Functional Metaprogramming

Gareth Powell

A thesis submitted to the
University of Bristol in accordance
with the requirements for the degree
of PhD in the Faculty of Engineering,
Department of Computer Science

August 1991
Revised July 1993

Abstract

Functional metaprogramming consists of the study of functional languages and individual functional programs. A number of areas of functional metaprogramming have been investigated.

A design is presented for a functional program development system, which is intuitive for functional programmers and which may be easily customized and extended. This design is extended to incorporate an assistant for the proof of program properties.

A proof system is introducted, providing flexible mechanisms within which proofs may be constructed in a fashion designed to make proof accessible and attractive to functional programmers. The proof is constructed in a fashion similar to that used for ordinary programs. The presentation of the proof rules resembles function applications, and the statements are written in the underlying functional language and interpreted appropriately.

In order to be able to make precise statements of program properties, extensions are provided to the standard types system to enable subtypes to be easily represented. Additionally, an algorithm is presented which enables suitably annotated defining equations of functions to be treated as totally independent equations.

For Ben, Jo,
and Maggy,
who helped
when I needed
it most;
my mother,
who made it
possible;

and my father
who would have
loved to
see it.

No portion of the work referred to in this thesis has been submitted in support of an application for another degree of qualification of this or any other university or institution of learning.

Gareth Powell

Contents

List of Figures

Chapter 1

Introduction

`Functional Metaprogramming` consists of the study of functional languages and the manipulation and proof of individual functional programs. I have investigated a number of areas in Functional Metaprogramming with the intent of constructing the design for an environment in which to develop functional programs. An important part of this design is a tool to allow entry of statements about a program under development, and to assist in proving these statements.

Within this environment it should be possible to take the initial design for a functional program, and to derive properties from it. The design for the program may then be implemented using a suitable functional language. This implementation may then be compiled, executed and debugged within the environment. The derived properties may be expressed as boolean expressions within the functional language, and the proof tool used to deduce that these expressions will be true under all circumstances. The environment should provide additional facilities, for example, to manage program development.

Proof is not easy for most people, and this environment assumes that proof will be carried out by functional programmers. They are expected to be familiar with the programs they write, and to be able to derive suitable properties for proof, but not necessarily to be able to construct proofs of these by hand. This restricts the choice of available proof methods. The proof system must be rigorous, but it must be intuitive to functional programmers. The proof assistant must at least handle the details of the proof, and prevent the programmer from making fundamental mistakes. If it can actively help in developing the proofs, for example, by suggesting possible ways in which the proof may be advanced, the process of developing proofs of properties may become substantially quicker.

An important consideration with this environment is the user interface. The environment consists of a number of disparate tools. These must interface cleanly with each other and with the user if the environment is to be used effectively. While the design for the overt user interface is fairly standard, its internal design introduces a number of new ideas.

This thesis presents a design for the underlying environment. The environment server or coordinator is presented in some detail, as is the method in which it is connected to the other tools constituting the environment. A type system which gives much finer graining of values than is available with the standard type inference algorithm is presented, along with means of deducing automatically the range of values over which the individual equations making up a function

apply. A proof system for use by programmers as part of the environment is presented, as is an assistant which forms one of the tools of the environment. No other tools are presented in more than outline, so that it may be seen how they fit into the environment.

The remainder of this chapter provides a brief introduction to many of the core ideas and terms included in this thesis.

1.1 Functional Programming

`Functional Programming` differs substantially from more conventional, procedural, programming. A functional program defines its effects entirely in terms of a single expression which is to be evaluated. Although the form of the expression changes during evaluation the current form at any point is always equivalent to the original in the sense that it has the same effect. The effect of the expression is called the value of the expression. No order of evaluation is made explicit within the program. Instead a set of evaluation rules are repeatedly applied to the expression to be evaluated and its subexpressions until the program terminates. This occurs when the form of the expression can no longer be changed by applying the functions defined in the program.

A `function` in a functional language defines possible replacements for expressions. A `defining equation` for a function consists of two parts: a left hand and right hand side. If any portion of the expression to be evaluated matches the left hand side of the equation, then this may be replaced by the right hand side of the equation.

Functional languages may be divided into several classes. `Pure` functional languages hold exactly to the description given above, in contrast to `hybrid` functional languages, which augment the above with some procedural constructs (for example, variable assignment). `Strict` functional languages require that once evaluation has begun on a sub-expression it must continue until that sub-expression has been fully evaluated, whereas `non-strict` or `lazy` functional languages use a set of evaluation rules designed in such a way that a subexpression is never evaluated unless its value is required, and that evaluation on a sub-expression only continues until the minimum necessary information about it is obtained. Theoretically, any particular functional language may be any combination of these two categories, but it is difficult for a language to be both strict and pure.

1.2 Miranda

`Miranda`[1] is a proprietary, pure, non-strict functional language. In addition to a collection of built in functions and types, facilities are provided for defining new functions and types, and collecting these together into modules, which are called "scripts". An environment for entering expressions to be evaluated interactively is provided, which also provides access to an editor to create and modify scripts. Expressions and definitions are entered using a simple syntax, and special syntax is provided for common constructions, for example numbers and lists. A number

[1] Miranda is a trademark of Research Software Limited

of advanced features (for example, modules and list comprehensions) are also provided.

1.3 Haskell

A number of pure functional languages have been developed and are currently available. These differ primarily in syntax, and `Haskell` is a language designed to embody the consensus of ideas across these languages. The Haskell report defines the Haskell language, and at least two implementations are being developed.

The idea of Haskell is to provide a uniform, non-proprietary language for teaching and research. New ideas can be built into the basic structure of Haskell, and the results presented in a fashion which is then more easily understood across the research community.

However, the existing functional languages differ in a number of areas more significant than the syntactic differences already mentioned. The means for providing advanced features, for example, program modules, differ widely amongst languages. For these features Haskell provides innovative solutions, including a form of arrays for constant time lookup, and a class system to augment the types of the language.

1.4 Flair

`Flair` is a very simple, pure, non-strict, functional language. It is essentially a subset of Haskell, removing all of the complex features. It has a simple block structure, allows the definition of a single final expression together with function and type definitions. Function definitions are similar to Haskell. Scoping is handled very simply. Special syntax is provided for numbers, characters, and strings. The essence of Flair is that it can be easily comprehended and manipulated, since there are very few special cases. A number of features are included in Flair which are not in Haskell to support meta-programming. A simple compiler for Flair is presented in the appendix.

The ideas presented here are all presented in terms of Flair, although they could be equally applied to any other pure, non-strict functional language (such as Miranda or Haskell) provided that any algorithms or techniques demonstrated here are extended to cover all the features of the language, and provided that the language is extended to support any extra features required.

1.5 Types and Classes

Any expression in a functional language has a certain value, which can be determined by evaluation. Different values with similar properties can be grouped together. Most functional languages permit the grouping of values into `types`. The means by which this may be accomplished, and the properties of the resulting types varies from language to language. In Miranda and Haskell values belong to exactly one type. In Flair, an extended system of types allows the intersection and union of types to produce new types. A value may belong to any

number of types (or none of them). This allows greater expressiveness, which is particularly important in the proof system.

Haskell further groups types together into classes. These classes share properties across all member types. For example all members of the `Eq` class have an equality operation defined on them. These classes are not supported in Flair, although an alternative formulation for a class-style idea is presented, using an extension to the standard idea of polymorphism.

1.6 The Toolbench

In addition to an underlying functional language, a suite of tools is required, which form an environment in which programs may be produced and executed. The most important of these tools are an editor with which to create programs, a functional language compiler to produce executable versions of programs, and some means (for example an interpreter) for executing the resulting programs. The editor is the centre of this activity, since it is necessary to use the editor to produce the original program, and subsequently to modify it (either to remove initial errors or to produce a different behaviour).

A wide variety of tools may be desired beyond this however. For example, specialised tools to produce documentation, to maintain source code archives, to assist in debugging programs, and for meta-programming. It is therefore important that a toolbench be developed, providing the essential tools, but to which any new tools can be added without excessive difficulty. The toolbench should provide standard means of providing services (such as the user interface) and the individual tools should interact with the toolbench in a consistent and well-defined manner.

1.7 Environment Design

The result of my investigations is the design for an environment for workstations such as the Sun/3 and Sun/4 families, which has a single, consistent, configurable user interface. Within this environment a programmer may develop a functional program, entering, compiling and executing it under the control of an editor. The design of the environment makes it easy to extend the environment by adding additional tools. The design also provides for a tool which enables the programmer to state and interactively prove properties about programs.

The only interface between the programmer and the environment is through a coordinator. Multiple on-screen windows are associated with the current activities, such as editing programs or proving properties. Communications between the tools and the coordinator provide the only interface between the tools and the programmer. There are two basic types of window used for interaction between the programmer and the environment: browsers allow a selection to be made, whereas edit windows allow the creation, modification, and observation of text files.

The `coordinator` provides the programmer with access to all the tools in the environment, and separate tools in the environment communicate through the coordinator. This arrangement is illustrated in Figure 1.1. The coordinator's primary role is to organise the programmer's use of these tools. Each of the

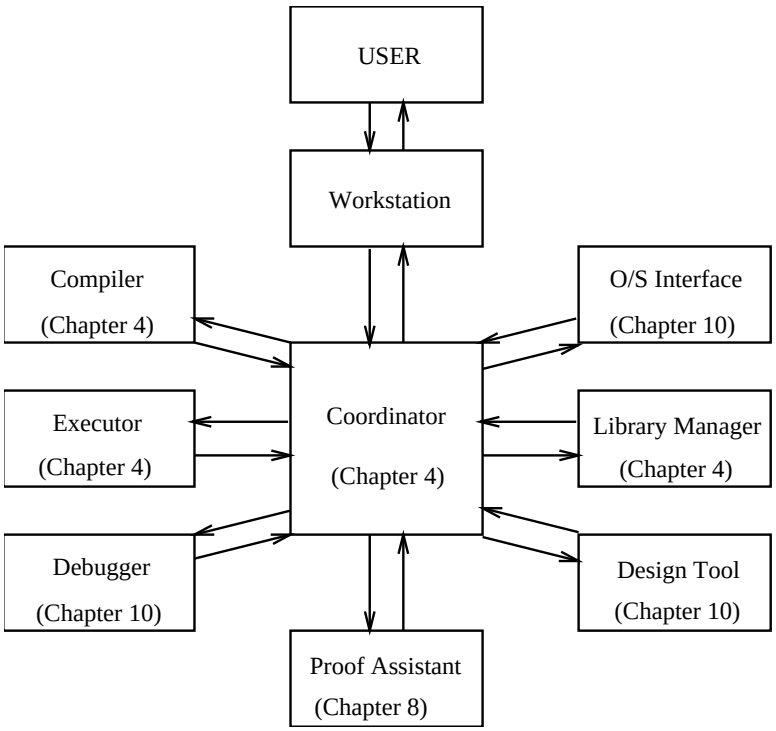

Figure 1.1: Arrangement of tools within the environment

tools has one precise task to fulfill.

The `compiler` accepts programs (or parts of programs) written in the underlying functional language and generates executable versions. This code is common across the environment, and may be executed directly by the executor, or internally by the editor and proof assistant. Errors found during the compilation are reported to the programmer through special windows opened by the coordinator. These may then be corrected and the modified program submitted for compilation. Ideally, an incremental compiler would be used, which enables a much faster recompilation time when errors are found.

The `executor` accepts executable code (which will have been generated by the compiler at some point) and executes it. Such code may take advantage of the connection with the coordinator to provide a window-based user interface. Otherwise, a standard editor window is opened to conduct an interactive session. The `debugger` provides an opportunity to execute programs in a controlled environment, allowing the program to examine the results of various expressions, and to control the evaluation process. No description of the debugger or how it might work is presented here, although a brief description of how a debugger might be used in the environment, together with references on previous work on functional language debugging is given in Chapter 10.

The `proof assistant` supports the statement and proof of program properties. It accepts portions of code and proof from the coordinator in their standard form and stores these internally. It then permits the programmer to review these

in a variety of ways which are more intuitive than the standard form. A proof, once constructed, may be automatically checked to ensure that the property has indeed been proved, and it is possible to store results, once proved, as a foundation for constructing and checking other proofs. While a proof is under construction, the assistant may help by suggesting possible ways in which the proof may proceed, for example, by suggesting a rule or expression which seems most applicable at the current stage.

The `library manager` provides an interface between the coordinator (and hence the rest of the system) and the filestore. In the simplest case (which is assumed here) this simply means that all the reading and writing of files is done centrally by one tool. In a more advanced case, the manager should support multiple users, with file-locking and update control, a revision or version control system, directory facilities (for example, accessing files by the functions they provide), and a means to combine files of various types (for example documentation along with code).

The tools described above provide all the facilities needed for a good development system. For a deliverable system for ordinary users, an extended executor tool is required to provide an easy interface to the programs available within the environment. This tool would provide an interface to the operating system analogous within this environment to a conventional operating system shell although the facilities provided, and particularly the interface design might be substantially different. Most of such a tool already exists within the system already described: the library manager provides the appropriate access to both programs and the filestore; the coordinator provides the necessary user interface components. This tool could also be of use to developers producing programs, as a replacement for the standard operating system shell. No such tool is described here.

A single, standard form of communication exists between the various tools. Each tool communicates directly only with the coordinator; communication with other tools is indirect, via the coordinator. Within the coordinator a functional language interpreter coordinates the environment activities through a requests/responses style input/output mechanism. This is translated by another part of the coordinator, and messages to other tools are encoded as operating system-specific messages (for example, over streams or as remote procedure calls). These messages are then accepted and decoded by the various tools as required, which generate appropriate response messages which are in turn encoded and decoded across the environment.

A tool may respond to a request with another request — for example, a request to compile a program may produce a request for access to the source of an included file: this will elicit a response from the coordinator, and eventually a response from the compiler to the coordinator's original request will be forthcoming. All user input is decoded by part of the coordinator, and passed to the functional interpreter directly.

The system may be easily extended by adding new tools to provide new services. The coordinator is customised using a functional programming language. The programs customizing the coordinator must be extended or altered to provide an interface to the new tool. An example of this is presented when the proof assistant is added to the system in Chapter 8.

1.8 Proof

When developing programs, the programmer is expected to prove that these programs meet certain requirements — in general, that they conform to a user specification. These properties are expressed as combinations of functions within the system, and will be boolean valued. The programmer must show that these properties will always hold, and it is important that such proofs are completely formal. In order for this to be practical, the programmer must be given automated assistance. This is provided by a tool known as a proof assistant, which is responsible for rigorously checking the proofs provided by the programmer, and assisting in the construction of the proofs. As a further aid to the programmer constructing the proofs, it is important that the basis of the proof system be intuitive.

Several styles of proof have been investigated in the course of this work, and an overview of these styles is presented. One style has been developed into a completely detailed, rigorous proof language, and the issues of how proofs may be automatically checked in this style have been addressed. Possible means of machine assistance in the construction of the proof are outlined, although detailed mechanisms for assistance in proof construction are outside the scope of this work.

The proof assistant is, of course, integrated into the environment described in the previous section, and this provides an extensible proof assistant: that is, the proof assistant may be programmed in a relatively simple manner by the programmer to provide assistance in any way that is required. This enables any programmer using this development system to customize the proof assistant with any heuristics or algorithms which are found to be useful following previous experience of developing proofs.

1.9 Layout

Chapter 2 provides detailed background on a functional language, its syntax, features, and denotational semantics, while Chapter 3 provides background information on operational semantics, and implementation information for a functional language.

Chapter 4 presents, in detail, a design for the extensible functional environment described above, concentrating on the mechanisms for communicating between the tools and a means of coordinating the various user activities.

Chapter 5 presents an extension to the type system which makes it more suitable for meta-programming, supplying additional features to commonly accepted type systems, while still providing the standard features.

Chapter 6 discusses various styles of proof which could be used with functional languages in an attempt to develop a style which is well suited to functional programmers, and fits well with the design of the system.

Chapter 7 describes a proof system for use with functional languages. It provides means for expressing properties within the functional language, and then for proving that they are correct.

Chapter 8 shows how a proof assistant to support proofs as detailed in Chapter 7 may be integrated into the environment of Chapter 4.

Chapter 9 discusses the proof system in Chapter 7 and shows, by comparison to classical or constructive logic, that proofs presented in this system are valid.

Chapter 2

Functional Languages

This research is directed towards the design of an integrated environment for developing functional programs and proving properties of these programs. This chapter discusses the functional language Flair in detail, ending with a discussion of its formal semantics.

The functional language Flair is essentially a subset of Haskell, the emerging, non-proprietary standard for pure non-strict functional languages. Many complex features, such as infix operators and constructors, list comprehensions, and arrays are excluded from Flair. The type system is considerably extended, in order to better express the type properties of programs, although this is not discussed until Chapter 5. The type and class systems of Haskell are ignored for the present.

A Flair program is logically divided into three sections. These may all coexist within a single file, although generally they will be kept physically distinct. The first section consists of standard data and function definitions, which define the object program. The second section provides the extended type definitions which may be required for proofs. The third section contains the proofs of program properties. The statements of program properties, where used, may be placed in either the first or third sections. A notional fourth section contains meta-level definitions required for the proofs. This is kept totally separate from all other parts of a Flair program.

2.1 Values and Patterns

Values are covered formally in a number of papers, for example [70]. Informally, each data value is either a constant or is formed by combining one or more values using a constructor. A constructor may be introduced into a Flair program, as in Haskell, by defining a type which uses the constructor. A constant pattern consists of the constructor written on its own. The type definition is introduced by the keyword data followed by the name of the type being defined. The data type `zero`, containing only the constructor `Zero`, apart from the non-terminating value $\perp$, might be defined:

```
data zero = Zero
```

Non-constant patterns, which describe sets of values, are written as a constructor followed by one or more type names. The number of type names is called the arity of the constructor (by extension, constant constructors are said to have arity zero). For example, the type `list` consists of the empty list, represented by a constant constructor, `Nil`, and a pair, combined using the constructor `Cons`, of a first element and a `list` of remaining elements. Multiple patterns can be used in the same type definition by separating the patterns with a bar `|` indicating a union of the sets of values.

```
data list = Nil | Cons element list
```

Two restrictions are placed on any constructor introduced in this way: it must be written with an initial upper case letter to distinguish it from other syntactic objects; and it may not appear in more than one pattern.

In addition to defining the names of constructors and their arities, the above declarations also define functions which allow values to be formed by combining smaller values; restrictions upon which smaller values may be used in forming such values; and the type to which the new value will belong.

In the above example, `Nil` defines a constructor function which takes no arguments, and which produces a single value of type `list`; `Cons` defines a function of two arguments: the first is required to be a value of type `element`, while the second must be of type `list`: the result will be a value of type `list`. The type `list` consists of all the values which may be defined in these ways, together with partial and infinite lists, generated by the implicit inclusion of the value $\perp$ in the type.

A constant constructor directly represents a single value. A non-constant constructor defines a pattern for a set of values. The value is constructed by applying a constructor to its correct number of arguments, that is, equal to its arity. These values correspond to ground terms in logic programming languages, and the set of all values in all types is approximately equivalent to the Herbrand Universe. The Herbrand Universe consists of all the ground terms which may be formed by the application of any of the function symbols to a number of ground terms corresponding to the arity of the function symbol. However, expositions of the Herbrand Universe do not usually deal with the non-terminating value $\perp$ or consider the types of the ground terms.

Polymorphic type variables may be used to define more general types. Instead of defining each type name used as an argument in the definition of a pattern, a variable may be used, which is replaced by a "real" type name when the type is used. For example, the type element in the above example may be replaced by a polymorphic variable `k`, and the type `list element` may be used in place of the original `list`.

```
data list k = Nil | Cons k (list k)
type listel == list element
```

Whenever an instance of this type is used, for example, in the definition of `listels`, the desired type is substituted for `k` wherever it appears on the right hand side of the definition of `list`. Any number of different instantiations of the variable `k` may appear within a single functional program, each generating a different, but nevertheless related, type definition.

