```
>ordered [] = True
>ordered [a] = True
>ordered (a:b:x) = a<b & ordered (b:x)
```

C.1.6 Unify: Find a unifying substitution for two expressions

The model for the unfication function involves two functions for testing the results of a unification (`ok` and `val`), as well as a function for performing substitutions. The first property indicates when a unification must succeed, and the second property ensures that the resulting substitution produces two identical expressions.

```
>type expr, sub, sub_res
>with
>   unify:: sub -> expr -> expr -> sub_res
>   ok:: sub_res -> bool
>   val:: sub_res -> sub
>   apply_sub:: sub -> expr -> expr
```

```
>{s s' a b}
>   applysubs s' (applysubs s a) = applysubs s' (applysubs s b)
>       |- ok (unify s a b);;
```

```
>   ok (unify s a b) ==
>       applysubs (val (unify s a b)) a = applysubs (val (unify s a b)) b
```

C.1.7 Awk: Report generator

The model for awk considers an awk program as a list of pattern/action rules. For each input line and each pattern that matches the line, awk applies the corresponding action to produce its output.

```
>type tuple, newtuple, pat, act
>with
>   awk:: [tuple] -> [(pat,act)] -> [newtuple]
>   match:: pat -> tuple -> bool
>   act:: action -> tuple -> [newtuple]
```

```
>{prog t ts}
>       awk [] prog == [];;
>       awk (t:ts) prog == concat [act a t|(p,a)<-prog; match p t] ++
>                                 awk ts prog
```

C.1.8 Diag: Cartesian diagonalisaton of lists

Given two lists, diag[1] produces a list of all pairs consisting of an element from the first list and an element from the second. The important property of diag is that it does this in such a way that even if both lists are infinite, all pairs made up from finite initial segments of the lists will be produced with only a finite amount of work. The use of + to extend the quantification of the lists is therefore essential to capture the desired properties of diag.

The use of a double ampersand (&&) in the first property is crucial, since it ensures that we do not distinguish between False and Undef results to the membership test. The second and third properties give connditions under which False must be returned.

[1]This example was suggested by John Hughes.

```
>diag:: [*]->[**]->[(*,**)]
```

```
>{diag: a as+ b bs+ fas}
>   member as a && member bs b == member (diag as bs) (a,b);;
>   ~member as a |- ~member (diag as bs) (a,b);;
>   ~member bs b |- ~member (diag fas bs) (a,b)
```

C.2 Models for retrieval from the experimental library

C.2.1 dropwhile

`dropwhile` is a function for droping initial segments which meet a predicate from lists.

```
>{f x}  ~f (hd (dropwhile f x)) \/ dropwhile f x = []
```

Precision 50%, completeness 100%, recall 1.

C.2.2 filter

`filter` keeps only those elements of a list that meet the given predicate.

```
{f x}  is_each (filter f x) f
```

This is a property of the function takewhile as well as of filter, so the precision is 50%, completeness 100% and recall 2.

C.2.3 drop

`drop` removes the first n elements from a list.

```
{s n}   (#s>=n) |- #drop n s == #s-n
```

Precision 100%, completeness 100%, recall 1.

C.2.4 lastn

`lastn` takes the last `n` elements from a list.

```
{n l a b}  l=a++b & #b=n |- last n l = b
```

Precision 100%, completeness 100%, recall 1.

C.2.5 betweenlists

`betweenlists` returns the first list which lies between the two given lists.

```
>{a b l}  between_lists a b l ~= []
>         |- member (subseqs l) (a++between_lists a b l++b)
```

Precision 100%, completeness 100%, recall 1.

C.2.6 parts

`parts` returns the list of all ways of partitioning into a list of sublists.

```
>{xs}   is_each (=xs) (map concat (parts xs))
```

Precision 100%, completeness 100%, recall 1.

C.2.7 before

`before` returns the initial part of the given list, up to the given element.

```
>{x xs}  ~member (before x xs) x
```

Precision 50%, completeness 100%, recall 2.

Appendix D

Miramod Grammar

This appendix gives the context free grammar for the extensions to Miranda which constitute Miramod. These are defined in terms of non terminals from the grammar for Miranda found in the Miranda manual[45].

D.1 Conventions

The conventions used are similar to those used in the Miranda manual:

Non terminals are represented by lower case words and in the case of non terminals from the Miranda grammar these are in italics. Non terminals defined here replace any definition of the same non terminal in the Miranda grammar. Terminals are in bold font.

The production symbol is written ':=' and alternative productions are written on separate lines.

For any non-terminal x,

x* means any number of occurrences of x,

x?	means x is optional,
x-list	means one or more x's separated by commas
x(;)	means that all the tokens of x must lie below or to the right of the first, and if the token succeding x is to the right of x then it must be a semicolon (this is the off-side rule, see page 62).

D.2 Miramod grammar

script:= decl*

decl:= *def*
tdef
spec
pstatement

pstatement:= { pstatemain (;)

pstatemain:= pvar* } pexprs *whdefs*?

pvar:= quantvar
quantvar-list :: type (;)

quantvar:= *identifier*
identifier +

pexprs:= pexpr
pexpr **(gen)**
pexpr **(elim)**
var-list :: type (;)
pexpr ;; pexprs
pexpr **(gen)** pexprs
pexpr **(elim)** pexprs

```
            var-list :: type (;) pexprs

pexpr:= ( pexpr )
            pexpr |- pexpr
            pexpr == pexpr
            pexpr ~== pexpr
            pexpr => pexpr
            pexpr && pexpr
            pexpr << pexpr
            exp
```

Appendix E

The proof of 'rtree'

The example is based on a tree data structure, defined as the algebraic type:

```
tree * ::= Node * [(tree *)]
```

To improve readability the tree and list selector functions are replaced by the following function names:

```
    SEL-Node-1  ==  tval
    SEL-Node-2  ==  subts
       SEL-:-1  ==  hd
       SEL-:-2  ==  tl
```

The functions `rtree` and `rlist` are defined over this type; `rtree` reverses a tree and `rlist` reverses a list of trees.

```
rtree (Node v l) = Node v (rlist l)

rlist [] = []
rlist (a:x) = rlist x ++ [rtree a]
```

The compiled versions of these functions are:

```
rtree a = ifdef a (Node v (rlist l))

rlist x
    = ifdef x
        (IF (x==[]) []
            (rlist (tl x) ++ [rtree (hd x)])
```

The built in definition of the list append operator is:

```
a ++ b
    = ifdef a
        (IF (a==[]) b
            (hd a:(tl a ++ b))
```

The following induction templates are supplied for the three functions:

Applications: `rtree a, rlist x`
Measured subset: {`a,x`}
Changables: {`a,x`}
Unchangables: {}
Case1:

Condition: `REC-: x \/REC-Node`
Substitutions:{<`a, hd x`>, <`x, tl x`>}
{<`a, hd (subts a)`>, <`x, tl (subts a)`>}

This induction template is justified by the measure: `count (a:x)` where count is defined by:

```
count Undef = 0
count [] = 1
count (a:x) = 1+count a + count x
count (Node v l) = 1+count l
```

Since:

```
Rec-: x \/ Rec-Node a
|-
    count (hd x: tl x) < count (a:x) &
    count (hd (subts a):tl (subts a)) < count (a:x)
```

There is also an induction template associated with the list append operator:

Applications: `a++b`
Measured subset: {`a`}
Changables: {`a`}
Unchangables: {}
Case1:
    Condition: `REC-: a`
    Substitutions:{<`a, tl a`>}

E.1 Proof outline

The aim is to prove the property:

```
{a} rtree (rtree a ) = a
```

from the above definitions and induction templates.

An outline of the proof is given initially. This gives the main steps involved in the proof but ignores intermediate simplifications and the proof of bases cases.