In addition to the simple patterns described above for defining types, consisting of a constructor and zero or more typenames, patterns are also used in a more general way in equations for defining functions. These consist of a constructor, together with arguments corresponding to the arity of the constructor, each of which is either a free variable or a pattern. For example,

```
Cons x (Cons y r)
```

is a general pattern of type `list`, describing any list with at least two elements, `x` and `y`, and the remaining list `r`. This is illustrated in Figure 2.1.

A value may be matched against a pattern. If the match is successful, then the free variables in the pattern are instantiated to the corresponding parts of the value. In order for the match to be successful, the pattern's constructor must be the same as the value's, and, whenever a pattern appears as an argument inside the pattern, it must match the corresponding argument in the value.

For example, the value `Nil` does not match the above pattern, since it has a different constructor. The value `Cons Zero Nil` also fails to match, since the second argument in the above pattern is a pattern, and the pattern `Cons y r` does not match the value `Nil`. The value `Cons Zero (Cons Zero Nil)` does match the above pattern, and gives rise to the instantiations `x`/`Zero`, `y`/`Zero`, `r`/`Nil`. These matches are illustrated in Figure 2.2.

Two types which will be frequently used are `nat` and `tree`. These are defined using the following `data` definitions.

```
data nat  = Zero  | Succ nat
data tree = Empty | Tree tree nat tree
```

2.2 Types as Sets of Values

Each type is defined as one or more simple patterns. A simple pattern is a pattern consisting of just a single constructor applied to an appropriate number of type arguments. Each simple pattern describes a (possibly empty) set of values. Consequently each type represents a set of values, which is the union of the sets described by the patterns.

The set of values described by a type is fairly easy to determine. It consists of all the values which match one or more patterns in the type definition, and whose arguments belong to the types specified in the patterns. The value $\perp$, representing the value of an expression which either does not terminate, or results in an error, is included as a member of every type.

If types are viewed as sets of values, then sub-types, overlapping types, and union types become easy to describe; representing them adequately is somewhat more subtle. This is addressed in Chapter 5.

2.3 Functions and Expressions

Data values represent data objects in a functional language, and are the final results of all computation. Types and patterns provide means to group these values into sets. The process of computation may be described by the introduction of expressions and functions.

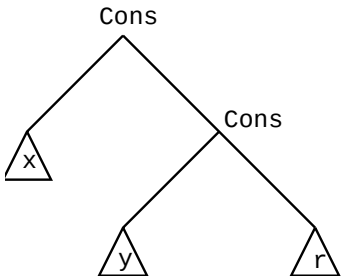

Figure 2.1: The pattern `Cons x (Cons y r)`

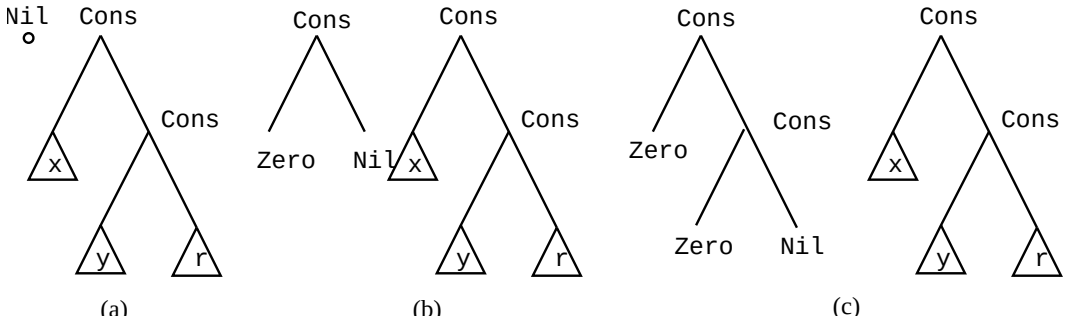

Figure 2.2: Matching against the pattern `Cons x (Cons y r)`

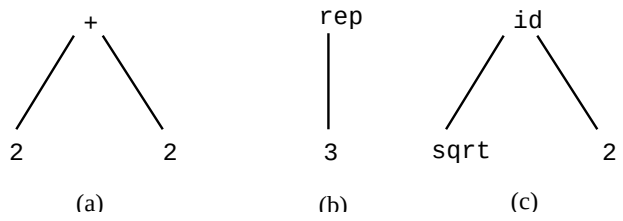

Figure 2.3: Correctly, under- and over-applied functions

Expressions are tree shaped objects similar to values. An expression consists of either a constructor applied to zero or more arguments or a function applied to zero or more arguments: each argument is an expression. It is not a requirement that a function or constructor be applied to a specific number of argument expressions, as constructors must in forming data values, although functions and constructors still have arities — the number of arguments they expect to be applied to. If applied to this number, the result will generally be a regular, non-function value: for example, in Figure 2.3(a), `+` is applied to two arguments, its arity, and the result is an integer value, `4`. If applied to less arguments than the arity the result is a function value with arity equal to the number of arguments still required: Figure 2.3(b) shows the `rep` function, which takes two arguments, just applied to its first, numeric, argument. The result is a function of one argument which produces a list of three copies of this argument. It is also possible for a function to be applied to more arguments than its arity: for example, Figure 2.3(c) shows the `id` function, which has arity one, applied to two arguments. In this case, the function is applied to the first arity arguments, the result being a function which is to be applied to the remaining arguments.

Functions are defined using equations. The left hand side consists of the function applied to general patterns and the right hand side consists of an expression involving some or all of the free variables used in the patterns. Any number of equations may be used to define a function, and it must always be possible to identify which equation is intended for use with any given set of argument values. Two methods may be used to determine which equation of a definition is to be used: firstly, an equation may only be used if the patterns on the left hand side of the definition match the values used as arguments to the function application. Secondly, a guard may be used to determine if the case applies, the guard being a boolean-valued expressing usually involving one or more of the free variables in the patterns. Guards are provided in Haskell, but not in Flair (instead, the programmer must use a function `if`, which takes a boolean, and the two alternative values). In the simplest case, the patterns are distinct, and no guards are used. Consequently, there is no combination of values for which more than one equation applies. For example, a simple function to reverse a list would have two equations. The first covers the case for an empty list, while the second covers the case for non-empty lists.

```
reverse Nil = Nil
reverse (Cons e res) = append (reverse res) e
```

In more complex cases, more than one equation may appear to apply to a given set of arguments. In Haskell, such problems are resolved by the order in which the equations are written; alternatively, the cases may be dealt with simplest first, that is, the case which matches the smallest set of values, and either the equations must be distinct or there must be a simplest case to choose — if not, the equations are deemed to be inconsistent. For example, in the following definition of another simple list processing function, `drop`, the expression `drop 0 Nil` matches both of the first two left hand sides.

```
drop 0 es = es
drop k Nil = Nil
drop k (Cons e res) = drop (k-1) res
```

In Haskell, the first equation is used. If the simplest case is being chosen, these equivalences are deemed inconsistent, and a simplest case must be added:

```
drop 0 Nil = Nil
```

Note that if this is added as the first equation, it does not alter the actions of Haskell in interpreting this definition.

Chapter 3

Interpretation of Functional Languages

Functional languages, as described in the last chapter, may be compiled into a sequential form, which may then be executed on any sequential computer. The evaluation of functional programs is used as a model upon which proofs are based in Chapters 6 and 7, and a suitable intermediate code, called G-Code, is described in Section 3.1. `G-Code` was originated as a simple means of representing sequentially executable functional programs by my supervisor, Dr. Ian Holyer, based upon the Spineless, Tagless G-machine [63], and a linearised version of their `T-Code`. A method for translating expressions from Flair and compiling pattern matching into G-code is discussed in Section 3.2, which contains some original work on an algorithm to separate overlapping cases in defining equations.

3.1 Evaluation

Evaluation of functional programs may be viewed as the sequential execution of a series of low-level instructions, provided that the functional program is first translated into a suitable form. One such form is described here. It is called `G-code`, because it is based on the G-machine approach to evaluation of functional programs, particularly the Spineless, Tagless G-machine. G-code consists of a few instructions, and essentially one addressing mode. It may be subsequently translated into assembler for most microprocessors or RISC processors, or may be interpreted by a relatively small C program. Such a C program is given in Appendix E. The form of reduction described here is tree reduction, although the method may be easily extended to cover graph reduction.

3.1.1 The Evaluation Model

The process of evaluating a functional expression involves repeatedly rewriting the expression in such a way as to reduce it to a normal form, which is a value constructed as described in the previous chapter. Data for the functional program is stored in a single memory space, comprising two portions: the `heap`

is used to store structured data objects, such as expressions and values in a random-access fashion, while the `stack` is used to store pointers into the heap in a last-in, first-out fashion. Each expression may be represented as a tree, each node consisting of a function applied to arguments as pictured in Figure 3.1. This may be translated into a form suitable for storing in the heap/stack model as shown in Figure 3.2. Code may be stored as a collection of blocks, as in Figure 3.3. Evaluating a function application is performed by creating a new heap cell representing the right hand side of an expression from the arguments to the function presented on the left hand side. For example, consider a simple definition of `reverse` for a non-empty list (pattern matching is addressed later).

```
reverse (Cons x l) = append (reverse l) x
```

When `reverse` is entered, one argument is on the stack, pointing to a heap cell containing a `Cons` node of two arguments: the head and the tail of the list. Both of these are possibly unevaluated expressions. The code for `reverse` removes the pointer from the stack and generates a new heap cell containing a suspension node with two pointers: the first points to the function reverse, while the second is the pointer to the tail of the list. The pointer to this new cell, together with the pointer to the head of the list, are then placed on the stack. After rearranging the stack in this way, `reverse` transfers control of execution to the function `append`.

The node to evaluate `reverse l` is called a suspension node, because evaluation of that node is suspended until it is required. The first word of a suspension node points to a code block to be executed, and the remaining words point to variables free in the sub-expression. The only time that a suspension is evaluated is when the head constructor needs to be determined for pattern matching. When the suspension node is entered, the code pointed to by the first word is executed, and this may retrieve and use the other pointers in the suspension node as data. For `reverse l`, the code for the suspension simply recovers the free variable, `l`, from the suspension node, pushes it onto the stack, and calls `reverse`.

The stack is arranged in frames, each frame consisting of a continuation address, and the variables stored by the caller for use by the code at the continuation address. In this way, each frame on the stack resembles a suspension node on the heap: it consists of a code pointer and some free variables. The difference is that the suspension nodes represent sub-expressions which are awaiting evaluation; the stack frames represent super-expressions waiting for the evaluation of sub-expressions to complete before continuing. At the top of the stack, arguments to the function to be called are stored.

The heap grows continually as new nodes, representing new intermediate expressions, are created. When old intermediate expressions are no longer required, the old heap nodes become inaccessible, but are not immediately reused. The stack alternately grows and shrinks: when a sub-expression needs to be evaluated, an extra frame is placed on the stack, when that has finished, the frame is removed from the stack. The evaluation of the sub-expression may, of course, require the evaluation of other sub-expressions, so causing multiple levels of stack to build up. The code segment is of a fixed size, totally determined at compile time.

If the heap and stack fill up the available memory space, it is necessary to reclaim unused space. There is never any unused space inside the stack (it is

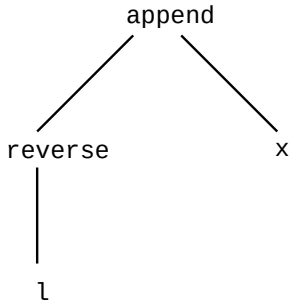

Figure 3.1: `append (reverse l) x` as a tree

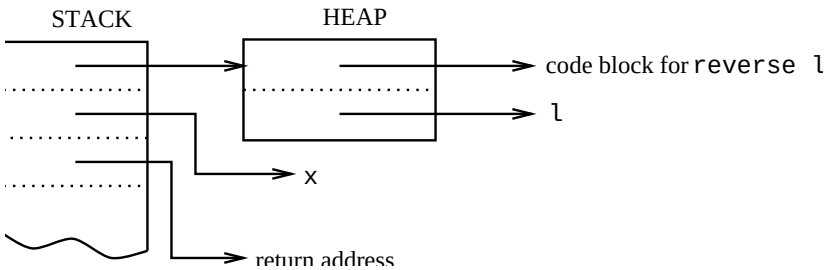

Figure 3.2: Stack and heap prior to `append`, during `append (reverse l) x`

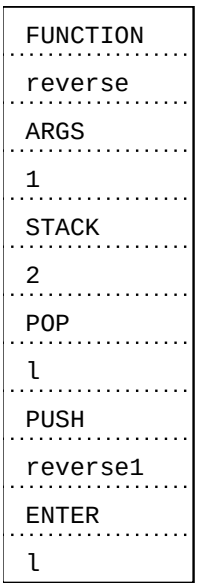

Figure 3.3: Code block for `reverse`

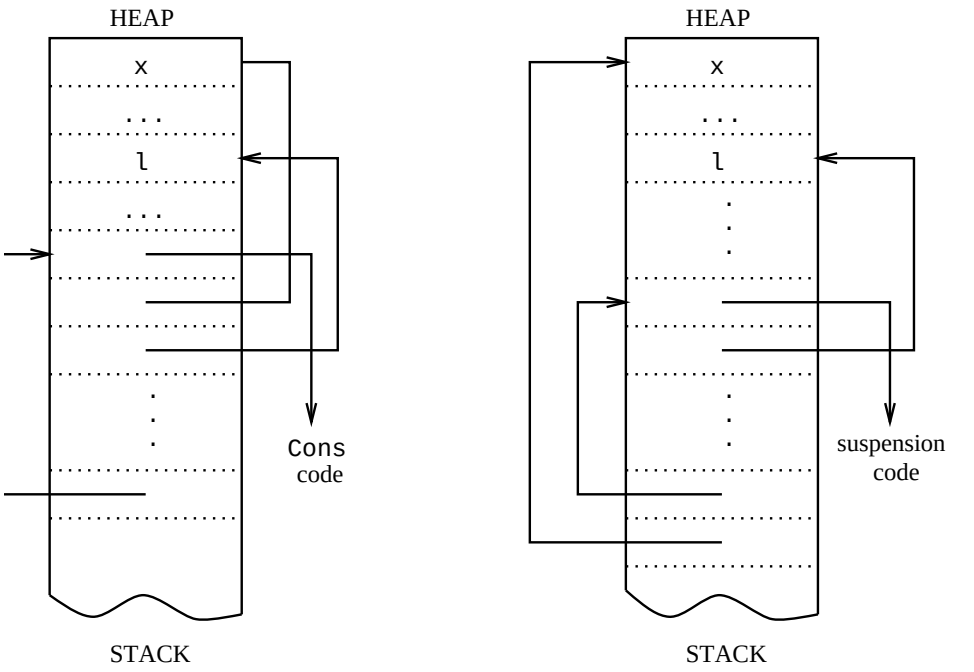

Figure 3.4: Evaluation of `reverse (Cons x l) = append (reverse l) x`

freed when the stack shrinks), but the heap may contain many dead nodes from intermediate representations of expressions. These may be collected and the heap reorganised in such a way that there are no holes in the heap, and the heap may once more start growing. This process is called garbage collection, and is not discussed here further, but see [41].

In addition to the pointers necessary for maintaining the current heap, stack and code positions, there are three special registers in G-Code. The `NODE` register holds a pointer to the current node: when evaluating a suspended sub-expression, this points to the suspension node; when evaluating an expression already in head-normal form, it points to the node containing the expression. The `TAG` register holds a small integer which is used to identify the head constructor of an expression once it has been evaluated. The `ACC` register is a numeric accumulator used for arithmetic operations.

3.1.2 G-Code

Code is presented in basic blocks. Each block consists of a heading line, indicating the kind of block and naming an entry point into the block, followed by data manipulation instructions, and ending with an instruction to transfer control to another basic block.

Each block may be of one of four types. A `FUNCTION` block indicates the entry point for a function defined in the source code. If it is necessary to evaluate a sub-expression (generally when performing pattern matching) a block ends by pushing a continuation entry point, and transferring control to the sub-expression — later, control returns to the previously pushed entry point, which is a `CONTINUE` block. When pattern selections are to be made, it may be necessary to choose different basic blocks depending on the constructor matched. In this case there is a different sort of block ending, with more than one transfer instruction, all but the last conditional. The destination blocks of conditional jumps are `CASE` blocks. Suspensions and constructors are represented by `LABEL` blocks.

On entry, each basic block must perform certain checks. The checks that need to be performed depend upon the type of block. A `FUNCTION` block must check that it has been given sufficient arguments. If not, the result of the function will be a function value: a partial application of the function applied to the arguments provided. `FUNCTION` and `LABEL` blocks must check that there is enough stack space available for their use, and for all the blocks associated with them (`CONTINUE` and `CASE` blocks). `FUNCTION`, `LABEL`, and `CONTINUE` blocks must check that there is sufficient heap space for their use, and for any `CASE` blocks associated with them.

Figure 3.5 shows the first basic block associated with the function `reverse`: it includes a heading line, and the two required checks (there is no heap check, since no heap space is used). The argument is recovered, and after storing the continuation address on the stack, the argument is evaluated.

There are two stack manipulation functions: `PUSH` places its argument on the stack, decrementing the stack pointer, while `POP` takes an item off the stack, placing it in its argument, and incrementing the stack pointer. The stack is generally used for passing arguments to functions, and storing temporary results and return addresses across function calls.

The second basic block, shown in Figure 3.6, illustrates the continuation block executed after completion of the evaluation of the argument to `reverse`. It first checks that sufficient heap is available (to be used in the fourth block), then removes the continuation from the stack, then examines each expected case in turn, and ends with a catchall — there is a missing case in the definition. This missing case check handles two situations: firstly, when a function has been defined by cases, and not all cases have been covered by the equations; and secondly, for generality, the interpreter assumes an untyped language, with a flat constructor space, and any mistyped function applications will cause this error to be generated.

Four instructions are provided for control transfer: `JUMP` takes the name of a basic block entry point and uses this as the next basic block to be executed. All other special registers remain unchanged. `ENTER` takes as its argument a node in the heap. It uses this to set the special register `NODE`, and then transfers control to the basic block pointed to by the first cell of the new node. `RETURN` sets the special register `TAG` to the value of its argument, and then transfers control to the basic block pointed to by the value at the top of the stack. `SWITCH` tests the register `TAG`, and if it is zero, transfers control to the entry point given as argument, otherwise `TAG` is decremented by one. The instructions `TRY` and `UNTRY` are provided to enable easy comparisons between the value of `TAG` and specified constructors.

There are six heap manipulation functions: `NEW` takes the first free element on the heap and places its address in the argument to `NEW`. Locations on the heap are allocated in sequential order with no breaks. `FILL` places its argument in the first free location on the heap, advancing the first free location pointer. `STORENODE` saves the current value of the special register `NODE` in its argument; `SETNODE` reverses the process by setting `NODE` equal to its argument. `PUT` places its argument in the location pointed to by `NODE`, and advances `NODE`; `GET` recovers the contents of the location pointed to by `NODE`, and advances `NODE`.

The third and fourth basic blocks are shown in Figure 3.7 and process the two cases of the definition of the function `reverse`. The block `reverse2` transfers control to the constructor `Nil`. The block `reverse3` recovers the expressions `x` and `l` from the current node, known to be a `Cons` node, then builds and saves the arguments to `append` on the stack in reverse order, and calls `append`. Since one of the arguments to `append` is a sub-expression, a suspension is created for this, using a pointer to code block `reverse4`, and saving the free variable, `l`, of the sub-expression.

The final block, `reverse4`, shown in Figure 3.8, evaluates `reverse l`, where `l` is a free variable recovered from the suspension created in the block `reverse3`. The variable `l` is recovered from the suspension, and pushed onto the stack as an argument to the function `reverse`.