Initially the function definitions are opened up to try and introduce more than one function from the recursion group.

```
    Rec-Node a &
    REC-: subts a
 -> a == rlist (rlist (tl (subts a)) ++ [rtree (hd (subts a))])
```

Eliminating the destructors **subts**, **hd**, and **tl** gives:

```
rlist (rlist x ++ [rtree a]) == a:x
```

The conjecture is now in a suitable form for induction on the mutually recursive functions since both appear in the conjecture. Infact, the induction template for **rlist** and **rtree** is the only one applicable since the ++ operator has a term in the position of a changable variable. The resulting induction step is:

```
    REC-Node a \/ REC-: x &
    rlist (rlist (tl x) ++ [rtree (hd x)]) == (hd x):(tl x) &
    rlist (rlist (tl (subts a)) ++ [rtree (hd (subts a))])
    == (hd (subts a)):(tl (subts a))
|-
rlist (rlist x ++ [rtree a]) == a:x
```

Opening up (rlist x) and (rtree a) in the conclusion gives nine cases, the first eight of which simplify trivially. The final case (after elimination of destructor terms) is:

```
    rlist (rlist x ++ [rtree a]) == a:x &
    rlist (rlist y ++ [rtree b]) == b:y
 -> rlist ((rlist x ++ [rtree a]) ++ [Node v (rlist y++[rtree b])])
    == Node v (b:y):a:x
```

The equality hypotheses are then used to cross fertilise, giving:

```
   rlist ((rlist x++[rtree a])++[Node v (rlist y++[rtree b])])
== Node v (rlist (rlist y++[rtree b])):rlist (rlist x++[rtree a])
```

The terms `(rlist x++[rtree a])` and `(rlist y ++ [rtree b])` are then generalised giving:

```
rlist (x ++ [Node v y]) == Node v (rlist y):rlist x
```

Both of the induction hypotheses have been used up and the conjecture is now ready for another induction. Only one induction scheme for `++` which inducts on x is now possible (the mutually recursive scheme does not apply because the function `rtree` does not appear in the conjecture). The induction step is therefore:

```
   REC-: x &
   rlist (tl x ++ [Node v y]) == Node v (rlist y):rlist (tl x)
-> rlist (x ++ [Node v y]) == Node v (rlist y):rlist x
```

This simplifies, with the elimination of the destructor term `(tl x)` to:

```
   rlist (x ++ [Node v y]) == Node v (rlist y):rlist x
-> Node v (rlist y):(rlist x ++ [rtree x])
   == rlist (x ++ [Node v y]) ++ [rtree x]
```

Finally, using the equality hypothesis to cross fertilise, the clause can be simplified to:

```
true
```

Hence the conjecture has been proved for defined values.

E.2 The complete proof

The property to be proved is:

```
b == rtree (rtree b)
```

which simplifies, opening up `(rtree b)`, to:

```
1
    $Undef == b
 -> b == rtree $Undef
```

and

```
2
    $Undef ~== b
 -> b == rtree (Node (tval b) (rlist (subts b)))
```

Clause 1 simplifies to:

```
true
```

Clause 2 simplifies , opening up `(rtree (Node ...))` to:

```
    $Undef ~== b
 -> b == Node (tval b) (rlist (rlist (subts b)))
```

which simplifies, opening up `(rlist (subts b))`, to:

```
2.1
    REC_Node b &
    $Undef == subts b
 -> b == Node (tval b) $Undef
```

and

```
2.2
    REC_Node b &
    $Undef ~== subts b &
    [] == subts b
 -> b == Node (tval b) []
```

and

```
2.3
    REC_Node b &
    $Undef ~== subts b &
    [] ~== subts b
 -> b == Node (tval b) (rlist (rlist (tl (subts b)) ++
                          [rtree (hd (subts b))]))
```

Clause 2.1 simplifies to:

```
    REC_Node b &
    $Undef == subts b
 -> b == Node (tval b) $Undef
```

Eliminating destructors gives:

```
    REC_Node (Node B C) &
    $Undef == C
 -> Node B C == Node B $Undef
```

which simplifies to:

```
true
```

Clause 2.2 simplifies to:

```
    REC_Node b &
    [] == subts b
 -> b == Node (tval b) []
```

Eliminating destructors gives:

```
    REC_Node (Node D E) &
    [] == E
 -> Node D E == Node D []
```

which simplifies to:

```
true
```

Elimination of destructors from 2.3 gives:

```
    REC_Node (Node F G) &
    $Undef ~== G &
    [] ~== G
 -> Node F G == Node F (rlist (rlist (tl G)++[rtree (hd G)]))
```

which simplifies to:

```
   $Undef ~== G &
   [] ~== G
-> G == rlist (rlist (tl G)++[rtree (hd G)])
```

Eliminating destructors gives:

```
   $Undef ~== D:E &
   [] ~== D:E
-> D:E == rlist (rlist E++[rtree D])
```

This cannot be simplified further and must by proved by induction. We will induct according to the following scheme:

```
Accounts for: rtree D, rlist E
Score: 1
Changing and Unchanging: [D,E] []
Cases:
Hypothesis: [REC-Node D \/ REC-: E]
     [hd E/D,tl E/E]
     [hd (subts D)/D,tl (subts D)/E]
```

Giving the clauses:

```
2.3.1
    ~ REC-Node D \/ REC-: E
 -> rlist (rlist E++[rtree D]) == D:E
```

and

```
2.3.2
    REC-Node D \/ REC-: E &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D)
 -> rlist (rlist E++[rtree D]) == D:E
```

Clause 2.3.1 simplifies, opening up (rtree D), to:

```
2.3.1.1
    ~ REC-Node D &
    ~ REC-: E &
    $Undef == E
 -> rlist $Undef == D:E
```

and

```
2.3.1.2
    ~ REC-Node D &
    ~ REC-: E &
    $Undef ~== E
 -> rlist ([$Undef]) == D:E
```

Clause 2.3.1.1 simplifies to:

```
    ~ REC-Node D
 -> $Undef ~== E
```

This is evidently false (both litterals are "irelevant") therefore the conjecture is not true for undefined values.

Clause 2.3.1.2 simplifies to:

```
true
```

Clause 2.3.2 simplifies, opening up (`rtree D`), to:

```
2.3.2.1
    REC-Node D &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D) &
    $Undef == E &
    $Undef == subts D
 -> rlist ($Undef++[Node (tval D) $Undef]) == D:E
```

and

```
2.3.2.2
    REC-Node D &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D) &
    $Undef == E &
    $Undef ~== subts D &
    [] == subts D
 -> rlist ($Undef++[Node (tval D) []]) == D:E
```

and

```
2.3.2.3
    REC-Node D &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D) &
    $Undef == E &
    $Undef ~== subts D &
    [] ~== subts D
 -> rlist ($Undef++[Node (tval D) (rlist (tl (subts D))++
                                      [rtree (hd (subts D))])])
    == D:E
```

and

```
2.3.2.4
    REC-Node D &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D) &
    $Undef ~== E &
    [] == E &
    $Undef == subts D
 -> rlist ([]++[Node (tval D) $Undef]) == D:E
```

and

```
2.3.2.5
    REC-Node D &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D) &
    $Undef ~== E &
    [] == E &
    $Undef ~== subts D &
    [] == subts D
 -> rlist ([]++[Node (tval D) []]) == D:E
```

and

```
2.3.2.6
    REC-Node D &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D) &
    $Undef ~== E &
    [] == E &
    $Undef ~== subts D &
    [] ~== subts D
 -> rlist ([]++[Node (tval D) (rlist (tl (subts D)) ++
                                [rtree (hd (subts D))])])
    == D:E
```

and

```
2.3.2.7
    REC-Node D &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D) &
    $Undef ~== E &
    [] ~== E &
    $Undef == subts D
 ->    rlist ((rlist (tl E)++[rtree (hd E)])++[Node (tval D) $Undef])
    == D:E
```