Some special functions, for example, the arithmetic functions `ADD`, `SUBTRACT` and `COMPARE`, are defined as instructions using the special registers `ACC` and `TAG`. Input and output are handled by similar instructions, `READ` and `WRITE`. The instruction `STOP` terminates execution of the program, it's argument giving the exit status of the program.

While G-Code is intended for use as a target language for compiling functional programs, it is not necessarily the case that all programs in G-Code need be functional. Indeed, code has been written to implement updateable arrays in G-Code, using trailers after the style of [37, 42].

```
FUNCTION reverse
    args 1
    stack 2
    pop l
    push reverse1
    enter l
```

Figure 3.5: First block of function `reverse`

```
CONTINUE reverse1
    heap 2
    pop reverse1
    try _tNil
    switch reverse2
    untry _tNil
    try _tCons
    switch reverse3
    untry _tCons
    jump _misscase
```

Figure 3.6: Continuation block of function `reverse`

```
CASE reverse2          CASE reverse3
    jump Nil               get _pCons x l
                           push x
                           new v
                           fill reverse4
                           fill l
                           push v
                           jump append
```

Figure 3.7: The two cases for `reverse`

```
LABEL reverse4
    stack 1
    get reverse4 l
    push l
    jump reverse
```

Figure 3.8: Code for the suspension `reverse l`

3.1.3 Performing pattern matching

Pattern matching is important in functional languages, since it is during pattern matching that expressions are reduced to weak head normal form `WHNF`, that is, may be represented in a single heap cell as a constructor together with (unevaluated) arguments. For example, in the program

```
reverse (Cons 'x' Nil)

reverse Nil = Nil
reverse (Cons x l) = append (reverse l) x

append Nil v = Cons v Nil
append (Cons x l) v = Cons x (append l v)
```

the expression `Cons 'x' Nil` is initially stored on the heap as a suspension — as the application of the function `Cons` to two arguments — or directly as a value — a node consisting of a pointer to some code identifying the node as a non-empty list, together with two pointers, pointing to the two arguments. A pointer to this suspension is passed to `reverse` as its argument. The first action of reverse is to determine which equation should be used: this requires that the argument be sufficiently evaluated to determine the constructor — either `Nil` or `Cons`.

G-Code for pattern matching is very stylised — the same pattern of instructions is repeated for each argument requiring evaluation. The sequence covers parts of several basic blocks: the initial block is completed by saving the free variables in that block, and then evaluating the appropriate argument; a complete `CONTINUE` block examines the register `TAG` and then conditionally jumps to code for the corresponding constructor; one or more `CASE` blocks are started with instructions to recover the variables saved by the initial block, and any arguments in the evaluated heap cell (see Figure 3.9).

The start of the initial block is either the `CASE` block of a previous match, or, if there is no previous match, the function header — the initial entry point, the checks, and the recovery of function arguments. The completion of each of the `CASE` blocks is provided either by the initial block of the next match, or, if there is no next match, by code to evaluate the right hand side of the appropriate (matching) equation. By the time a right hand side is evaluated, all the variables free in the equation will have been recovered by instructions at the start of the `CASE` blocks.

3.2 Compilation

The first stage in compiling a functional program is to parse it and extract all relevant information. Subsequent stages of processing may involve checking, for example, that the function definitions in the program are type correct. It is assumed that all such stages have been completed successfully, and that the only remaining operation is to generate code to be executed in the fashion described above. The original source code is assumed to have been parsed and organised into two collections: a dictionary containing type information; and a list of lists of trees representing all the function definitions, each element in the list being

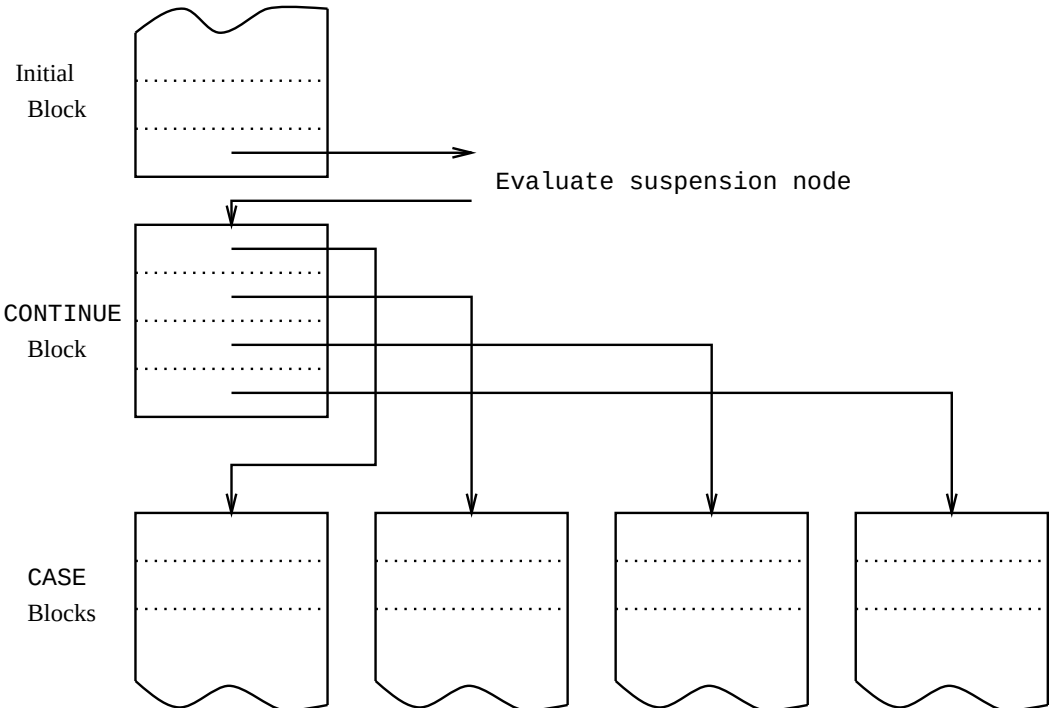

Figure 3.9: Code blocks in pattern matching

one function definition, and each tree representing one defining equation for that function.

Code is first generated for each constructor defined in the program, as discussed below in Section 3.2.1.

For each function, code generation takes place in three stages. The first stage is simple: a function header is output, consisting of the function block declaration, the checks on arguments, heap, and stack, and the recovery of arguments from the stack into temporary variables. The second stage consists of generating pattern matching code to decide which equation applies and to extract the free variables used in that equation: this is described in Section 3.2.3. Finally, code is generated for the right hand side of each equation in turn, which is described in Section 3.2.2.

Appendix C contains a compiler from parsed source code to G-Code, written in Miranda. Appendix B provides a simple parser.

3.2.1 Compilation of Constructors

Three pieces of code are generated for each constructor: a tag is defined as a small integer, distinct for each constructor; a code block to be used once an expression has been reduced to weak head normal form (WHNF); and a constructor function to build an expression heap cell given arguments on the stack. For example, given the two constructors for `lists`, `Nil` and `Cons`, the code shown in Figure 3.10 is generated.

tag :	`EQUATE _tNil 0`	`EQUATE _tCons 1`
whnf block :	`LABEL _pNil` `return _tNil`	`LABEL _pCons` `return _tCons`

constructor function :	`FUNCTION Nil`
	
	`heap 1`
	`new _v`
	`fill _pNil`
	`enter _v`
		`FUNCTION Cons`
		`args 2`
		`heap 3`
		`new _v`
		`fill _pCons`
		`pop _p`
		`fill _p`
		`pop _p`
		`fill _p`
		`enter _v`

Figure 3.10: Code generated for the constructors `Nil` and `Cons`

The code block used once an expression has been reduced to weak head normal form simply returns the tag value associated with that constructor: this will be used to perform a `SWITCH`. The constructor function first checks that enough arguments have been provided (if any are needed), then checks that there is enough heap space to create a new expression heap node. A new node is created, and a pointer inserted in the first word to point to the WHNF block. Then each argument is recovered from the stack and placed in the heap node in turn.

3.2.2 Compilation of Expressions

Each expression on the right hand side of a defining equation for a function consists of a function applied to zero or more arguments. For example, the expression for the recursive case of `reverse` is

```
append (reverse l) x
```

where `l` and `x` are free variables. Expressions are compiled into G-Code recursively. Suspensions are created to represent sub-expressions, and the arguments are pushed on the stack in reverse order. Once this has been done, a transfer instruction is generated to enter the function being applied. `LABEL` code blocks to evaluate sub-expressions are then generated.

Code for the recursive case of `reverse` is generated as follows: a suspension is created for `x`, which simply consists of a pointer to `x` itself, which is then pushed onto the stack. This generates the G-Code `PUSH X`. Next, a suspension for `reverse l` is created, which is a heap cell containing a pointer to a label code block `reverse4`, and all the free variables of the expression `reverse l`, namely `l`. A pointer to this heap cell is then pushed onto the stack. This generates the code:

```
new v
fill reverse4
```

```
fill l
push v
```

Finally a transfer instruction to `append` is generated. Since append represents a global function, not a suspension, the `JUMP` instruction is used (`ENTER` is used for suspensions). This generates the code:

```
jump append
```

Additional basic blocks are then generated for each of the sub-expressions requiring additional code. In this case, only one such code block is required, `reverse4`, to evaluate the sub-expression `reverse l`. For any such block, a header is first generated, consisting of the `LABEL` declaration, appropriate heap and stack checks, and a sequence of `GET` instructions to recover the information stored in the suspension node.

```
LABEL reverse4
    stack 1
    get reverse4 l
```

(The argument to the stack instruction may only be determined from code yet to be generated: this is no problem with lazy evaluation; otherwise some method of back-patching must be used.)

After the header has been generated, the expression `reverse l` is compiled using the compilation algorithm recursively. This generates a suspension for `l`, pushing it onto the stack, and then an instruction to transfer control to `reverse`:

```
push l
jump reverse
```

3.2.3 Compilation of Pattern Matching

There are two stages to the compilation of pattern matching: firstly, the patterns must be examined to determine which equations apply to any given arguments, and secondly, code is generated to test the arguments, and switch to the code for the right equation.

This process may be made more comprehensible by the use of another intermediate code. There are three fundamental actions in pattern matching: to head-evaluate an argument; to switch between one of several courses of action depending on the result of a head evaluation; and to select an equation as the desired choice. A fourth action, to fully evaluate an argument, is presented here for completeness. This code is called `HESF`-code from the initials of its four instructions. Any pure pattern-matching algorithm may be expressed in HESF. However, when pattern-matching interacts with guarded equations (for example, in Miranda), this code is inadequate, and would need to be extended.

The process of determining what this intermediate code should be contains all of the hard work in compiling pattern matching: translating this code to the stylised form of G-Code described in Section 3.1.3 above is relatively easy.

Acquiring HESF-code is not so easy. The algorithm described here is more complex than is absolutely necessary, since it was originally developed with the objective of calculating the range of values actually covered by each of the

defining equations of a function, which is required for Chapter 7. The use of the data structures created here to determine these ranges of values is covered fully in Section 5.7.

The HESF generation algorithm comes in five stages:

First, the patterns are massaged, if necessary, into a standard form. For example, in Miranda it is possible to write a the same variable twice in a single defining equation, the implication being that the same value be found in each place if the match is to succeed. In this case it is necessary to separate out this test into an appropriate form. For example, the function `member` to test whether a list contains a particular value might be defined:

```
member Nil x = False
member (Cons x l) x = True
member (Cons y l) x = member l x
```

This would be rewritten to

```
member Nil x = False
member (Cons y l) x =
  if (x == y) True (member l x)
```

No such massaging is needed in Flair.

Second, a table is constructed which describes the equation coverage, that is, the sets of values over which each equation is apparently defined.

Third, a selection function is applied which reduces the sets of the values in the table to make the equations disjoint, that is, only one equation is defined over any given argument values. This operation is performed on the table generated at the second stage, but can be viewed as being a source to source transformation on the original code. For example, a version of the `min` function which gives answer `0` on the empty list rather than an error might be defined

```
min0 Nil = 0
min0 l = min l
```

These two equations overlap on the argument `Nil`: assuming the first equation is to be used in this case, the second equation may be made disjoint by rewriting it to

```
min0 (Cons x l) = min (Cons x l)
```

The algorithm described in Section 5.7 is equivalent to this source to source transformation.

Fourth, the new table is used to generate HESF-code.

Fifth, the new table may be used to generate the sets of values over which the equations actually hold.

The first and third points, the massaging and selection steps, are entirely dependent on the pattern-matching convention being used, and will not be considered further — Appendix C includes the particular case of the Flair conventions, which are very simple. The fifth step is concerned with the use of the equations in proof, and will be discussed in Section 5.7. The two remaining steps are described in detail here.

The second step consists of constructing a table describing the equation coverage. It consists of a list of pairs: the first element of each pair is the value coverage, consisting of a list of constructors; the second element is the result, which may come in one of two forms: it may be a table constructed analogously, or it may be a list of zero or more equations which apply with these constructors. In fact, the top-level table itself may be just a list of zero or more equations which are always selected. For example, with `reverse`, the following table is constructed:

```
Constructor        Result

Nil                [1]
Cons               [2]
```

The algorithm for constructing the table begins with a list of lists of patterns: one pattern for each argument position for each equation. From this it first deduces a list of the constructors which may appear as the head of each of the patterns for each argument position. In general, the full complexity of the algorithm described here will not be needed; as a consequence, for an example, the somewhat pathological function `mink` will be used. `mink` takes two arguments, a strict natural number, `n`, and a strict list, `l`, and returns the minimum value of the first `n` elements of `l`, or `Zero` if `n` is `Zero` or the list is empty.

```
mink Zero l = Zero                          (1)
mink k Nil = Zero                           (2)
mink k (Cons a Nil) = a                     (3)
mink (Succ Zero) l = hd l                   (4)
mink (Succ k) (Cons a (Cons b l)) =         (5)
  mink k (Cons (if (<= a b) a b) l)
```

The equation numbers on the right hand side will be used when referring to the equations, particularly to the right hand side of each equation when deciding which to execute.

The list of lists of patterns extracted from this function definition is:

```
[[Zero, k,   k,          Succ Zero, Succ k          ],
 [l,    Nil, Cons a Nil, l,         Cons a (Cons b l)]]
```

The head constructors for each argument position may be determined by examining each of the patterns in each of the lists, substituting `*` for any pattern which is simply a variable rather than a simple pattern involving a constructor. The semantics of `*` is that it represents any constructor not otherwise listed, but does not match $\perp$. This produces the list of sets

```
[{Zero,Succ,*},{Nil,Cons,*}]
```

The `value coverage` of this function, that is, the possible combination of head constructors, may be found by forming the cartesian product of this list of sets:

```
{(Zero,Nil),(Zero,Cons),(Zero,*),
 (Succ,Nil),(Succ,Cons),(Succ,*),
 (*   ,Nil),(*   ,Cons),(*   ,*)}
```

For each element of the value coverage, it is possible to examine the patterns in the defining equations above, to see which could apply to the element of the value coverage under consideration. A variable in the pattern will match any constructor in the appropriate position in the value coverage; the constructor `*` in the value coverage will only match patterns not beginning with one of the specific constructors in that argument position. For example, `*` in the first argument position of the value coverage will not match `Zero` or `Succ` in the first argument position of any pattern.

The results of performing this matching for the value coverage above are presented below:

```
(Zero,Nil )  [1,2]
(Zero,Cons)  [1,3]
(Zero,*   )  [1]
(Succ,Nil )  [2,4]
(Succ,Cons)  [3,4,5]
(Succ,*   )  [4]
(*   ,Nil )  [2]
(*   ,Cons)  [3]
(*   ,*   )  []
```

Each element of the value coverage has an arity: this is the sum of the arities of the constructors forming the value coverage, where the pseudo-constructor `*` is given arity zero. If the arity of an entry in the value coverage is greater than zero, then the entry may be further subdivided by the arguments to that constructor. For example, the pair (`Succ`,`Cons`) has arity three — `Succ` has arity one and `Cons` arity two. The possible constructors to which these may be applied in the patterns are determined as before. The generic variable `*` appears wherever a variable appeared in the original pattern, or where a variable appeared as an argument to the given constructor. For the (`Succ`,`Cons`) case, this gives rise to a new list of sets of constructors:

```
[{Zero,*},{*},{Nil,Cons,*}]
```

which, in turn, gives rise to the value coverage determined as a cartesian product of this list of sets:

```
{(Zero,*,Nil),(Zero,*,Cons),(Zero,*,*),
 (*,   *,Nil),(*,   *,Cons),(*,   *,*)}
```

which can be matched against the original patterns to decide which equations hold over which elements of the value coverage in the same fashion as before:

```
(Succ Zero) (Cons * Nil)   [3,4]
(Succ Zero) (Cons * Cons)  [4,5]
(Succ Zero) (Cons * *)     [4]
(Succ *)    (Cons * Nil)   [3]
(Succ *)    (Cons * Cons)  [5]
(Succ *)    (Cons * *)     []
```

The entries in this table involving `Cons` have arity two, and so need a further iteration of the algorithm, which simply produces the singleton set `{(*,*)}` for the value coverage. The resulting tables of the full application of this algorithm are presented in Figure 3.11, where Table 1 is the top level table, and the other tables appear inside it as noted. This completes the second stage of the HESF-generation algorithm.

The third stage of the algorithm processes the generated table to produce a new table. The new table is identical to the first, except that at most one equation number of each list for the value coverage in the original table may appear in the corresponding list in the new table, although it is perfectly permissible for no equation numbers to appear, signifying that the entry of the value coverage is not covered by the equations. Any number of methods may be used, depending on the pattern-matching convention being used. A frequently used (and simple) method is to choose the equation which appears textually first in the function definition, which corresponds to choosing the lowest numbered equation. The table resulting from this is given in Figure 3.12. An alternative method is to make sure that in order for an equation to be selected it must be the most precise. In this case, the table coverage is examined, and the equation selected whose coverage forms the smallest proper subset of the coverage for the listed equations. This has the advantage of only depending on the meaning of the equations, not on their layout, but has the disadvantage that some systems of equations are illegal. For example, a function of two arguments which has three equations (specific on first argument, specific on second argument, general on both arguments) would not be permitted.

The fourth step of the HESF-generation algorithm, and the last to be described here, consists of generating HESF-code from the table in Figure 3.12. The algorithm is recursive and is applied to a "current table", initially Table 1 of Figure 3.12, and a list of argument position names. The resulting HESF-code consists of repeated head-evaluations followed by case analysis on the resulting head constructor. If arguments to be evaluated appear equally likely to resolve which equation should be used, then the choice is made arbitrarily. This may affect the termination characteristics of the function, but no one choice can be said to be better than any other in the abstract.

The first step is to decide which argument should be head-evaluated: the argument chosen should cause the greatest separation amongst the equations, that is, the table is divided into sections by cases of the selected argument, and the best separation occurs when the resulting tables have the least number of equations in common.

After a column has been chosen, a case analysis is performed, and for each possible constructor, a new table is formed, and each processed recursively. The recursion ends when no constructors remain in the table to be matched.

Code is generated at three points during this algorithm: once a column has been selected, a head-evaluate instruction for the corresponding argument is generated; a switch instruction is generated corresponding to the case analysis; and an equation-select instruction is generated when the recursion ends.