and

```
2.3.2.8
    REC-Node D &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D) &
    $Undef ~== E &
    [] ~== E &
    $Undef ~== subts D &
    [] == subts D
 ->    rlist ((rlist (tl E)++[rtree (hd E)])++[Node (tval D) []])
    == D:E
```

and

```
2.3.2.9
    REC-Node D &
    rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
       rlist (rlist (tl (subts D))++[rtree (hd (subts D))])
    == hd (subts D):tl (subts D) &
    $Undef ~== E &
    [] ~== E &
    $Undef ~== subts D &
    [] ~== subts D
 -> rlist ((rlist (tl E)++[rtree (hd E)]) ++
       [Node (tval D) (rlist (tl (subts D))++[rtree (hd (subts D))])])
    == D:E
```

Clause 2.3.2.1 simplifies to:

```
true
```

Clause 2.3.2.2 simplifies to:

```
true
```

Clause 2.3.2.3 simplifies to:

```
true
```

Clause 2.3.2.4 simplifies to:

```
true
```

Clause 2.3.2.5 simplifies to:

```
true
```

Clause 2.3.2.6 simplifies to:

```
true
```

Clause 2.3.2.7 simplifies to:

```
true
```

Clause 2.3.2.8 simplifies to:

```
true
```

Elimination of destructors from 2.3.2.9 gives:

```
REC-Node (Node F G) &
rlist (rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
rlist (rlist (tl G)++[rtree (hd G)]) == hd G:tl G &
$Undef ~== E &
```

```
   [] ~== E &
   $Undef ~== G &
   [] ~== G
-> rlist ((rlist (tl E)++[rtree (hd E)]) ++
          ([Node F (rlist (tl G)++[rtree (hd G)])])
   == Node F G:E
```

Eliminating destructors gives:

```
   rlist (++ rlist (tl E)++[rtree (hd E)]) == hd E:tl E &
   rlist (rlist I++[rtree H]) == H:I &
   $Undef ~== E &
   [] ~== E &
   $Undef ~== H:I &
   [] ~== H:I
-> rlist ((rlist (tl E)++[rtree (hd E)])++
          [Node F (rlist I++[rtree H])])
   == Node F (H:I):E
```

Eliminating destructors gives:

```
   rlist (rlist BA++[rtree J]) == J:BA &
   rlist (rlist I++[rtree H]) == H:I &
   $Undef ~== J:BA &
   [] ~== J:BA
-> rlist ((rlist BA++[rtree J])++[Node F (rlist I++[rtree H])])
   == Node F (H:I):J:BA
```

Using the equality hypotheses produces:

```
   rlist (rlist I++[rtree H]) == H:I
-> rlist ((rlist BA++[rtree J])++([Node F (rlist I++[rtree H])]))
   == Node F (H:I):rlist (rlist BA++[rtree J])
```

Using the equality hypotheses produces:

```
rlist ((rlist BA++[rtree J])++[Node F (rlist I++[rtree H])])
== Node F (rlist (rlist I++[rtree H])):rlist (rlist BA++[rtree J])
```

The clauses resulting from generalisation are:

```
rlist (BC++[Node F BB]) == Node F (rlist BB):rlist BC
```

This clause cannot be simplified further and must by proved by induction. We will induct according to the following scheme:

```
Accounts for: rlist BC, BC++[Node F BB]
Score: 2.0
Changing and Unchanging: [BC] []
Cases:
Hypothesis: [REC-: BC]
     [tl BC/BC]
```

Giving the clauses:

```
2.3.2.9.1
    ~ REC-: BC
 -> rlist (BC++[Node F BB]) == Node F (rlist BB):rlist BC
```

and

```
2.3.2.9.2
    REC-: BC &
    rlist (tl BC++[Node F BB]) == Node F (rlist BB):rlist (tl BC)
 -> rlist (BC++[Node F BB]) == Node F (rlist BB):rlist BC
```

Clause 2.3.2.9.1 simplifies, opening up (rlist BC), to:

```
2.3.2.9.1.1
    ~ REC-: BC &
    $Undef == BC
 -> rlist $Undef == Node F (rlist BB):$Undef
```

and

```
2.3.2.9.1.2
    ~ REC-: BC &
    $Undef ~== BC
 -> rlist ([Node F BB]) == [Node F (rlist BB)]
```

Clause 2.3.2.9.1.1 simplifies to:

```
$Undef ~== BC
```

(we have already assumed that the conjecture is false in the undefined case, so this clause is ignored.)

Clause 2.3.2.9.1.2 simplifies, opening up (rlist ([Node F BB])), to:

```
true
```

Clause 2.3.2.9.2 simplifies, opening up (++ BC ([Node F BB])), to:

```
    REC-: BC &
    rlist (tl BC++[Node F BB]) == Node F (rlist BB):rlist (tl BC)
 ->    Node F (rlist BB):rlist BC
    == rlist (tl BC++[Node F BB]))++[rtree (hd BC)]
```

Eliminating destructors gives:

```
   REC-: (H:I) &
   rlist (I++[Node F BB]) == Node F (rlist BB):rlist I
-> Node F (rlist BB):rlist (H:I) == rlist (I++[Node F BB])++[rtree H]
```

which simplifies, opening up (rlist (H:I)), to:

```
   rlist (I++[Node F BB]) == Node F (rlist BB):rlist I
-> Node F (rlist BB):(rlist I++[rtree H])
   == rlist (I++[Node F BB])++[rtree H]
```

Using the equality hypotheses produces:

```
   Node F (rlist BB):(rlist I++[rtree H])
== (Node F (rlist BB):rlist I)++[rtree H]
```

which simplifies, opening up ((Node F (rlist BB):rlist I)++[rtree H]), to:

```
true
```

Bibliography

[1] L. Augustsson, "Compiling Pattern Matching," in *Functional Programming Languages and Computer Architecture, Nancy,* pp. 368-81. Springer Lecture Notes in Computer Science, vol 201, September 1985.

[2] D.R.Barstow, "Domain Specific Automatic Programming", *IEEE Trans. Software Eng.,* Vol. SE-11, No.11, November 1987.

[3] W.Bartussek and D.L.Parnas, "Using Assertions about Traces to Write Abstract Specifications for Software Modules", in *Software Specification Techniques*, ed. N. Gehani and A.D. McGettrick, Addison-Wesley, 1986.

[4] V.R.Basili, "Reusing Existing Software", Tech.Report UMIACS-TR-88-72, University of Maryland, October, 1988.

[5] V.R.Basili and H.D.Rombach, "Towards a Comprehensive Framework for Re-use: A Re-use Enabling Software Evolution Environment", Tech.Report UMIACS-TR-88-92, University of Maryland, December,1988.

[6] R.Balzer, "Transformational Implementation: An Example", *IEEE Trans. Software Eng.,* Vol. SE-7, No. 1, January 1981.

[7] R.Balzer, "A 15 Year Perspective on Automatic Programming", *IEEE Trans. Software Eng.,* Vol. SE-11, No.11, November 1987.

[8] T.J.Biggerstaff and A.J.Perlis, Foreword to "Special Issue on Software Reusability", *IEEE Transaction on Software Engineering,* Vol. SE-10, No. 5, September 1984.

[9] R.Bird and P.Wadler, *Introduction to Functional Programming,* Prentice Hall, 1988.

[10] B.W.Boehm, "Software and Its Impact: A Quantitative Assessment", *Datamation,* 19, No. 5, pp 48-59, May 1973.

[11] B.W.Boehm, *Software Engineering Economics,* Prentice-Hall, 1981.

[12] R.S. Boyer and J.S. Moore, *A Computational Logic*, New York: Academic Press. 1979.

[13] A. Bundy, *The Computer Modelling of Mathematical Reasoning*, Academic Press Inc., London, 1983.