In the example, the first column is selected for head evaluation, and the instruction

```
Head 1
```

```
Table 1: top level table
  Zero Nil   [1,2]
  Zero Cons  see Table 2
  Zero *     [1]
  Succ Nil   see Table 3
  Succ Cons  see Table 4
  Succ *     see Table 5
  *    Nil   [2]
  *    Cons  see Table 6
  *    *     []

Table 2: Context: Zero, Cons
  * Nil  [1,3]
  * *    [1]

Table 3: Context: Succ, Nil
  Zero  [2,4]
  *     [2]

Table 4: Context: Succ, Cons
  Zero * Nil   [3,4]
  Zero * Cons  see Table 7
  Zero * *     [4]
  *    * Nil   [3]
  *    * Cons  see Table 8
  *    * *     []

Table 5: Context: Succ, *
  Zero  [4]

Table 6: Context: *, Cons
  * Nil  [3]

Table 7: Context (Succ Zero), (Cons * (Cons))
  * *  [4,5]

Table 8: Context (Succ *), (Cons * (Cons))
  * *  [5]
```

Figure 3.11: Table construction for the function `mink`

```
Table 1: top level table
  Zero Nil   [1]
  Zero Cons  see Table 2
  Zero *     [1]
  Succ Nil   see Table 3
  Succ Cons  see Table 4
  Succ *     see Table 5
  *    Nil   [2]
  *    Cons  see Table 6
  *    *     []

Table 2: Context: Zero, Cons
  * Nil  [1]
  * *    [1]

Table 3: Context: Succ, Nil
  Zero  [2]
  *     [2]

Table 4: Context: Succ, Cons
  Zero * Nil   [3]
  Zero * Cons  see Table 7
  Zero * *     [4]
  *    * Nil   [3]
  *    * Cons  see Table 8
  *    * *     []

Table 5: Context: Succ, *
  Zero  [4]

Table 6: Context: *, Cons
  * Nil  [3]

Table 7: Context (Succ Zero), (Cons * (Cons))
  * *  [4]

Table 8: Context (Succ *), (Cons * (Cons))
  * *  [5]
```

Figure 3.12: Selected table for the function `mink`

is generated. The case analysis gives three possibilities: `Zero`, `Succ`, and `*`, where `*` represents the default case. The table is split into three according to this analysis, and each table transformed by eliminating the selected column from the table and introducing from subsidiary tables the arguments corresponding to that constructor. This gives rise to the three tables of Figure 3.13. The only complex case is that of Table 10, since the other two merely require the deletion of the first column.

```
Table 9: top level table for recursive case Zero
  Nil   [1]
  Cons  see Table 2
  *     [1]

Table 10: top level table for recursive case Succ
  Zero Nil   [2]
  Zero Cons  see Table 12
  Zero *     [4]
  *    Nil   [2]
  *    Cons  see Table 13
  *    *     []

Table 11: top level table for recursive case *
  Nil   [2]
  Cons  see Table 6
  *     []

Table 12: recursive case Succ, Context Zero Cons
  * Nil   [3]
  * Cons  see Table 7
  * *     [4]

Table 13: recursive case Succ, Context * Cons
  * Nil   [3]
  * Cons  see Table 8
  * *     []
```

Figure 3.13: Recursive tables for the function `mink`

Once the first column, consisting of `Succ`, has been eliminated, the columns in the inner tables corresponding to the arguments of `Succ` may be retrieved, consisting of `Zero` and `*`. These are brought to the top level, and the cartesian product of these with the other arguments in the table are formed, giving the value coverage

```
Zero Nil
Zero Cons
Zero *
*    Nil
*    Cons
*    *
```

The equations corresponding to the coverage are then examined and lifted. Inner tables are split and regrouped according to the constructors that have been lifted to the top level of value coverage. This causes Tables 3, 4 and 5 to be each split in two. The two halves of each of Tables 3 and 5 appear directly in the new top level table, Table 10, and Table 4 is split into two new inner tables with identical coverage, Tables 12 and 13, which are referenced in the top level table, Table 10.

This case analysis gives rise to the HESF switch instruction:

```
SWITCH {
    Zero -> compile Table 9;
    Succ -> compile Table 10;
    *    -> compile Table 11
}
```

The compilation proceeds recursively, until no further matches may be performed, and the code given in Figure 3.14 is produced.

```
Head 1 Switch {
  Zero -> Eqn 1
  Succ -> Head 2 Switch {
    Nil -> Eqn 2
    Cons -> Head 2.2 Switch {
      Nil -> Eqn 3
      Cons -> Head 1.1 Switch {
        Zero -> Eqn 4
        * -> Eqn 5
      }
      * -> Head 1.1 Switch {
        Zero -> Eqn 4
      }
    }
    * -> Head 1.1 Switch {
      Zero -> Eqn 4
    }
  }
  * -> Head 2 Switch {
    Nil -> Eqn 2
    Cons -> Head 2.2 Switch {
      Nil -> Eqn 3
    }
  }
}
```

Figure 3.14: HESF code for pattern matching for function mink

HESF code may be easily converted into G-Code. Head-evaluating an argument corresponds to completing a block by pushing any current free variables, the address of a new continuation block, and then entering the argument to

be head-evaluated. The new continuation block is generated, recovering the variables saved by the original block.

Switching consists of examining the `TAG` value, comparing it to each case needing consideration, and switching to an appropriate `CASE` block for each. If there is a default case in the switch, then an unconditional jump to an appropriate `CASE` block is generated for this; otherwise an unconditional jump to the error "missing case in definition" is generated. Each case block is generated recursively, starting by recovering the contents of the current heap node — the arguments of the constructor.

An equation-select instruction is compiled into G-Code as a series of instructions to organise the internal machine state ready for the equation, and then transferring control to the first block of the equation.

There is a subtlety here involving guarded equations, which have not been mentioned in the above discussion (and are not included in FLAIR). In some cases, guarded equations can interfere with pattern matching, in the sense that the most obvious matching equation is not selected because all its equations are covered by guards, none of which are satisfied. For example, the function `f` may be defined

```
f (Cons x l)
= 1, x >= 0
f l = 0
```

which has two defining equations: the first holds for non-empty lists with positive first elements; the second equation holds not only for empty lists (the natural result of pattern matching) but also for non-empty lists with negative first elements. This can be resolved by suitable program transformation (adding an extra default case to the guarded definitions). For example, `f` may be transformed to:

```
f (Cons x l)
= 1, x>=0
= 0, otherwise
f l = 0
```

Chapter 4

Functional Program Development Environment

The research described here is directed towards producing a design for an environment for the development, manipulation and proof of functional programs. Such an environment was briefly discussed in Section 1.7 and consists of a collection of cooperating tools. This environment offers a standard interface to the tools available: an editor, a compiler, an executor, a file manager, a debugger, and a proof assistant. This idea itself is not new — many compilers are provided with such environments. What is new here is the inclusion of a proof assistant with the set of development tools, and the way in which a programmer is encouraged to move from program development to documentation and proof through the spectrum of tools.

The environment aims to provide a range of tools to assist in the development of functional programs. A program is initially constructed using an editor, based upon an original specification and design. The documentation for the program is developed in parallel with the program. When the program, or part of the program, has been completed it may be tested and debugged in the usual way, with the possible assistance of the debugger. Properties, obtained from the original specification or design, are stated, and then tested in the same way as other program fragments, to check that they are correct. Once the programmer is reasonably assured of the correctness both of the program and the properties, proofs of these properties are attempted using the proof system (described in Chapter 7) through the proof assistant (described in Chapter 8). The initial structure of these proofs may be obtained from a trace produced by the debugger during the testing of the property.

This attitude to proof has two advantages over some other methods of proof: firstly, it is performed at the workstation, with machine assistance, as a natural part of program development, rather than as a necessary part of the quality assurance programme; secondly, the process of proof is an extension of the testing process, since the initial structure of the proof is obtained directly from the testing of the property. This method of extracting the outline of a proof from testing the property contrasts with methods for extracting a program from a proof of a specification (for example, see [81]), in which a program may be automatically generated from a constructive proof that there is a function which

matches a given specification, in three ways: the primary development is of a program, not a proof; the proofs are of general properties, not function specifications, and are proved from the known properties of defined functions; and the proofs are constructed manually, with machine assistance, from programs constructed manually, rather than extracted automatically from proofs created entirely by hand. Chapter 10 discusses a possible method by which it might be possible to develop code and proof simultaneously from a specification, with machine assistance.

The environment described here has two primary objectives: it is intended to be natural and easy for functional programmers to use, and to develop and prove program properties with; and it should be sufficiently flexible to allow customisation, and the easy introduction of new tools. However, it is not the intention at this stage to develop an entirely functional system, although Chapter 10 contains some ideas relating to a purely functional development environment.

4.1 Previous Environments

Previous environments have, in the main, concentrated on providing support for programmers with imperative languages. Interestingly, Smalltalk [26] and InterLisp [75], which have been two of the most successful environments, have been based on non-traditional models. Since we are designing an environment for use in developing functional language programs, a highly functional approach seems desirable. The biggest leap to be made in realising this goal is overcoming the lack of implicit state in functional languages, typically used as a cornerstone in programming environment development.

Previous work has concentrated on acheiving three primary goals in assisting programmers: working with a single language for all programming tasks – coding, command language, and debugging; merging a set of tools to give the illusion of working with a single tool; and introducing meta-programming languages into the environment. Many environments have been created explicitly to make a single feature, for example, an advanced debugger, more accessible. Almost all environments assume that the user has access to a high-resolution bit-mapped display with a mouse and a considerable amount of CPU power.

Smalltalk and Interlisp, as mentioned above, are probably the most successful of the environments to date. These environments, together with MESA, consist of a single tool, with a single address space, but are open and extensible, allowing new functionality to be incorporated into the system at any time. GNU Emacs, with its lisp configuration language, provides an environment somewhat similar to InterLisp, providing vast and diverse functionality within a single system.

Within the auspices of the Gandalf project at CMU [29], Garlan and Miller [23] have developed a different approach to environments. The idea in this work is specifically aimed at solving the problems encountered by novice programmers. As a consequence, the environment is very much editor-centred: a family of menu-driven structure editors, semi-automatically derived from a BNF-like language definition are used to assist the programmer in writing correctly structured programs. As these are written, the parse trees for them are constructed and checked, and then their meaning is also checked. Errors are automatically collected, and modification is provided when requested. Program execution is

also driven structurally from within the editor. Much of this work is similar to that of the Cornell Program Synthesizer [74].

The MESA programming environment [73], developed at the Xerox PARC (as was Smalltalk), was expressly monolingual from its inception, and was built up, by usage rather than design, from reusable components: for example, in its user interface. The intention was to produce a system which integrated the operating system, programming language, and development environment under one large umbrella. To the user this system should appear as a single cooperative tool, while providing varied functions in a collection of windows. This illusion was created using a number of techniques: most importantly, each tool was designed with a single purpose - to assist the user in acheiving a specific goal - and to work with the other tools in doing so; the reuse of components that gave a consistent interface across all the tools; the sharing of information (and particularly the commonality of data types, made possible by the using of a single programming language); and the considerable use of first-class functions, even though MESA is not functional. Sweet, the paper's author, comments that in his opinion, this use of procedure variables in a wide variety of roles was the single most important feature of the MESA programming environment.

Magpie [16] uses a similar set of principles to create an illusion about its user interface. By allowing and encouraging tools to cooperate - by sharing information, understanding what other processes are trying to achieve, and performing many updates incrementally while the user is inactive, (e.g. thinking). Magpie is targetted at PASCAL programmers, and at integrating the standard tools (editor, compiler, debugger) into a single system. In particular, the editor window shows both static information (the code) and dynamic information (for example, breakpoints, and the values of variables). All the interaction with Magpie takes place in a single programming language - PASCAL - augmented with standard windowing, to control the actions performed, the debugging, and of course, the program being developed. The advantage of this approach is that program development becomes the task of manipulating one compound object, with the connections already made, rather than a number of distinct objects.

Blair and Malik [7] describe how functional languages can be combined with Object Oriented programming to produce a language particularly suited for developing programming environments. The model they present for Object Oriented programming is radically different to that normally proposed [85]. Instead of having objects, classes and an inheritance relation, they set forth a model based on encapsulation, whereby an object has a well-defined behaviour entirely expressed by its interface; abstraction, which allows objects with similar properties to be grouped together; and polymorphism, which permits two objects to have the same method, either with identical behaviour (as with inherited functions) or with radically different behaviours, but similar intentions (for example, print on a list and a number). From this basis the authors build up a programming support environment intended for use with the ADA programming language (although all the concepts they present are in fact language independent). Their version of Object Oriented programming, combined with essentially functional methods, enables the system description, and object heiracrchy, to be introduced incrementally, and new tools to be introduced seamlessly as and when they become relevant to the environment. This approach also encourages sharing and reuse, as with the MESA environment.

A more theoretical approach is pursued in [33]. Here, the three distinct

types of language normally encountered by programmers - programming languages, command languages, and debugging languages, are considered, and the differences between them noted. An attempt is then made to produce a unified programming language, whose features (in one way or another) may then be used in all three modes. This has some interesting consequences. Files are commonly used by a command interpreter such as the UNIX shell [8], which in a programming language is equivalent to having a persistent store. The distinction between a program and a procedure is necessarily blurred almost to the point of non-existence, since a program is simply a procedure with an extra layer of protection from the outside world. The impact on input and output for both programs and procedures is to produces a standardised idea of streams. These can be used for communication between procedures (like pipes in the shell) as well as for interaction with the user of the system. Integrating the debugging language leads to a generalised mechanism for event and interrupt handling, as well as generic process control (e.g. tracing and breakpoints) using coroutines.

As part of the Alvey Flagship project, Darlington et al. [15] propose an environment using the parallel functional language Hope+. As well as providing the basic components of an environment, this provides a system for meta-programming, where the meta-programs are written (generally interactively) in the same functional language. Operations such as memoisation and inversion of functions, together with a wide selection of transformation functions, are provided as meta-programming primitives, as are types for representing programs and program fragments. Incorporating this into a standard functional programming language allows tacticals like those used in [60] to be incorporated directly into meta-programs. It also provides a uniform development environment for both programs and meta-programs. Meta-programming for parallel programs (particularly with ALICE, a parallel graph-reduction system built on Flagship) is also described.

A different approach to graphical programming systems is described in [19]. Here, instead of working with a textual representation, each object is represented as a graphical object. These objects may then be combined by linking them together graphically to represent function calls in the program. Although not described, it is relatively easy to see how this could be extended to include dynamic information in the display, by connecting this information to the links, and likewise meta-programs (if written similarly) could be incorporated in the same model.

From these previous attempts to produce programming environments, several key points stand out. Focus is essential: each tool in the environment must know its place, and be attempting to acheive something specific, and must be prepared to cooperate with other tools with different goals. Minimising the number of languages maximises the programmer's effectiveness. As much information as possible should be presented and manipulated graphically. Information displayed to the user should be grouped in the way that the user expects, and should be relevant to what the user is trying to acheive, and not just in the way that is easiest for the tool generating it.

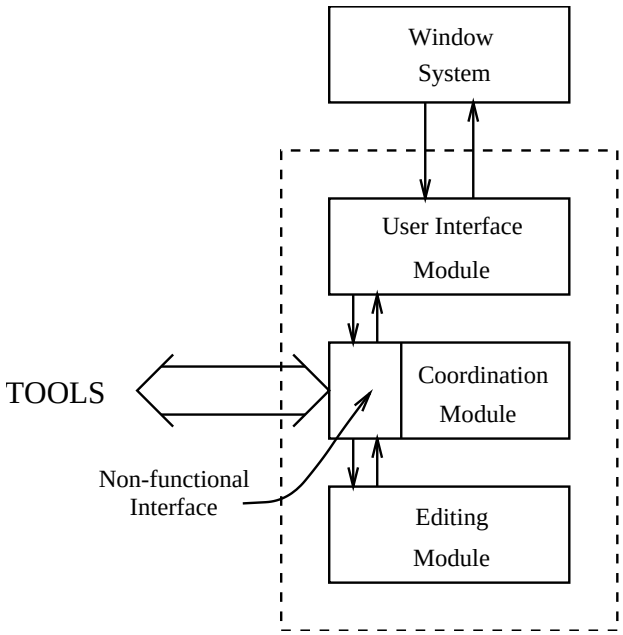

Figure 4.1: Exploded View of Coordinator

4.2 Environment Architecture

The central idea of a development environment organised as a collection of cooperating tools was set out in Chapter 1. This section provides a design for this system as a whole, and, in particular, the tool marked on Figure 1.1 as the "coordinator". Figure 4.1 gives an exploded view of the module structure of the coordinator, comprising four components: a user-interface module providing graphical and textual user-interaction services (for example, windows and icons); an editor module providing basic (and advanced) text handling and editing facilities; a functional program module actually responsible for the coordination; and an interfacing module which allows for a message-passing interface which allows any of the tools (and especially the central coordinator) to believe they live in a functional world. In this model just about everything is driven by the coordinator: the only communication path between any other tools or modules is through the coordinator; the user's interaction with the system is always directly to the UI module, and from there to the coordination module - never directly to any of the tools.

This model has a number of advantages: since all the interaction goes through one center, the system can always be regulated so that it remains consistent; optimisations can be made on what actually needs doing; facilities provided by the coordination (eg. UI and editor) do not need to be replicated; configuration information may all be placed together, and, where relevant, stored; and if, as here, the central coordinating module is programmable, the environment becomes an order of magnitude more powerful.

From the point of view of the central coordinating module, it is a standard functional program, running with the stream-based requests and responses

model. The requests issued are directed at specific tools, and the responses generated constitute input to the program. A facility is also provided within the interfacing module for handling asynchronous I/O. This is primarily intended for user input, for example, keypresses or mouse selections. Consequently, all it sees of the environment is a sequence of responses in reply to a sequence of requests.

Likewise, each of the tools in the environment is provided with a view of the rest of the environment which is very homogenous - a set of services are available, either synchronously or asynchronously, and appear to be provided by a single tool - "the environment" - although fulfilling a request could involve passing a message to every tool in the environment - even itself!

The environment is coupled together using message passing. When the environment is started, it is only really the coordinator which starts, and this immediately loads and executes the driving functional program. This program starts by performing its initialisation sequence, which will typically involve reinitialising tools from previous sessions, recovering data, and displaying windows for the user. The requests and responses streams for this functional program must always be kept in synch and $\texttt{responses}_{i+1}$ will always be the response to $\texttt{requests}_i$. Consequently, as the system starts up, the functional program is fed the response `Initialise` before it generates any requests. After this, any generated request must always have a response.

To start a tool, the central coordinating module generates the message `StartTool tool-name tool-args`, which is acted upon by the interface module. This generates a response either acknowledging the creation of a tool and giving a handle for further communication with it, or giving a diagnostic that the tool could not be started. A negotiation then begins between the coordinator and the tool to agree a protocol that they should speak - this generally consists of three numbers: a tool identifier; a protocol level; and a protocol revision. The purpose of the tool ID is to clarify the type of protocol, eg. compilers have a very different set of operations to debuggers, and so each has a specialised protocol. The level is a way of categorising the features it presents, eg. compiling a standard file is a basic (level 0) facility for a compiler; incremental compilation would be at a higher level (eg. level 2). The revision number is provided for smaller enhancements, and to allow minor changes (for example for performance) to be included. At a minimum each particular tool should be able to speak level 0, revision 0 protocol. Any others are optional. Tool 0 always addresses the interface module directly. Although physically part of the coordinator, the editor and UI modules are considered logically separate, and so must be stated in this way. Negative tool, level and revision numbers are reserved for experimental, local or temporary use. In addition to the tool-specific protocols, other more general protocols could be supported.

The protocol negotiation consists of a series of messages passed back and forth between the tool and the interface module of the coordinator. Each message consists of a suggested protocol. The coordinating module starts the negotiation, to which the tool responds. When either the tool or the coordinator receive a suggestion that is acceptable, they send an accept-protocol message. Thereafter they may communicate using that protocol.

Asynchronous I/O is handled by the interface module. Any messages received by the interface modules "out of turn" are placed in one of several queues. The number of queues is implementation-dependent, but at least two are rec-

ommended: one for standard requests, and one for priority requests (such as interrupt). When the coordinating module has nothing to do (eg. after finishing initialisation) it sends a message to the interface module asking for an event. If the interface module has asynchronous events pending, it returns the first one from the highest priority list. If no events are pending it will either block or indicate this status, depending on the exact request.

This design provides several useful features: the environment may be viewed, manipulated and controlled by an entirely functional program written in the same functional programming language in which the user of the system (a functional programmer) will be writing programs on a daily basis; the environment appears to be a single entity when iewed from any angle; the system of protocols allows for easy substitution of tools, and for an extensible architecture; the use of a complete programming language for configuration allows many more options than usual; and the separation of functionality allows massive re-use, particularly of the UI components.

4.3 Coordinator facilities

The previous section described the overall architecture of the system. The purpose of this section is to look closer at the four modules comprising the coordinator, as shown in Figure 4.1. The central coordinating module is just a functional language program executor running the user-specified control program, which defines a function from responses to requests. The interface module has been discussed in some detail, although a more formal presentation of its level 0 protocol will be given here. The editor and UI modules have not yet been discussed. The minimum requirements, and the consequent level 0 protocols for these two modules will be presented.

The presentation of the protocols is given as if defining abstract data types. However, union types are used frequently, for example, each protocol is a superset of the interface protocol, including for example, the success and failure values.

Each message is surrounded by a wrapper that identifies the tool to which the message is to be sent. The tools are identified by an integer handle; 0 is reserved for the interface module; all the others are passed to the central module as responses when the tools are started.

The interface module can be seen as a very low-level core for the environment. It provides both a requests/responses interface to a non-functional environment, and the basic services necessary for the functioning of the environment. It initially always starts talking level 0 Interface protocol, but may be negotiated higher (on the initiation of the coordinating module). The level 0 protocol is:

```
data interface_0.0_req =
  ReadRawFile FileName |
  WriteRawFile FileName [char] |
  StartTool ToolName Args |
  StopTool Tool |
  AsyncEvent |
  AsyncEventNoBlock
```

```
data interface_0.0_resp =
  Initialise Args |
  Data [char] |
  NewTool Tool |
  Event EVid [event] |
  NoEvent |
  Success |
  Failure ErrCode ErrMess

type FileName = [char]
type Args = [[char]]
type ToolName = [char]
type ErrCode = int
type ErrMess = [char]
```

The level 0 negotiation protocol (spoken by each of the tools) is:

```
data negotiation_0.0_req =
  Protocol ToolType Level Revision |
  AcceptProtocol

type negotiation_0.0_resp = negotiation_0.0_req

type Level = int
type Revision = int
```

The editor module of the coordinator provides editing services to the environment. These services essentially consist of maintaining and manipulating a collection of buffers. Each buffer contains a single piece of textual information, and will often correspond to the contents of a file.

The services provided by different editor modules may vary drastically. However, certain facilities must be provided as an absolute minimum, and these are specified within the level 0 editor protocol (see Figure 4.2).

We want the capabilities to store multiple buffers because the user is going to want to be working with more than one file at any one time, and additionally, various tools will be wanting to open editing sessions for various internal reasons (for example, the proof assistant will want scratch buffers for displaying proofs). Each of these buffers is then uniquely identified by an integer handle. Since this system is intended to be functional, the protocol for the editor has no attached state, for example, no idea of a "current buffer", although the editor module itself might well have such a notion internally (for example, for efficiency reasons).

The obvious facilities to provide are capabilities to insert and delete text. Additinally, two search procedures - one for strings, the other for regular expressions are provided. The insert and delete requests give responses of either success or failure (the most likely reasons for failure being non-existent handles or points outside the cited buffers). Search requests may produce failure, or else a pair of numbers indicating the region of text found. A region of text is always measured including the character at the start position and excluding the one at the finish position. A region where the second number is less than or equal to the first contains no text.

```
data editor_0.0_req =
  Create BuffName |
  Destroy EdBuf |
  Insert EdBuf Pos Text |
  Delete EdBuf Pos Pos |
  Search EdBuf Pos Pos Text |
  RESearch EdBuf Pos Pos RegExp |
  NewMark EdBuf Pos |
  SetMark EdBuf Mark Pos |
  DelMark EdBuf Mark |
  GetMark EdBuf Mark |
  Recover (edbuf e) (pos start) (pos end) |
  GetChanges (edbuf e) (int tidemark)

data editor_0.0_resp =
  NewBuffer BuffName EdBuf |
  LocatedAt Pos |
  Text ([char] t) |
  Changes (int tidemark) [(pos start,pos end,[char] text)]

type BuffName = [char]
type EdBuf = int
type Text = [char]
type RegExp = [char]
type Pos = Mark | Loc
data Mark = Mark int
data Loc = Loc int
```

Figure 4.2: Level 0 Editor Protocol

In addition to buffers, the other "non-obvious" feature required of the editor module is marks. A mark is a point in the buffer which does not move relative to the text. For example, if the text "`f x = x+2`" is in the buffer, and a mark is placed over the " `=` " character, then inserting " `y` " as a second argument to the function `f` will advance the position of the mark so as to leave it still over the " `=` " character. If a region containing the mark is deleted, then the mark is left pointing to the character after the region. If text is inserted at the mark, it is placed before the mark. Functions exist in the protocol to create, set, interrogate and delete marks. Two marks `Mark 0` and `Mark 1` are created and maintained automatically by the system, and mark the beginning and the end of the buffer. They are particularly useful as arguments to, for example, delete and search to specify the entire buffer.

In order to make full use of views in the user interface module, it is also necessary to be able to recover text, and to note changes over a period of time. In order to do this, the facilities to recover a particular area of text, and to get all the changes since a specified prior point in time - a tidemark - are provided. The latter recovery requires an integer specifying the tidemark, and along with the returned list of changes, the response gives a new tidemark, representing the current state of the buffer.

The user interface module should provide a complete range of services, at high, intermediate and low levels. Such a range of services would however, be totally impractical to describe here. Instead, four categories of operations are presented: text windows, formatting/layout windows, dialog boxes, and a "drag and drop" facility. While all these facilities are doubtless familiar from current window systems, a brief description of their intended use follows.

Text windows are used primarily for editing sessions, interactive sessions (e.g. with command line shells or debuggers), and for the presentation of textual data in the form of lists. For this reason, they are generally associated with an editor buffer, are scrollable vertically (and often horizontally), and need to be updated whenever an edit takes place.

Layout windows use Tex-style [46] formatting and layout constructions to build up a formatted structure. This can be used for presenting help screens, drawing simple diagrams and constructing complex displays.

Dialog boxes allow for user entry in a number of ways and forms of structured data. The applications for this range from answering simple questions to entering all the data for a particular program or task.

These three categories of operations all result in the presentation of a window on the screen containing some information. The fourth category - "drag and drop" - is orthogonal to these. It represents the transfer of data between any two windows. "Drag and drop" indicates that the information is selected, usually by a suitable mouse gesture, and then copied to another point where it is placed.

Specifically not included in this simple protocol are: any form of graphics display, including simple graphics, postscript display, raster display, or any form of 2-D or 3-D rendering; low-level access to attributes such as colors, shading, patterns or input devices; and any intermediate level access to menus, popups, dragging, rubberbanding or buttons.

The level 0 protocol is defined as the union of the three window protocols. The drag and drop is internal to the UI module, and so does not have a separate protocol.

Text windows are created by specifying a view to be associated with, a

```
type UI_0.0_reqs =
  UI_tw_0.0_reqs |
  UI_lw_0.0_reqs |
  UI_db_0.0_reqs

type UI_0.0_resps =
  UI_tw_0.0_resps |
  UI_lw_0.0_resps |
  UI_db_0.0_resps
```

Figure 4.3: User Interface Protocols

```
data UI_tw_0.0_reqs =
  CreateTWindow View Size Size bool bool |
  Refresh Window

data UI_tw_0.0_resps =
  NewWindow Window

type Size = int
```

Figure 4.4: Text Window Protocol

minimum size in characters, and whether or not horizontal and vertical scrollbars are to be provided. In the absence of a horizontal scrollbar, characters will be wrapped to the next line; in the absence of a vertical scrollbar, any lines which do not fit will be truncated from the display. The only other request supported by a text window is to redisplay itself.

Layout windows follow TeX layout and formatting instructions. Consequently, they contain aspects such as boxes, glue, groupings, fonts, text, lines, etc. While the protocol itself is very small, the data type `texl` is very large. Figure 4.5 contains the protocol together with a sample of the data type `texl`.

As with layout windows, dialog boxes have a fairly simple protocol, coupled

```
data UI_lw_0.0_reqs =
  CreateLWindow texl

data UI_lw_0.0_resps =
  NewWindow Window

data texl =
  Text [char] |
  Font FontName FontSize texl |
  Box [Attributes] texl |
  ...
```

Figure 4.5: Layout Window Protocol

```
data UI_db_0.0_reqs =
  CreateDBox DBox

data UI_db_0.0_resps =
  NewWindow Window |
  DBoxActivated Window Int DBoxRtn
```

Figure 4.6: Dialog Box Protocol

```
data dbox =
  Label [char] |
  Active [char] int |
  Select [[char]] |
  Scroll [[char]] int int |
  Overlay [([char],dbox)] |
  Outer |
  Check [char] bool |
  Text bool int [char] [[char]] |
  TBBox [dbox] |
  LRBox [dbox]

data dboxrtn =
  Str [char] |
  OvRtn [char] [([char],dboxrtn)] |
  CkRtn bool |
  BoxRtn [dboxrtn]
```

Figure 4.7: Dialog Box Data Structure

with a complex data type. The window is created, and a handle returned. When one of the action buttons is activated, the UI module generates an asynchronous message to the coordinator, indicating the activation button pressed and giving all the data captured in the window. The protocol is given in Figure 4.6.

Creating the dialog box is done using an object of the class `dbox`. A workable definition is given in Figure 4.7 although it is easy to see other features which could be provided.

The layout of the dialog boxes uses some of the same TeX-style boxes as the layout window. However, in this case the primitives are generally much higher level objects such as scrolling lists and popup menus, and are all active, accepting and responding to input.

An overlay is a dialog box where a selection style box with popup menu changes the entire look of the menu (i.e. all the options change). For example, in a tool for entering citations, changing the type of citation would change many of the fields available for entry. The `Overlay` constructor generates a box for this prime selection but does not place it. That is left for the `Outer` constructor, which is only meaningful inside an `Overlay` data structure. Generally an `Outer` constructor will appear in each of the inner dialog boxes in the same place.

A text entry field takes a boolean specifying whether it is fixed size or scrol-

lable, an integer specifying the size of the field, a string containing the initial value, and a list of strings which form an optional pop-up menu.

Each of these objects returns a structured value, as defined in the data type `dboxrtn` (see Figure 4.7). `Label`s, `Active`s and `Outer`s do not return a value in this structure. `Select`s, `Scroll`s and `Text`s all return a string, `Overlay`s and `Check` boxes return `OvRtn` and `CkRtn` values respectively, and the boxes return `BoxRtn` structures. The program which created the dialog box may break down this structure and extract the relevant values.

4.4 Views

In order for a programming environment such as the one presented here to be truly effective, it must provide a high-bandwidth communication path between the user and a program. In other words, when using the environment, the user should find that all relevant information is grouped together, and less relevant information, when available, is placed elsewhere, but directly accessible from the core information displayed.

Without such an environment, the user generally manipulates a program using one tool at a time. Each tool gives the user information about a certain part or aspect of the program, but not a complete, coordinated view of the entire program.

The environment takes the information present in the individual tools and combines it so that the user sees a coherent picture of the program. The method used here to achieve this is to allow user-definable views of a collection of editor buffers.

Each buffer has some text in it which has been generated in some way (typically this is by a user writing a program, but also by tools such as the compiler and the debugger). In more advanced versions of the environment, the buffers might also contain graphical images and diagrams. Such an environment could contain all the source text for a program, the errors produced by the compiler, the current executing state of the program, information from the debugger, property proofs, design documents, provisional documentation, etc.

A user requests a view into this information by creating a window on the screen and specifying the viewing function to be used and any additional parameters that may be required (in general, views will be named on creation and most frequently asked for by name). The window is presented, and a repaint request is sent from the UI manager to the central coordinating module. Here a decision is made as to exactly what needs repainting, the relevant view is recovered, and the appropriate display instructions are generated and sent to the UI manager (merged with requests for information directed at the relevant editor buffers).

When a change is made within a window containing a view, this change has to be reflected not only in the view, but also in the buffer underlying the view. For this purpose, a second function is provided which translates back from a position in the view window into a position within the corresponding source buffer. The writeability of any portion of a view depends upon the writeability of the source buffer. A third function is provided which determines whether or not a complete refresh of the screen is required when a single change is made. These latter two functions can both be written in a general way, working off

data generated by the original projection function.

Each view consists of a projection function, some buffers and marks within buffers, and some maintained state information. The projection function is a mapping from the sections of buffers and state information to a dialogue (a function from responses to requests), e.g.

```
proj :: [sub-buff] -> state -> responses -> requests
```

The state is given by a mapping from arbitrary keys (although a special enumerated type would be useful) to arbitrary values.

The requests and responses are used for two purposes: firstly to collect necessary information from the editor buffers, and secondly to display it using calls to the user interface module. In order to improve efficiency, it might be desirable to perform the projection initially into an editor buffer and from there display it. Any subsequent refreshes of the window would not require a complete reprojection of the view, although when a change was made, a full reprojection into the buffer might be necessary.

The primary purpose of views is to allow important information from different tools to be combined easily. However, because the text is produced by the projection function extra text (annotations) may be slipped in. For example, when a new buffer is used, a line could be inserted identifying the buffer, the tool, or its status.

In order to take advantage of a single function to handle the required unprojection operations, it is necessary to store a 'trace' of exactly what went into making up the displayed view. This consists of a list of structures, each identifying a number of characters and a region of a buffer from whence they came. If they were added explicitly by the projection function, a distinct buffer identifier can be used to indicate this. Working from this list, any position in the view buffer may easily be translated back into a position in a real buffer, and any changes made to a buffer may be easily tested to examine their impact on the displayed view. As these occur, a partial or total reprojection may be performed as appropriate.

4.5 Introducing tools into the system

So far only the coordinator has been presented. For the system to be of any real use, more functionality must be provided. This could be done by writing ever more code to fit into the coordinator's program. However, there are three problems with this: it requires that the functionality be provided in exactly the same programming language; it prevents existing tools from being modified and used; and it reduces the modularity of the system.

The alternative is to have separate tools which are controlled by the coordinator. In this design, they communicate using message passing in a style similar to that already discussed for internal requests within the coordinator. When the central coordinating module requests a new tool to be started, the interfacing module will start the process and establish a two-way communications link. Once this is done, the coordinator will start to negotiate a protocol with the tool. The interfacing module accepts the messages from the coordinator, translates them into a binary form for transmission, and then transmits them

over the communication link to the tool. When the tool responds, the process is reversed.

Each tool within the environment communicates directly only with the interfacing module. However, because these messages are then passed to the relevant tool for processing, each tool believes that it is communicating with a single tool providing the services of the entire environment.

Several stages must be performed in order to integrate a new tool cleanly into this environment. To begin with, a tool must be provided, and given a suitable message-passing interface understanding a relevant protocol. If necessary, a new protocol must be devised, and the coordinator reprogrammed to understand and negotiate the protocol, and to start off the tool. Finally, any other tools which are affected (for example, they wish to use some new features) should be modified.

The most difficult part of this operation is generally to provide the tool itself. If a tool is being written from scratch, it would seem sensible to try and make it as functional as possible, so that it fits neatly into the requests and responses model, although this is certainly not a requirement (and any programming language and method may be used). Once this is done, or if an existing tool is to be used, an interface must be constructed on the tool to communicate with the coordinator, and an interpreter for the protocol must be implemented within the coordinator.

In order to support an entirely new form of tool, a whole new protocol needs to be developed. However, in general it is felt that most tools brought into the environment will communicate using one of the existing protocols. If a tool presents some radical new functionality, then a new version of a protocol may be necessary (for example, going from a full to an incremental compiler). A new tool of an existing type probably still needs to get its details inserted somewhere. These changes will all have to be made to the coordinator's program. For early compatibility when introducing a new tool, the coordinator can negotiate a minimum protocol with it (for example, forcing a compiler capable of incremental compilation to do full compilation).

When a new tool has been merged into the environment, existing tools will generally be unaffected. However, if a new tool provides some new functionality which is of use to one or more existing tools, then these tools might want to be updated to take advantage. For example, a tool which generated line-oriented statistics, e.g. line number, function nesting, etc.) might want updating when a new tool was introduced offering test-coverage information. Another example might be if a new tool required detailed structural information about a program, then a new compiler might be required to provide this information.

4.6 Standard Tools

A wide variety of tools could be provided within this environment. Besides the coordinator already described, at least two are undoubtedly needed: a compiler and an executor. This section describes these two along with a simple file manager. Chapter 8 describes introducing the proof assistant into the environment, and Chapter 10 describes three other tools which are areas of current research.

A functional language compiler is provided for two reasons: it is needed to compile the programs that run internally in the central tool; and it is required

in any functional program development environment. Such a compiler receives just one message in general: to compile a program, with source code included in the message. An acknowledgement of the request is returned immediately, and once the program is compiled, one of two messages is sent asynchronously: either to report a successful compilation, returning the object code; or to report failure, returning a list of errors found. Aside from the interface, the compiler would behave much as any other functional language compiler, performing a sequence of steps on the initial source program until appropriate object code is produced. More advanced compilers could easily be provided: for example, an incremental compiler, which might engage in a dialogue with the central tool concerning which functions need to be recompiled.

An executor for user programs, very similar to that provided within the central tool is required to execute programs generated by the compiler. How this works depends largely on the form of the object code produced by the compiler: for example, G-Code, as described in Chapter 3, is most easily executed using a small, tight interpreter, such as that presented in the Appendix. The executor takes in general two kinds of message: the first is the presentation of object code, which replaces any function definitions previously stored in the executor, and the second is a request to evaluate an expression, within the context of the object code most recently presented to the executor.

A file manager is provided to allow a consistent, easily expanded interface to the operating system filestore. This is responsible for maintaining the environment's idea of the filestore, and should always be used in requesting the storage or retrieval of a file. This allows tools to access the most up-to-date version of a file if, for example, this is located in an editor buffer rather than in the file store. More advanced versions of the file manager might provide facilities such as those provided by SCCS [1].

Chapter 5

Extended Types

This chapter introduces extensions to the standard Haskell type system to provide a sharper system of types, which forms a foundation for the proof system to be developed in the following chapters. The standard type system provides a flexible means for describing values, and for grouping these together into disjoint sets, which makes the task of writing correct programs easier. The system presented here extends the usual system by providing a notation in which subtypes may be defined and manipulated. These subtypes need not be disjoint, and set operations such as union, intersection and difference may be performed on them. Such subtypes allow program properties to be investigated at varying levels of granularity, according to the needs of the proof.

Sections 5.1 and 5.2 introduce the notations to describe algebraic and polymorphic extended types as collections of data values, analogously to the standard Haskell `data` definition. Section 5.3 extends the Haskell type synonym notation to support the operations union, intersection and difference between extended types. Section 5.4 covers the specification of the types of functions in a sharper fashion (made possible by the use of subtypes to represent the arguments and results of functions). Section 5.5 gives an algorithm which allows all of the extended types in a system to be represented uniquely, while Section 5.6 provides an algorithm to check that the unique representations found are also valid over infinite, as well as finite values. Section 5.7 presents a means of resolving overlapping equations in function definitions by placing restrictions on each of the defining equations, describing the values of the free variables of the equations over which the equations are regarded as holding. Section 5.8 discusses overloading and Haskell's `class`es feature, and how these may be represented using the extended types system.