[14] R.M. Burstall and J. Darlington, "A transformation system for developing recursive programs", *J. Assoc. Computing. Machinery.* 24(1), 1977.

[15] J.Buxton, *Stoneman: Requirements for Ada Programming Support Environments*, US Department of Defence, 1980.

[16] F.W.Calliss et al. "A Knowledge-Based System for Software Maintenance", *Proc. Conf. on Software Maintenance-1988,* pp. 319-324. IEEE Computer Society Press, 1988.

[17] F.W.Calliss, "Problems With Automatic Restructurers", *SIGPLAN Notices,* vol.23, no.3, pp.13-21, March 1988.

[18] M.J.Cavaliere, "Hartford reusable code a plus for productivity," *ITT COMMNET,* vol. 2, Sept.1982.

[19] G.A.Curry and R.M.Ayers, "Experience with Traits in the Xerox Star Workstation", *IEEE Trans. on Software Eng.,* Vol. SE-10, No.5, pp. 519-527.

[20] J.Darlington, "An Experimental Program Transformation and Synthesis System", *Artificial Intelligence,* vol. 16, pp. 1-46, 1981.

[21] R.Prieto-Diaz and P.Freeman, "Classifying Software for Reusability", *IEEE Software,* Vol. 4, No. 1, 1987.

[22] E.W.Dijkstra, "Notes on Structured Programming", *Structured Programming,* New York: Academic, 1972.

[23] S.F.Fickas, "Automating the Transformational Development of Software", *IEEE Trans. Software Eng.,* Vol. SE-11, No.11, November 1987.

[24] N. Gehani and A.D. McGettrick (editors), *Software Specification Techniques*, Addison-Wesley, 1986.

[25] J.A.Goguen, "Parameterized Programming", *IEEE Trans. on Software Eng.,* Vol. SE-10, No.5, pp.519-527.

[26] G.Goos and J.Hartmanis, *Fundamentals of Artificial Intelligence*, Springer Lecture Notes in Computer Science, vol 232, 1986.

[27] I. Hayes (ed), *Specification Case Studies*, Prentice-Hall International (UK), 1987.

[28] C.Hawksley, "Coercion In Class-based Software Environments", Ph.D. thesis, University of Keele, 1987.

[29] E.Horowitz and J.B.Munson, "An Expansive View of Reusable Software", *IEEE Trans. on Software Eng.,* Vol. SE-10, No.5, pp. 477-487.

[30] J.Hughes, "Why Functional Programming Matters", *The Computer Journal,* Vol. 32, No.2, pp. 98-107.

[31] S.C. Johnson, "Yacc — Yet Another Compiler-Compiler. Comp Sci. Tech. Rep. No. 32.", Bell Laboratories: Murray Hill, New Jersey.

[32] C.B.Jones, *Systematic Software Development Using VDM,* Prentice-Hall (UK), 1986.

[33] B.W.Kernighan, "The Unix System and Software Reusability", *IEEE Trans. on Software Eng.*, Vol. SE-10, No.5, pp. 513-518, 1984.

[34] M.E. Lesk, "Lex — A lexical analyser Generator. Comp Sci. Tech. Rep. No. 39.", Bell Laboratories: Murray Hill, New Jersey.

[35] S.L.Peyton Jones, *The Implementation of Functional Programming Languages,* Prentice-Hall International, 1987.

[36] R.G.Lanergan and C.A.Grasso, "Software Engineering with Reusable Designs and Code", *IEEE Trans. on Software Eng.,* Vol. SE-10, No.5, pp. 498-501, 1984.

[37] B.P.Lientz, E.B.Swanson and G.E.Tompkins, "Characteristics of Application Software Maintenance", *Communications of the ACM,* Vol.21, No.6, June 1978.

[38] B.P.Lientz, E.B.Swanson, *Software Maintenance Management,* Addison-Wesley, 1980.

[39] B.P.Lientz, "Issues in Software Maintenance", *ACM Computing Surveys,* Vol.15, No.3, September 1983.