The work in this chapter is to a large extent collaborative with my Supervisor, Dr. Ian Holyer. It is the only part of the work described here which had been started before I started work on my PhD in August 1988. Section 5.1 describes the basic system of extended types introducted by Holyer, and set out in his paper [36]. Section 5.2 describes my extensions to this system to deal with polymorphic types. The type operators described in 5.3 and the functional notation of 5.4 also appeared in [36], as did the idea of a unique representation of these types. The algorithms presented here however are original work. The problems encountered with systems of types with infinite values which cannot be correctly partitioned was mentioned in Holyer's work, together with the streams

example, repeated here, and the "real-world" observations about how common a problem this is likely to be. The exact conditions under which this would occur, and an algorithm for detecting these, are original work. The work on pattern matching, both here and in Chapter 3, is original, although inspired by Holyer's comment on the desirability (for proof purposes) of having completely separate "true equations" representing the various cases of the function definitions.

5.1 Algebraic Types

Extended types have approximately the same syntax as that of the underlying object language. Extensions are provided to represent subtypes, and to permit the operations of union, intersection and difference over types. Since I am using Flair, essentially a subset of Haskell, as the object language, the syntax of extended types mirrors the type syntax of Haskell.

Two examples of standard type definitions are presented below. The type `tree` supports binary trees of integers, while the type `result` supports a success or failure situation: in the successful case, a tree is returned.

```
data tree = Empty | Tree tree int tree
data result = Fail | Success tree
```

Extended types are interpreted as sets of values (domains). In this standard case, these values are defined as all of those which match any of the patterns in the type definition, together with the value $\perp$ (which is taken to include exceptional values, if any). For example, the above types may be viewed as defining sets of values satisfying the equations:

```
trees = { ⊥, Empty, Tree t1 i t2 | t1,t2 ∈ trees, i ∈ ints }
results = { ⊥, Fail, Success t | t ∈ trees }
```

The presence of $\perp$ in a type permits the presence of approximate values. In addition to the values which may be given using the constructors `Empty` and `Tree`, the value $\perp$, representing an approximation to all other values is present in the type `trees`. Because it is a member of the type, it may appear as a value of type `trees` when a larger value is being constructed using the constructor `Tree`. Consequently, there are an infinite number of partial values contained in the type `trees`. An infinite value may be represented as the limit of an infinite sequence of approximations. The set of values described by an extended type includes all infinite values which are limits of sequences of partial values wholly contained in the extended type. So for example, the partial values

```
⊥, Tree Empty 0 ⊥, Tree Empty 0 (Tree Empty 0 ⊥), ...
```

form an infinite sequence of approximations to the infinite value

```
Tree Empty 0 (Tree Empty 0 (Tree Empty 0 (Tree Empty 0 ...)))
```

and since all the partial values are in the type `trees`, the infinite value is also.

The presence of $\perp$ in the second example only permits one partial value directly, $\perp$ itself, and no infinite values, although there are partial and infinite

values in the second type because the argument to `Success` may be a partial or infinite tree.

To enable subtypes to be defined, a new keyword `edata` (for extended data definition) is introduced, which describes a set of values in exactly the same way as the standard data definition. For example, the extended data definitions:

```
edata emptytree = Empty
edata success   = Success tree
```

define a type `emptytree` consisting just of the empty tree (together with $\perp$), and a type `success`, which only contains successful results, each returning an encapsulated tree, apart from the value $\perp$. These may be expressed in the set notation used above as:

```
emptytrees = { ⊥, Empty }
successes  = { ⊥, Success t | t ∈ trees }
```

If it is desired to have more direct control over the inclusion of $\perp$ in a type, an alternative extended type data definition `sdata` (for strict data definition) may be used. In this case, the value $\perp$ is not included in a subtype unless the constant `Bottom` is included directly in the definition. For example, in the definition,

```
sdata definiteresult = Fail | Success t
```

only the possibilities `Fail` and `Success` are acceptable definite results — it is not possible for a `definiteresult` to be $\perp$. This corresponds to the set of values:

```
definiteresults = { Fail, Success t | t ∈ trees }
```

Subtypes may also be formed in which one or more arguments to one or more constructors are subtypes of the types given in the original standard data definitions. For example, strictly finite trees may be represented by a strict data definition, such as that above for the type `definiteresult`, provided that the arguments to the constructor `Tree` are restricted to being finite.

```
sdata finitetree = Empty | Tree finitetree int finitetree
```

This corresponds directly to the set of values given by the equation:

```
finitetrees =
    { Empty, Tree t1 i t2 | t1,t2 ∈ finitetrees, i ∈ ints }
```

5.2 Polymorphic Types

The previous section introduced the means by which subtypes could be introduced and defined for standard algebraic types. Polymorphism provides a means for describing a family of algebraic types using a single definition. All the types in such a family share a common structure, the only differences being in the types referenced in some of the patterns. These differences are indicated by the presence in these positions of type variables in place of type identifiers. A

polymorphic type definition lists all the type variables it uses. Polymorphism thus allows a family of related types to be easily and precisely defined in a single definition. For example, the family of list types may be defined:

```
data list t = Nil | Cons t (list t)
```

As for the algebraic types, polymorphic types may be described by a set of values: of course, since a polymorphic type represents a family of types, it must be represented by a set of sets of values:

```
lists = { l1 t | t ∈ types }
  where  l1 t = { Nil, Cons v l | v ∈ t, l ∈ l1 t }
```

Subtypes of these types may be formed, which remain polymorphic in exactly the same way as the standard type: the only difference is to the contents of the inner sets of values. For example, the subtypes `emptylist` and `finitelist` may be defined using the keywords `edata` and `sdata` as:

```
edata emptylist t  = Nil
sdata finitelist t = Nil | Cons t (finitelist t)
```

Note that in the definition of the subtype `emptylist`, the polymorphic variable `t` is never used — this set in fact contains two values ⊥ and `Nil`. The sets of sets of values may be presented as before:

```
emptylists = { l2 t | t ∈ types }
  where  l2 t = { ⊥, Nil }
finitelists = { l3 t | t ∈ types }
  where  l3 t = { Nil, Cons v l | v ∈ t, l ∈ l3 t }
```

In all polymorphic subtypes, the polymorphic structure is preserved. The type definitions have the same number of polymorphic variables, in the same order, while the constructors have exactly the same polymorphic arguments as in the standard definition: it is not possible to replace these with subtypes.

5.3 Type Operators

New types may be formed by performing set operations (such as union, intersection and difference) on existing types. This uses an extension of the type synonym notation in Haskell. The set operations are represented by the relational operations `&` and `|` for intersection and union, and `-` for difference. These operations act on existing types, and it is not possible to introduce patterns in these equations, and it is not legal to use this notation to define one set of values in terms of itself (either directly or indirectly). For example, the types `nat`, `nonpos`, and `int` represent three groups of integers. The integers themselves (`int`) might be represented using the Peano axioms as:

```
data int = Zero | Succ int | Pred int
```

although this representation does not give a unique form for any given integer. This corresponds to the set of values `ints`, given by:

```
ints = { Zero, Succ i, Pred j | i,j ∈ ints }
```

The sets of natural numbers (`nat`) and non-positive numbers may be given using the subtype mechanism as:

```
edata nat    = Zero | Succ nat
edata nonpos = Zero | Pred nonpos

nats    = { Zero, Succ i | i ∈ nats }
nonposs = { Zero, Pred j | j ∈ nonposs }
```

At the same time, this has eliminated all the non-canonical forms of the numbers (such as `Succ (Pred Zero)` for `Zero`). The extension to the type synonym notation allows the definition of canonical integers by forming the union of the sets `nats` and `nonposs`. That is,

```
etype canint = nat | nonpos

canints = { Zero, Succ i, Pred j | i ∈ nats, j ∈ nonposs }
```

The keyword `stype` is provided analogously to the pair of keywords `edata` and `sdata`. The difference between `etype` and `stype` is in their treatment of $\perp$, the non-terminating value. `etype` includes $\perp$ along with the result of the set operation, while `stype` does not; although obviously a type defined using `stype` may still contain $\perp$ if the set operation itself generates it.

Parameters to functions may be declared to belong to a specific extended type. This is useful in proofs, where it may be desirable to restrict the set of values over which a variable may range. The mechanism for declaring parameters is like that used to declare the types of functions in the object language. In Haskell, this is the operator `::`. The two kinds of declaration may be distinguished by examining the uses of the object being defined.

```
n, i :: naturals
t1, t2 :: finitetrees

search n Nil = False
search n (Tree t1 i t2) =
  (n == i) | search n t1 | search n t2

search :: naturals->finitetrees->booleans
```

Here, the declarations of the variables appear at the top. These use the operator `::` to declare variables of a given type, and at the end, the operator is used to assert the type of the function.

Declarations may also be made for polymorphic variables. A variable `t` such as that used in the definition of `list t` may be given the declaration

```
t :: types
```

This notation is used, and an extension proposed, in Section 5.8.2 which discusses a means of incorporating the Haskell class system into the extended types framework.

5.4 Function Types

Function types are defined using an extension of the syntax in the underlying language. In Haskell, a function type is written as a series of type expressions separated by the special operator `->`. For example, the function `mirror`, which produces the mirror image of a binary tree, is given the type

```
tree->tree
```

which means that the function takes one argument of the type `tree`, and produces a value of the type `tree`. On any other type, the result type is `error`.

Subtypes of function types may be defined, by extending this syntax to allow for subtypes of arguments. In addition to the usual form `t->u`, the form

```
t1->u1 & t2->u2 & t3->u3 & ...
```

is introduced, where the $\mathtt{t}_i$ are disjoint data types. This represents the set of values,

$$\{\ \mathtt{f}\ \mid\ (\forall\ x\ \in \mathtt{t1}\ \cdot\ \mathtt{f}\ x\ \in \mathtt{u1})\ \wedge\ (\forall\ x\ \in \mathtt{t2}\ \cdot\ \mathtt{f}\ x\ \in \mathtt{u2})\ \wedge\ \ldots\ \}$$

As with the standard type definitions, if any value v is not in the union of the $\mathtt{t}_i$, then this produces the value $\bot$. In general, the union of the sets of values $\mathtt{t}_i$ will be the same as the set `t`, and the union of the sets of values $\mathtt{u}_i$ will be a subset of the set `u`.

With the function `mirror`, given by the definition

```
mirror Empty = Empty
mirror (Tree t1 i t2) = Tree (mirror t2) i (mirror t1)
```

each case of the definition gives rise to one typing for the function. If the subtypes of `tree`, `emptytree` and `nonemptytree` are defined using the syntax in Section 5.1 as

```
edata emptytree    = Empty
edata nonemptytree = Tree tree int tree
```

then the function `mirror` may be given the extended type:

```
emptytree->emptytree & nonemptytree->nonemptytree
```

which refines the original definition by stating that `mirror` will never produce a non-empty tree from an argument which is an empty tree, or vice versa.

Functions may be declared in this style in exactly the same way as functions are defined to have their standard types in the object language. For example,

```
mirror :: emptytree->emptytree & nonemptytree->nonemptytree
```

and, similarly, the type synonym notation may be used to define types containing such functions:

```
etype treefns = emptytree->emptytree & nonemptytree->nonemptytree
```

It is important to note that these typings for functions are definitions, and cannot be checked or inferred by the type system, but require definition and proof from the user.

5.5 Unique Representation of Extended Types

It is important to be able to determine whether two subtypes describe the same set of values. After the process of enriching the system of types as described above, this becomes a non-trivial task. Section 5.5.1 provides an algorithm to enable the unique representation of algebraic types; Section 5.5.2 extends this to handle polymorphic types; and Section 5.5.3 shows how these unique representations for data types allow the unique representation of the extended function representations given above.

5.5.1 Algebraic Types

In a system of Extended types, or E-types, it is possible for two or more types to overlap, that is, for there to be one or more values which appear in more than one E-type. Indeed, in a limited this way this also occurs with the standard Hindley-Milner type system, since $\perp$ appears in all types, and the empty list appears in all of the polymorphic list types. Since E-types are sets of values, these overlaps may be expressed as intersections between the sets. It is highly desirable that there be a unique representation for all the defined subtypes, since this makes it possible to tell when two types describe the same set of values. A representation which makes it easy to test whether or not two subtypes intersect, and, if they do, what values are in the intersection, is even more desirable.

Such a representation is described in [36]. Each type in the system is represented as a union of one or more smaller types. These smaller types, called `base types` are the smallest intersections of the defined subtypes and partition the value space, that is: firstly, any given value will be a member of at most one of them; and secondly, their union is the same set of values as the union of the original types. This representation in fact has limited problems with infinite values in some cases: this problem is addressed in Section 5.6.

Finding these base types in the general case is quite complex. The algorithm described in [36] demonstrates that an algorithm is possible, but is not efficient enough to be used as an implementation. A more efficient, practical, algorithm is presented here[1].

The following definitions present five subtypes: the natural numbers, represented by the standard Peano representation; the non-positive numbers (or negative naturals), represented analogously; the integers, represented as a union of the naturals and non-positives; finite lists, defined in the standard way, without any special syntax; and trees, as presented earlier (note that `Nil` is used both for the empty list and the empty tree). These are all defined as subtypes of a single standard data definition.

```
data val =
  Zero | Succ val | Pred val |
  Nil | Cons val val | Tree val val val

sdata nat  = Zero | Succ nat
sdata np   = Zero | Pred np
stype int  = nat  | np
sdata list = Nil  | Cons int list
```

[1] Complete Miranda code for this is presented in the Appendix

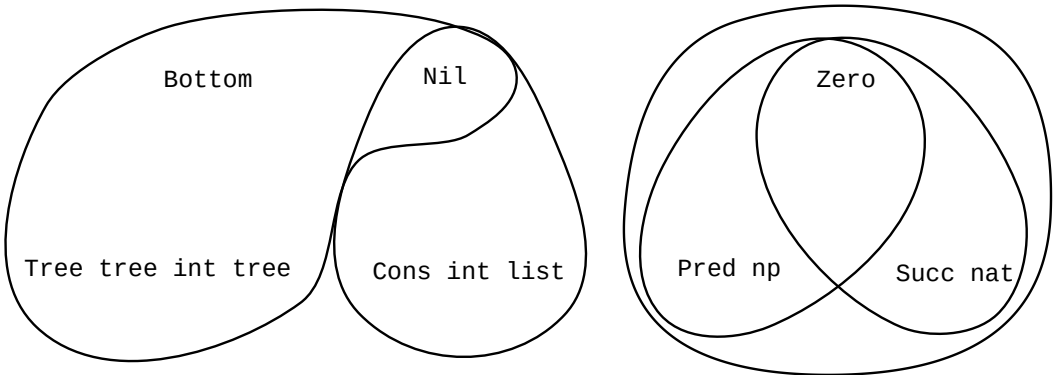

Figure 5.1: Intersections between types

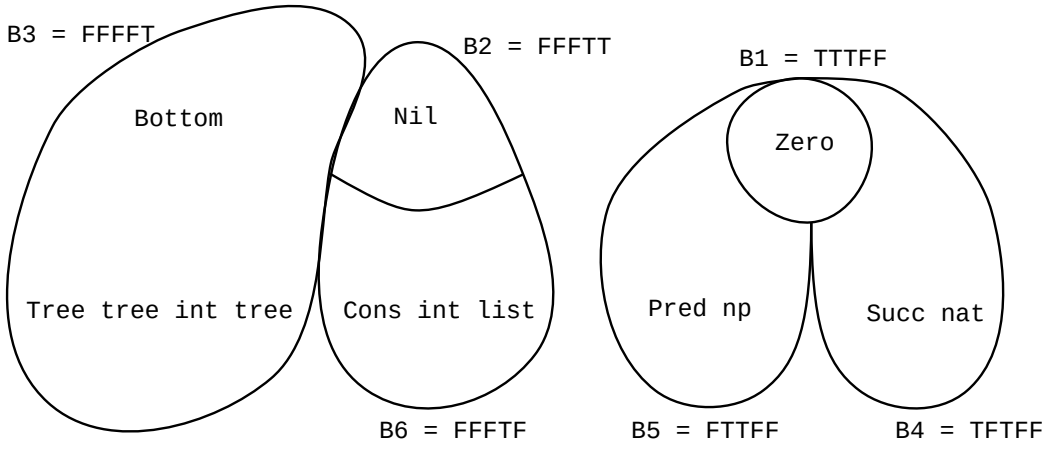

Figure 5.2: Base types from Figure 5.1

```
sdata tree = Bottom | Nil | Tree tree int tree
```

Figure 5.1 shows the arrangement of these values in the form of a Venn diagram, and Figure 5.2 shows the base types for this system which are to be found by the algorithm.

As described above, each of the original types declared in the system may be expressed as a union of base types. The base types may be expressed as the union of a collection of patterns, as with standard definitions. Alternatively, they may be represented uniquely as intersections of the original types and their complements. In this representation, for each of the original types, either the original type or its complement appears. For example, the base type `B2` (see Figure 5.2) is represented by the intersection `(-nat)&(-np)&(-int)&(list)&(tree)`. Since this representation is very stylised, it is easily represented as a list of boolean values: `True` represents an intersection with the original type; `False` represents an intersection with its complement. To make the representation unique, the same order is always used, although which order is not important. Here I will

use the order in which the types are written above. The type `B2` is thus reduced to `FFFTT`. The algorithm finds all the non-empty intersections of this form, using what amounts to a fixpoint method. Because there are n types, each of which may be included or excluded in any base type, there must be 2^n such sets in total, although only a small number are typically non-empty.

Initially, the only known base type is the peripheral type, consisting of all the values not in any of the subtypes (represented by a list consisting entirely of `False`: the type `B0` in Figure 5.2, `FFFFF`). This type includes all the values which are in the standard type, but not in any of the subtypes.

The algorithm proceeds by attempting to construct values using the constructors in conjunction with the currently known base types, and testing these values for membership in the original types. The resulting lists of membership tests gives rise to new base types. The collection of base types is extended until no new ones can be found.

The simplest case is for constant patterns which consist only of single constant constructors. For each type which is directly defined in terms of patterns, a simple test is used to determine whether or not the constructor is included in the definition. If included, then the original type is used in the intersection; otherwise, the complement is used. Once this has been calculated for all the directly defined types, the intersections may be calculated for the types defined in terms of other types. The order in which this is performed is important, since the derived types may form a dependent hierarchy (although cycles are not permitted).

In the example, the constants are `Zero`, `Nil`, and `Bottom` (across the top in Figures 5.1 and 5.2). `Zero` is in the types `nat` and `np`, `Nil` is in `list` and `tree`, and `Bottom` is in `tree`. This gives rise to the base types

```
B1 TT?FF  Zero
B2 FF?TT  Nil
B3 FF?FT  Bottom
```

The values for `int` can now be calculated as `nat | np`, where `nat` is the first column and `np` is the second, which produces

```
B1 TTTFF  Zero
B2 FFFTT  Nil
B3 FFFFT  Bottom
```

Non-constant patterns consist of a constructor applied to a fixed number (the arity of the constructor) of type arguments. The original type definitions contain patterns involving the original types. Each pattern represents a set of values, and it is possible to create a pattern by applying a constructor to base types, instead of to original types. The set of values represented by such a pattern is given by considering the base types as intersections of the original types. If the values given by a pattern form a subset of the values of an original subtype, the pattern is said to intersect with that original type. Any given base pattern may be tested to see if it intersects with each of the original types in turn. Such a sequence of tests produces a sequence of boolean values, which constitute a base type: the one containing the values in the base pattern.

It is, however, not possible to determine the set of values given by a base pattern directly. Instead, it may be determined whether or not a given base

pattern intersects with a given original type by testing to see if it matches one or more patterns in the original type. A base pattern matches an original pattern if:

i the constructors of original and base pattern are the same; and

ii for each argument b_i in the base pattern, and corresponding argument t_i of the original pattern, the element of b_i — a sequence of booleans, one for each original type — corresponding to the type t_i is `True`.