[40] P.Mair, "Integrated Project Support Environments, State Of The Art Report", National Computer Center LTD (UK), 1986.

[41] Z.Manna and R.Waldinger, "A Deductive Approach to Program Synthesis", *Readings In Artificial Intelligence and Software Engineering*, C.Rich and R.C.Waters, ed. Morgan Kaufmann Publishers, 1986.

[42] J.Martin and C.McClure, *Software Maintenance,* Prentice-Hall, New Jersey, 1983.

[43] Y.Matsumoto, "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels", *IEEE Trans. on Software Eng.,* Vol. SE-10, No. 5, 1984.

[44] R. Milner, "A Theorem of Type Polymorphism in Programming", *Journal of Computer and System Sciences,* Vol. 17, 1978.

[45] *Miranda System Manual*, Research Software Limited, 1989.

[46] J.P.Morrison, "Data stream linkage mechanism," *IBM Syst. J.*, vol. 17, no. 4, pp.383-408,1978.

[47] J.H.Morrissey and L.S.Wu, "Software engineering - An Economic perspective," *Proc. 4th Conf. Software Eng.*, New York: IEEE, 1979,pp.412-422.

[48] J.M.Neighbors, "The Draco Approach to Constructing Software from Reusable Components", *IEEE Trans. Software Eng.,* Vol. SE-10, No.5, September 1985.

[49] G.C.Oddy, C.J.Tully *Information Systems Factory Study,* Final Report, Volume 1, Department of Trade and Industry, 1988.

[50] H.Partsch and R.Steinbrüggen, "Program Transformation Systems", *ACM Computing Surveys,* Vol.15,No.3,Sept. 1983.

[51] L.C. Paulson. *Logic and Computation — Interactive proof with Cambridge LCF*, Cambridge University Press, 1987.

[52] M.Rittri, "Using Types as Search Keys in Function Libraries", *Proc. Conf. Functional Programming Languages and Computer Architecture*, ACM Press, London, Sept. 1989.

[53] G.A. Robinson, and L. Wos, "Paramodulation and Theorem-Proving in First Order Theories with Equality." In D.Michie (Ed.),*Machine intelligence 4*. Edinburgh University Press.

[54] J.A., Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *J.ACM,* Vol12, pp 23-41.

[55] C.Runciman, I.Toyn, "Retrieving re-usable software components by polymorphic type", *Proc. Conf. Functional Programming Languages and Computer Architecture*, ACM Press, London, Sept. 1989.

[56] D.R.Smith, G.B.Kotik and S.J.Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute", *IEEE Trans. Software Eng.,* Vol. SE-11, No. 11. November 1985.

[57] H.M.Sneed, G.Jandrasics, "Formal Transformations and Methods", in *Proc. Conf. on Software Maintenance-1988*, pp.126-131, IEEE Computer Society Press, 1988.

[58] E.B.Swanson, "The Dimension of Maintenance", in *Proc. 2nd Int. Conf. on Software Eng.,* pp. 492-497, October 1976.

[59] D.A. Turner, "Miranda: A non-strict functional language with polymorphic types", in *Functional Programming Languages and Computer Architecture, Nancy,* Springer Lecture Notes in Computer Science, vol 201, September 1985.

[60] M.Wood and I.Sommerville, "A Knowledge-based Software Components Catalogue", *Proc. Conf. Software Eng. Environments,* University of Keele 1987.

[61] L. Wos, G.A. Robinson, and D.F. Carson , "Efficiency and completeness of the set of support strategy in theorem proving." *J.ACM*, Vol 12, pp 536-541, 1965.

[62] S.S.Yau and J.S.Collofello, "Some Stability Measures for Software Maintenance", *IEEE Trans. Software Eng.,* Vol SE-6,No.6,November 1980.

[63] M.V.Zelkowitz, “Perspectives on Software Engineering”, *ACM Computing Surveys,* Vol. 10,No.2, June 1978.