In order for all base types to be found, all possible combinations of base types and constructors must be tried. These are easily generated. In the example, there are four non-constant constructors (`Succ`, `Pred`, `Cons`, `Tree`) and three base types so far (ignoring `B0`, which will never appear in a successfully matched pattern). This gives a total of 42 base patterns to be generated, which are

```
Succ B1; Succ B2; Succ B3
Pred B1; Pred B2; Pred B3
Cons B1 B1; Cons B1 B2; ... Cons B3 B2; Cons B3 B3
Tree B1 B1 B1; Tree B1 B1 B2; Tree B1 B1 B3;
...
Tree B2 B1 B1; Tree B2 B1 B2; Tree B2 B1 B3;
...
Tree B3 B3 B1; Tree B3 B3 B2; Tree B3 B3 B3;
```

The matching process produces the following results for the patterns with constructor `Succ` or `Pred`, where results in no original type give rise to the peripheral type `B0`:

```
Succ TTTFF   matches   Succ nat, Succ np, Succ int
                  in   nats   producing   TF?FF
Succ FFFTT   matches   Succ list, Succ tree
               in no original type
Succ FFFFT   matches   Succ tree
               in no original type
Pred TTTFF   matches   Pred nat, Pred np, Pred int
                  in   nps    producing   FT?FF
Pred FFFTT   matches   Pred list, Pred tree
               in no original type
Pred FFFFT   matches   Pred tree
               in no original type
```

Again, the boolean operation defining `int` can be applied to the new base types, giving the two new base types:

```
B4 = TFTFF
B5 = FTTFF
```

The base patterns which match for the constructors `Cons` and `Tree` are (with patterns generating the peripheral type `B0` = `FFFFF` omitted):

```
Cons B1 B2      matches   Cons int list
                     in   lists   producing   FF?TF
Tree B2 B1 B2   matches   Tree tree int tree
                     in   trees   producing   FF?FT
Tree B2 B1 B3   matches   Tree tree int tree
                     in   trees   producing   FF?FT
Tree B3 B1 B2   matches   Tree tree int tree
                     in   trees   producing   FF?FT
Tree B3 B1 B3   matches   Tree tree int tree
                     in   trees   producing   FF?FT
```

Again applying the boolean expression for `int` gives rise to:

```
B6 = FFFTF
B3 = FFFFT
```

where `B3` is the same base type as generated above by the constant `Bottom`. These base types `B0`–`B6` are those indicated in Figure 5.2.

Repeating this algorithm to use all the new combinations of base patterns (805 in total) yields no new base types, and the algorithm terminates, having found all the base types in this system.

```
B0 = FFFFF = (-nat)&(-np)&(-int)&(-list)&(-tree)
B1 = TTTFF = ( nat)&( np)&( int)&(-list)&(-tree)
B2 = FFFTT = (-nat)&(-np)&(-int)&( list)&( tree)
B3 = FFFFT = (-nat)&(-np)&(-int)&(-list)&( tree)
B4 = TFTFF = ( nat)&(-np)&( int)&(-list)&(-tree)
B5 = FTTFF = (-nat)&( np)&( int)&(-list)&(-tree)
B6 = FFFTF = (-nat)&(-np)&(-int)&( list)&(-tree)
```

Once all the base types have been found in this way, each of the original types may be represented as a union of these base types. The representation of base types as intersections between the original types gives an easy method of deducing which base types are to be combined to form the union for each type — it is exactly those which have a positive intersection for that original type. If the boolean representation is viewed as a matrix, the transpose of this matrix gives the unions very easily, where now `True` is interpreted as meaning that a base type is part of the union, and `False` is interpreted as meaning that the base type is not part of the union.

```
nat  = FTFFTFF = B1 | B4
np   = FTFFFTF = B1 | B5
int  = FTFFTTF = B1 | B4 | B5
list = FFTFFFT = B2 | B6
tree = FFTTFFF = B2 | B3
```

This representation makes it very easy to deduce the desired properties of types, for example, the intersection of and differences between two types.

The performance of this algorithm is easy to improve in a limited way. A large number of checks can be avoided for each pattern where the constructor is not in the original type being tested. As the algorithm has been described,

```
k :: types

data val k =
  Zero | Succ (val k) |
  Nil | Cons k (val k) | Tree (val k) (val k) (val k) |
  Leaf k | Branch (val k) (val k)

edata nat k   = Zero | Succ (nat k)
edata list k  = Nil  | Cons k (list k)
edata tree k  = Nil  | Tree (tree k) (nat k) (tree k)
edata ptree k = Leaf k | Branch (ptree k) (ptree k)
edata leaf k  = Leaf k
```

Figure 5.3: Polymorphic type and subtypes

the same patterns are tested on every iteration, producing the same base types. This can be avoided by constructing the patterns to be tested more carefully.

This algorithm is guaranteed to terminate. At each iteration, the number of base types must either increase or stay the same. If it stays the same, the algorithm terminates immediately. There is a maximum number of 2^n base types (that is the most that can be represented) where n is the number of original types, and at least one must be added each iteration for the algorithm not to terminate. Consequently, 2^n is an upper bound for the number of iterations the algorithm will take. Each iteration of the algorithm takes a finite number of steps (computed as the sum of pattern numbers, where each pattern produces a^k steps, where a is the arity of the constructor, and k is the number of base types at this iteration, bounded above by 2^n). In practice, the number of of final base types is usually small, and generally only two or three iterations are required, so the algorithm is quite efficient.

5.5.2 Polymorphic Types

The above representation and algorithm may be easily extended to cover polymorphic types. The extensions all stem from two basic differences between algebraic and polymorphic types: firstly, the presence of polymorphic variables in the type definitions; and secondly, the fact that type variables do not behave the same as types — specifically, subtypes cannot be formed from them.

An example type and set of subtypes is presented in Figure 5.3. The standard data type `val` contains representations of the natural numbers, polymorphic lists, and two kinds of trees (with different representations). Although both trees appear to be polymorphic, the type `tree k` is in fact independent of its polymorphic variable — the trees only contain natural numbers.

Five subtypes of `val` are defined, for naturals, lists, both types of trees, and also for single-element polymorphic trees. Note that while it is impossible to form the subtype of a polymorphic argument to a constructor, it is possible to form the polymorphic subtype of a polymorphic type used as an argument to a constructor.

The extended algorithm begins by discarding all the polymorphic information in the subtype definitions: this means discarding arguments to constructors

```
edata nat   = Zero | Succ nat
edata list  = Nil  | Cons list
edata tree  = Nil  | Tree tree nat tree
edata ptree = Leaf | Branch ptree ptree
edata leaf  = Leaf
```

Figure 5.4: Non-Polymorphic subtypes

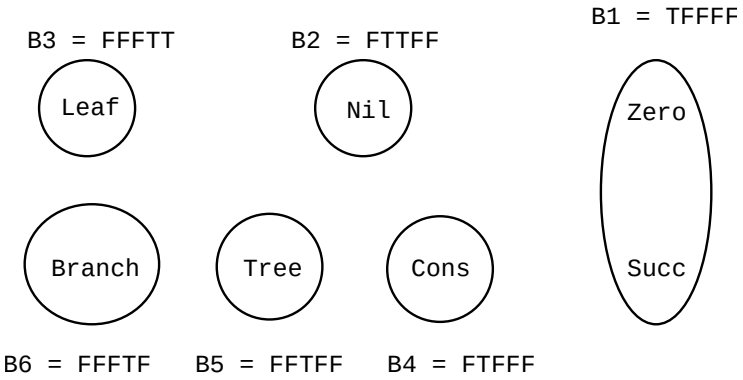

Figure 5.5: Base types after application of algebraic algorithm

which are polymorphic variables, and any polymorphic arguments to types. This produces the subtype definitions as given in Figure 5.4. It is impossible for this process to cause any discrepancies in the definitions, since all occurrences of any constructor must be treated equally.

The algorithm given in the previous section may then be applied to these subtypes (since they are no longer polymorphic), which produces the base types given in Figure 5.5.

The base types are given a polymorphic representation, where each base type takes the same arguments as the original subtypes. The original types may then be rewritten by the same transposition process used before. The final base types and definitions of the original subtypes are presented in Figure 5.6.

5.5.3 Function Types

An extended notation for the representation of function types was introduced in Section 5.4. This allows for more precise statements to be made about the nature of a function. Suppose the natural numbers to be defined as in Section 5.3 and that the subtypes `zero`, `nat`, `nonpos`, `positive` and `negative` have been defined from this (summarised in Figure 5.7).

The factorial function over natural numbers, `fac` may be defined as

```
fac Zero     = Succ Zero
fac (Succ k) = mul (Succ k) (fac k)
```

A possible type for this function (and the one inferred by the Hindley-Milner type checking algorithm) is `int->int`. Another possible type, defined by the

```
B1 k = TFFFF
B2 k = FTTFF
B3 k = FFFTT
B4 k = FTFFF
B5 k = FFTFF
B6 k = FFFTF

nat k   = B1 k
list k  = B2 k | B4 k
tree k  = B2 k | B5 k
ptree k = B3 k | B6 k
leaf k  = B6 k
```

Figure 5.6: Final polymorphic base types and subtypes as unions

```
data int       = Zero | Succ int | Pred int

edata zero     = Zero
edata nat      = Zero | Succ nat
edata nonpos   = Zero | Pred nonpos
edata positive = Succ nat
edata negative = Pred nonpos
```

Figure 5.7: The type `int` and its subtypes

user in the extended system described here, is `naturals->positives`. This is equivalent to another possible definition defined within the extended system, `zeros->positives & positives->positives`. There is clearly a requirement here, as for the data types, for a unique representation of the best function type. The best function type is defined relative to the available types. With only integers available, the typing `int->int` is the best function type. With the extended type naturals, the typing `nat->nat` is best, and with the types zeros and positives, the type given above is the best. The best function type is provided by considering the base types generated by the algorithm of Section 5.5.1. This gives the types illustrated in Figure 5.8.

Each combination of base types as arguments can be considered in turn. Since `fac` only has one argument, it is simply that each base type may be considered in turn: this gives rise to an intersection of function types. The result of the application to each type may be obtained by determining which equation (or equations) applies for that argument, and then determining the type of the result of that equation for an argument in the specified base type. There are three base types which are putative arguments to `fac` and correspond directly to the subtypes `zero`, `positive` and `negative`. The base type `TTFF` only matches the pattern `Zero`, which is the argument of the first defining equation of `fac`. Consequently, the type of `fac` for an argument in the base type `B1` is the type of the expression `Succ Zero`, which is `positives` which corresponds to the base type `B2`. This gives the partial function typing `B1->B2`. The argument `TFTF`, only matches the pattern `Succ k`, so the second defining equation applies. This

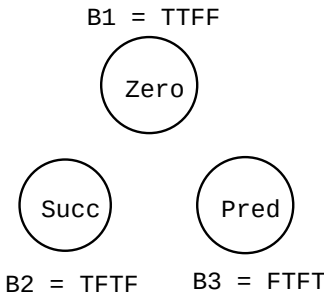

Figure 5.8: Base types of integers

```
k :: types

data  list k   = Nil | Cons k (list k)

edata elist k  = Nil
edata nelist k = Cons k (list k)
```

Figure 5.9: The function `map` and associated types

can be inductively shown to always return a positive answer (both arguments to `mul` must always be positive), so the resulting base type is `B2`, giving a partial typing `B2->B2`. The argument `FTFT`, corresponding to the type negatives, does not match any equation, and so produces no partial typing. The full typing of `fac` is thus the intersection of the two partial typings:

```
fac :: B1->B2 & B2->B2

fac :: zero->positive & positive->positive
```

Higher order functions may be handled in exactly the same way, except that, as for polymorphism, the typing does not interfere with non-data arguments. In this case, this means that no attempt is made to analyse by base type cases any arguments of a function which are themselves functions (according to the inference algorithm). This is necessary, since the representation of function types is not itself finite (there is no finite exhaustive list of function types), and therefore, no full representation can be given which involves listing all the possible arguments to a higher order function. Furthermore, the data types are broken down because they are expressed as unions of the smallest possible types: functions are represented as intersections of larger types. As an example, `map` (see Figure 5.9) is given the extended representation:

```
map :: ((k->k)->B1 k->B1 k) & ((k->k)->B2 k->B2 k)

map :: ((k->k)->elist k ->elist k ) &
       ((k->k)->nelist k->nelist k)
```

5.6 Infinite Values

E-types, as described here, include total, partial, and infinite values. As explained at the beginning of this chapter, infinite values are included in E-types as the limits of sequences of partial approximations. For example, in the definition of streams:

```
sdata stream = Bottom | Cons zero stream
sdata zero   = Zero
```

The partial values ⊥, `Cons Zero` ⊥, `Cons Zero (Cons Zero` ⊥`)`, ... form a sequence of partial approximations to the infinite value `Cons Zero (Cons Zero (Cons Zero (...)))`.

It is a requirement of the base types presented in Section 5.5 that the base types partition the value space, so that each value is in exactly one base type. Furthermore, Section 5.5.1 presents an algorithm for achieving this for all finite values. However, it is possible to find type systems for which this property does not hold for infinite values as limits of finite approximations. For example, it is possible to divide `streams` into two types — those of even length, and those of odd length:

```
sdata evenstream = Bottom | Cons zero oddstream
sdata oddstream  =          Cons zero evenstream
```

This represents a partitioning of streams for finite values — all streams must be of either even or odd length. The algorithm of Section 5.5 reflects this, and produces two base types, which correspond directly to the two types `evenstream` and `oddstream`. There are no finite values which are in the intersection. But, both `evenstream` and `oddstream` contain an infinite sequence of approximations to the infinite value `Cons Zero (Cons Zero ...)`, so the infinite value is in both, and must be in the intersection.

{ ⊥, `Cons Zero (Cons Zero` ⊥`)`, ...} ⊆ `evenstreams`
{ `Cons Zero` ⊥, `Cons Zero (Cons Zero (Cons Zero` ⊥`))`, ...} ⊆ `oddstreams`

However, the presence of infinite values is determined by the presence of an infinite sequence of approximations, and such a sequence does not appear in the finite intersection type, but rather, two different sequences appear, one in each of the non-intersecting types.

The base types are correct for finite values — they are not correct for infinite values. This problem could be solved by changing the way in which infinite values are treated in this system: either they could be specifically defined and included in various types, rather than being viewed as limits of infinite sequences; or they could be granted exceptions to appear in more than one type.

However, such problems with infinite values are quite rare (in fact, this is the most practical yet encountered), so rather than adjust the way in which all infinite values are represented, it was decided to prohibit any type definitions which gave rise to this problem. In consequence, an algorithm was needed to find general cases of this kind.

In order to demonstrate adequately the problems encountered in trying to detect the general case of this problem, a more complex, if unrealistic, example is presented below.

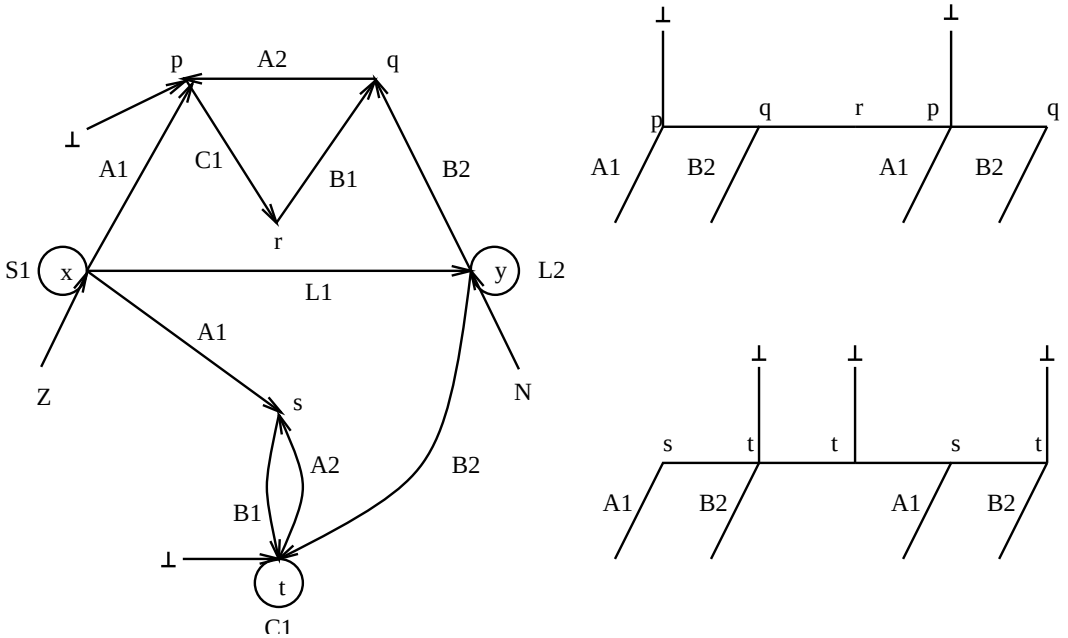

Figure 5.10: Subtypes as a graph

```
sdata p = Bottom | A x q
sdata q = B r y
sdata r = C p
sdata s = A x t
sdata t = Bottom | B t y | C s
sdata x = Z | S x
sdata y = N | L x y
```

where `x` and `y` are the familiar types natural numbers and lists of natural numbers. The types `p`, `q`, and `r` form a cycle, as do the types `s` and `t`. Among others, the infinite value `v` where `v = A Z (B (C v) N)` appears in both types `p` and `s`. This value is represented in Figure 5.10 together with a graph representation of the type system.

This problem of an infinite value occurring in more than one type, but not in their intersection is very rare, and only occurs under strong conditions:

i The graph of subtypes must contain two cycles, with specified starting positions, which match — that is, the same cyclic value may be generated by traversing each cycle an appropriate (though not necessarily equal) number of times. The cyclic value is not complete until the original starting positions have been reached simultaneously for both cycles.

ii $\perp$ must appear at least once in each cycle.

iii At at least one point during the traversal of the cycles, $\perp$ must appear in the current type in one cycle, but not in the other.

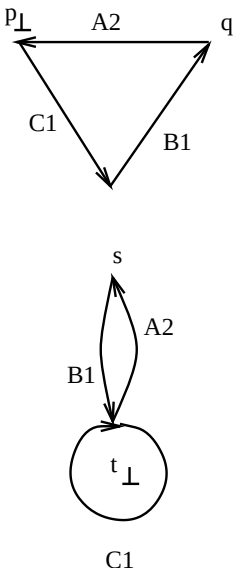

Figure 5.11: Cycles containing ⊥

The motivation for these conditions is that: (i) states that there must be a single cyclic value which appears in more that one type in the system; (ii) states that each of the values must generate an infinite sequence of approximations to a single infinite value; and (iii) states that there is at least one infinite sequence of approximations which will not be in the finite intersection.

An algorithm for identifying such cycles is now presented. First of all, all cycles defined by (i) and (ii) are detected and separated into a set of separate graphs, as in Figure 5.11, at the same time annotating all nodes which have an edge leading to ⊥. Note that the cycles not containing ⊥ in Figure 5.10 have been eliminated.

An exhaustive search is now possible, starting from each pair of nodes which have matching edges leading to them, and such that ⊥ is in one node and not in the other. In this case, this gives three possibilities: `(p,s) [A2]`, `(q,t) [B1]`, and `(r,t) [C1]`. From here it is possible to follow the edges backwards until there are no longer any possible transitions, in which case this pair of cycles does not cause a problem, or until the original starting position is reached, in which case the pair of cycles does cause a problem. This is illustrated in Figure 5.12.

This problem is very rare: most infinite values are represented by direct cycles (such as the definitions of `streams` and `trees`). It never occurs with standard Haskell data definitions, since these all include ⊥, and so there can never be the necessary imbalance between two cycles. The worst case execution time of this algorithm, an exhaustive search, is large — possibly as much as $\texttt{ca}.\texttt{T}^4$, where `ca` is the number of constructor/argument combinations, and `T` is the number of types in the system. In practice, it is vastly smaller, since virtually no starting positions for the search will be found.

Initial Position: (p$_\perp$,s)
A2 p$_\perp$ ← q s ← t$_\perp$

New Position: (q,t$_\perp$)
B1 q ← r t$_\perp$ ← t$_\perp$

New Position: (r,t$_\perp$)
C1 r ← p$_\perp$ t$_\perp$ ← s

New Position: (p$_\perp$,s)
Same as original, $\perp$ imbalance found --- error

Figure 5.12: Finding an inconsistency in type definitions

5.7 Equations and Restrictions

The pattern matching algorithm of Section 3.2.3 allows functions to be defined in an ambiguous fashion: the equations of the function definition are not independent whenever there is an overlap between the patterns used to define the cases of the function. In the following chapters, these equations will be used as axioms for proofs of program properties. It is therefore important that they should be true equations. The best solution to this is to rewrite the functions so that they are defined independently, and each rewritten equation holds over all values of the appropriate types which match the patterns. Unfortunately, this rewriting process changes the equations, which means that any programmer manipulating a program using the proof system will find it different to the one originally written.

An alternative is to associate with each equation defining the function a restriction on the values that it may be defined over: this is in fact just an extension to the idea that the patterns restrict the values that an equation may be defined over. A restriction consists of a boolean expression restricting free variables from matching certain patterns. For example, in the function `min0`, defined in Chapter 3,

```
min0 Nil = 0
min0 l = min l
```

the first equation has no need of a restriction — it applies whenever the argument is `Nil`. The second equation however, applies in every case except when the argument is `Nil`. Using the binary infix operator `~match` to represent the operation of a variable not matching a pattern, and writing boolean restrictions in curly braces to the right of the equation, this function may be written as

```
min0 Nil = 0        { True }
min0 l = min l      { l ~match Nil }
```

`~match` is a meta-function, that is, it is defined in terms of the value associated with the expression on the left and a pattern to the right, rather than being evaluated as an ordinary function.

Using the ideas of subtypes presented earlier in this chapter, these kinds of tests may be written more economically as type memberships. In this example, the second test may be written as

```
l :: nonemptylist
```

where `nonemptylist` is the type containing only lists which are non-`Nil` and non-$\perp$. The boolean expressions are not directly restricted in what may appear in them — as will be seen, the general form is for a conjunction of disjunctions — and if the object language being considered includes guarded equations, then these may be included directly as part of the restriction. For example, the function `min0` may be defined directly (without reference to `min`) using guarded equations as

```
min0 Nil = 0
min0 (Cons a Nil) = a
min0 (Cons a (Cons b l))
= min0 (Cons a l), a <= b
= min0 (Cons b l), otherwise
```

The patterns are all disjoint, so the first two equations always apply. The third and fourth equations match the same patterns, but the free variables are used to determine which should be used in any circumstance. The case "otherwise" is interpreted as all other cases being `False`. Guards must be either non-overlapping or must be rewritten to be non-overlapping before being used in restrictions. The rewriting process may be done automatically, presuming the guards are to be tested in a specific order, by adding the negation of all the previous tests to each guard. The above equations may be rewritten with restrictions as

```
min0 Nil = min0                                  { True }
min0 (Cons a Nil) = a                            { True }
min0 (Cons a (Cons b l)) = min0 (Cons a l)       { a <= b }
min0 (Cons a (Cons b l)) = min0 (Cons b l)       { ~(a <= b) }
```

The approach taken to determining such restrictions in the general case is to use the tables constructed in Section 3.2.3 for pattern matching. It will be recalled that a table was constructed to describe the relationship between equations and values, and then a selection function applied to ensure that only one equation may be applied for any given values, and that the resulting table was used to generate pattern matching code.

The first table, generated from the patterns, describes the sets of values over which the equations appear to be defined; the second table, used to generate the code, describes the sets of values over which an equation is applied. It will also be recalled that the second table is a subtable of the first: that is, no equation may apply to any given values in the second table to which it did not apply in the first table. It is clear that the restrictions may be characterised by being, in some sense, the difference between the two tables.

Each entry in a table represents a particular value coverage: at the top level, this is a direct relation between the arguments to the function, and the constructors appearing in the coverage. In nested tables, the constructors at

```
Zero Nil                          [2]
Zero (Cons * Nil)                 [3]
(Succ Zero) Nil                   [4]
(Succ Zero) (Cons * Nil)          [4]
(Succ Zero) (Cons * (Cons * *))   [5]
```

Figure 5.13: Differences in coverage for function mink

```
Zero l                          1 ->
k Nil                           2 -> Zero Nil
k (Cons a Nil)                  3 -> Zero (Cons * Nil)
(Succ Zero) l                   4 -> (Succ Zero) Nil
(Succ Zero) l                        (Succ Zero) (Cons * Nil)
(Succ k) (Cons a (Cons b l))    5 -> (Succ Zero) (Cons * (Cons * *))
```

Figure 5.14: Differences in coverage by equation number

each level need to be used as arguments to the appropriate constructors at the level above to recreate the values. For example, the tables in Figure 3.11 contain amongst others, the entries:

```
Table 1: Succ Cons    see Table 4
Table 4: Zero * Cons  see Table 7
Table 7: * *          [4,5]
```

At the top level, Table 4 applies whenever the arguments are of the form `(Succ *) (Cons * *)`. Within Table 4, Table 7 applies whenever the arguments are of the form `(Succ Zero) (Cons * (Cons * *))`, and in Table 7, Equations 4 and 5 apply whenever the arguments are of the form `(Succ Zero) (Cons * (Cons * *))`. The nesting of tables produces a tree structure, where the leaves are always (possibly empty) lists of equations which apply to the values as constructed from the nested contexts as shown here. In the table generated from the patterns, these lists may be of arbitrary size; in the selected table (such as Figure 3.12) each list at a leaf is either empty or a singleton list. The differences between the two lists may be simply characterised by a list of pairs covering all the leaves: the first element of each pair is a reconstructed value; the second element of each pair is the list of equations included in the generated table but excluded from the selected table. Any leaves in the list which have no difference between the two tables are discarded. Figure 5.13 gives the list of pairs for the function `mink`.

This list may be reorganised, and presented as a mapping from equation numbers to values, as shown in Figure 5.14. Once this has been done, with the original patterns written alongside, it becomes clear how the restrictions may be determined.

The structure of the reconstructed values matches that of the patterns, inasmuch as the latter is defined, and the free variables of the patterns correspond to patterns in the reconstructed values. The free variables in the values are represented by `*`. The restriction on a single variable for a single value may be represented by a negative match using the `~match` operator. If more than

```
mink Zero l = Zero                  { True }
mink k Nil = Zero                   { k ~match Zero }
mink k (Cons a Nil) = a             { (k ~match Zero) \/ (a ~match *) }
mink (Succ Zero) l = hd l           { (l ~match Nil) /\
                                          (l ~match (Cons * Nil)) }
mink (Succ k) (Cons a (Cons b l)) =
  mink k (Cons (if (a<=b) a b) l)   { (k ~match Zero) \/ (a ~match *) \/
                                          (b ~match *) \/ (l ~match *) }
```

Figure 5.15: The function mink with restrictions

one variable appears in a single pattern, then at least one of the variables must be restricted. This is represented by a disjunction of the variable restrictions. For example, Equation 3 is restricted by the values `Zero (Cons * Nil)`. The patterns for Equation 3 contain two variables `k` and `a`. The restrictions on these variables are

```
k ~match Zero
a ~match *
```

where `*` is a free variable, denoting any value. The second of these restrictions can never be true. The restriction for the whole equation, formed by the disjunction of these is

```
(k ~match Zero) \/ (a ~match *)
```

which may be simplified to `k ~match Zero`.

If a single equation has more than one restricting value on it, then all of these must be satisfied simultaneously. In Figure 5.14, two restrictions are placed on the variable `l` in order for Equation 4 to hold, namely

```
l ~match Nil
l ~match (Cons * Nil)
```

Both of these must hold, and so the restriction is their conjunction,

```
(l ~match Nil) /\ (l ~match (Cons * Nil))
```

The complete set of restrictions for mink is presented in Figure 5.15. The program fragment "restrict.m" in Appendix C provides an algorithm within the context of the compiler of Chapter 3 for generating such restrictions.

The addition of these restrictions may be thought of as a program transformation, which leaves the function unaffected, and preserves the function's termination conditions.

5.8 Overloading and Classes

So far, extended types have been considered as subtypes of types defined using the standard `data` definitions. Overloading may be viewed as representing super-types as well. This is discussed here by extending the notation to cover

unions of standard types as well as unions of subtypes. The use of overloading should fit in with the mechanism provided in the object language. Unfortunately, this is not yet fully defined. Section 5.8.1 introduces overloading and overloaded functions, while Section 5.8.2 discusses how the features of polymorphism and overloading may be combined to model the `class`es system in Haskell.

5.8.1 Overloading

Overloaded functions can be represented easily within this type system, since all function types are represented as a union of base types resulting from the application of each possible combination of base type arguments. As an example, consider overloading the function `reverse` to apply both to (finite) lists (inverting their order), and to (finite) trees (forming their reflection).

```
data nat = Zero | Succ nat
data list = Nil | Cons nat list
data tree = Leaf nat | Branch tree tree

n :: nat
l :: list
t1,t2 :: tree

reverse Nil = Nil
reverse (Cons n l) = join (reverse l) (Cons n Nil)

reverse (Leaf n) = Leaf n
reverse (Branch t1 t2) = Branch t2 t1
```

The base types produced by the given set of data declarations is exactly the set of initial types, together with the peripheral set `B0` = `FFF`. The Hindley-Milner algorithm would reject this function normally, but might be extended to handle union types and then would produce the type

```
(list|tree)->(list|tree)
```

It is possible to do better than this however, since the type of the result always matches the type of the argument, something not specified by the above type. Analyzing the four cases of the definition of reverse give rise to the four types

```
list->list
list->list
tree->tree
tree->tree
```

All other type combinations produce an error, that is:

```
nat->B0
B0->B0
```

which gives a complete type signature for the overloaded reverse of

```
list->list &
tree->tree &
nat->B0 &
B0->B0
```

This is not type correct in Haskell, which does not permit functions to have arguments which are union types, and the only means of supporting overloading is through classes (see Section 5.8.2). However, because of the extended type representations for functions, the type of this function is conceptually no more complex than those for `map` given above on Page 71.

5.8.2 Classes

Classes are a recently introduced feature of Haskell, and are the subject of separate ongoing research. A `class` in Haskell is a means of grouping together a collection of distinct types which, nevertheless, have some features in common, and provides a means of treating these features systematically. Specifically, classes in Haskell provide a means to control the overloading of operators. For example, the class `Eq` in Haskell covers all the types for which an equality operator is defined. When a new type is added, it may be added to this class by providing a suitable definition of the equality operator.

```
data tree = Nil | Tree tree int tree

instance (Eq tree) =>
    (==) : eqtree

eqtree Nil Nil = True
eqtree (Tree t1 i t2) (Tree t3 j t4) =
    i == j & eqtree t1 t3 & eqtree t2 t4
```

A class instance gives rise to a dictionary of functions: the keys are the standard names for the functions (e.g. `==`), while the values are the individual defining functions (e.g. `eqtree`).

A function signature may indicate that a function may act on values of any type which belongs to a given class. For instance, the function `quicksort` requires that the relations equality (`==`), less than (`<`), and greater than (`>`) be defined over the type of the elements to be sorted. This is indicated by the class `Order`, which includes all types which are ordered.

```
quicksort :: [Order t]->[Order t]

quicksort Nil = Nil
quicksort (Cons x Nil) = (Cons x Nil)
quicksort l =
  quicksort less ++ eqs ++ quicksort more
  where
    less = filter (<mid) l
    more = filter (>mid) l
    eqs = filter (==mid) l
    mid = l!(#l/2)
```

Any type which is an instance of the `Order` class may then use the `quicksort` algorithm, and appropriate functions will be used in place of `<`, `>`, `==`. The dictionary of function definitions is passed as an extra argument into any function that requires it. The semantics of functions defined in this way is entirely dependent on the evaluation process. Current research is directed toward simplifying and extending the class system, and giving it a semantics independent of the evaluation process.

Classes may be modelled in the extended types system given here by the use of polymorphic variables restricted to a specified collection of types. By itself, this only enables the type to be defined: implementation is discussed later. It is possible to collect all relevant instance declarations into a single definition, for example:

```
class numericals = { ==,+,-,*,/,<,> } { int, float, complex }
```

might be used to indicate that a class `numericals`, consists of the instance types `int`, `float`, and `complex`, and that the operators shown are overloaded by being defined on all of these types.

As described in Section 5.8.1, overloading may be implemented by typing a function as a mapping between union types, or as the intersection of several mappings between single types. The latter typing,

```
list->list &
tree->tree
```

may be rewritten as

$$\forall\ p \in \{\ \texttt{list, tree}\ \} \cdot (p \rightarrow\ p)$$

Here p is a type variable ranging over the types lists and trees, and so the function type is a restriction of a true polymorphic function which has the type

$$\forall\ p \in \texttt{types} \cdot (p \rightarrow\ p)$$

So `reverse` may be given a typing in the class system as follows:

```
class lt = { reverse } { list, tree }

reverse :: lt->lt
```

In Haskell, the class mechanism requires that the functions be defined separately to enable the inference algorithm to work. Using overloading as described in Section 5.8.1 using user-supplied function types, one function definition is sufficient, with a variety of cases to handle all the types.

In the same way that functions may be typed with polymorphic variables, they may be typed with restricted polymorphic, or class, variables. For example,

```
lt = class { reverse } { list, tree }
k :: lt
s :: k

palin s
```

```
= True,   reverse s == s
= False,  otherwise

palin :: k->bool
```

defines the class `lt` as above, with the overloaded function `reverse`. A class variable, `k`, is used to provide a type for the argument to the function `palin`. The function `palin` determines whether the object passed in as argument is palindromic. It is given the extended function type `k->bool`. In Haskell, it would be given the type `(Lt k)=>k->bool`.

This idea may be extended by using class variables as arguments to polymorphic types, for example, in the function `doublelist`:

```
k :: numericals
n :: k
l :: list k

doublelist Nil = Nil
doublelist (Cons n l) = (Cons (n+n) (doublelist l))

doublelist :: list k->list k
```

Since `k` is a polymorphic variable restricted to numeric types, the operator `+` is valid over any two values of type `k`. This extends polymorphism by allowing functions to "look inside" the elements of lists when the polymorphism is constrained by the class system. However, the value of this extension, and the consequences of it, are as yet undetermined.

Chapter 6

Proof in a Functional Language

The primary objective of this research is to produce an environment in which functional programs may be developed, properties of such programs may be stated, and the proofs of these properties may be constructed alongside the development of the programs. The design of the environment for development was described in Chapter 4; this is extended in Chapter 8 to include a tool to assist in the construction of proofs. This chapter discusses various styles of proof which were considered, with respect to two important criteria: how easy a functional programmer finds it to use to prove program properties; and how well it copes with the standard constructions of functional programs — equations and recursion. It has been assumed that the functional programmer will be acquainted with the principles of programming, and will have an intuitive understanding of the meaning of programs; it seems less reasonable to assume that the programmer has much formal training in logic or proof.

Several different logics and styles of proof were considered for use in the environment, and compared to see how well they met the two criteria outlined above. Section 6.1 reviews four logics, and Section 6.2 investigates five different styles for presenting proofs. In both cases a style which is close to the underlying functional language is sought. An additional consideration is that, since the proofs must be checked automatically by the proof assistant, sufficient detail must be provided in the proof to make it completely rigorous — it is not possible to omit either steps or details concerning individual steps in the interests of clarity, although it is assumed that many such details may be hidden from the user by the interface provided by the proof assistant. In this chapter, all excess detail is ignored — the general approach of each proof style is considered adequate.

6.1 Logics

There are a large number of different logics which may be used for proof. However, most of these have been created for specific purposes by mathematicians and are not "accessible" to programmers — the intricacies of the logic involved make understanding, let alone creating, a proof far too complex for program-

F	→	F	T	F	∨	F	F	F	∧	F	F	¬ F	T
F	→	T	T	F	∨	T	F	F	∧	T	T	¬ T	F
T	→	F	F	T	∨	F	F	T	∧	F	T		
T	→	T	T	T	∨	T	T	T	∧	T	T		

Figure 6.1: Interpretation of connectives for classical logic

mers without specific training. Thus the available choice of logic is dramatically reduced, and among those discussed here there is a considerable unity — for example, each is a variant of first order predicate logic. The differences include the number of truth values, the meanings of the connectives and quantifiers, and the available deduction rules.

In classical first order logic, there are exactly two truth values: `true` and `false`. The connectives $\rightarrow$, $\wedge$, $\vee$ and $\neg$ are given their understood meanings in English, and defined using tables of truth values (see Figure 6.1). The quantifiers $\forall$ and $\exists$ are also given the expected English meaning, and defined by substitution of all or at least one value into the bound expression. Indirect proofs are permitted — to prove the existence of an object, it is sufficient to prove it impossible for one not to exist: no actual object need ever be demonstrated. Likewise, it is possible to show a statement true by showing that if it is assumed false, a contradiction may be obtained, so demonstrating that it cannot be false.

Constructive first order logic is very similar to the classical logic illustrated above, except that indirect proofs are not permitted — specifically, to prove the existential quantification of a statement, it is necessary to demonstrate that there is a suitable object which satisfies the statement, and it is impossible to prove something true simply by assuming its negation and obtaining a contradiction.

A three-valued logic such as that described in [39], has three possible truth values — true, false, and unknown represented by `T`,`F` and `*` in Jones' notation. The connectives are computable — `*` represents the failure of an argument to a connective to terminate. As can be seen from the tables of truth values (Figure 6.2) the connectives $\wedge$ and $\vee$ are also commutative: the computability of the connectives, requires the arguments to be evaluated in parallel. In this logic, the deduction theorem, that $\Gamma, A \vdash B$ is equivalent to $\Gamma \vdash A \rightarrow B$ does not hold. However, the three-valued logic does have the advantage of being intuitive to programmers, who are used to computable boolean operators in programming languages, and to the concept of non-termination.

An alternative to this approach is to use the boolean operators found in functional languages directly, and then to justify the approach in terms of their non-computable counterparts in either classical or constructive logic. This avoids confusion between the characteristics of object-level and meta-level connectives, since their properties, which are not classical (or constructive), are understood by programmers. The constructors here are computable, but non-commutative (see Figure 6.3), but may be expressed in terms of non-computable, commutative operators (such as those given in Figure 6.1) by use of a suitable translation. For example, the operator `|` is given the translation into classical logic

$$\| \texttt{ p | q } \| \Longrightarrow \| \texttt{ p } \| = \texttt{True} \vee (\| \texttt{ p } \| = \texttt{False} \wedge \| \texttt{ q } \| = \texttt{True})$$

F→F	T	F∧F	F	F∨F	F	¬F	T
F→T	T	F∧T	F	F∨T	T	¬T	F
F→*	T	F∧*	F	F∨*	*	¬*	*
T→F	F	T∧F	F	T∨F	T		
T→T	T	T∧T	T	T∨T	T		
T→*	*	T∧*	*	T∨*	T		
*→F	*	*∧F	F	*∨F	*		
*→T	T	*∧T	*	*∨T	T		
→	*	*∧*	*	*∨*	*		

Figure 6.2: Interpretation of connectives for a three-valued logic

F→F T	F∧F F	F∨F F	¬F T
F→T T	F∧T F	F∨T T	¬T F
F→⊥ T	F∧⊥ F	F∨⊥ ⊥	¬⊥ ⊥
T→F F	T∧F F	T∨F T	
T→T T	T∧T T	T∨T T	
T→⊥ ⊥	T∧⊥ ⊥	T∨⊥ T	
⊥→F ⊥	⊥∧F ⊥	⊥∨F ⊥	
⊥→T ⊥	⊥∧T ⊥	⊥∨T ⊥	
⊥→⊥ ⊥	⊥∧⊥ ⊥	⊥∨⊥ ⊥	

Figure 6.3: Interpretation of connectives for an interpreted logic

where $\|e\|$ means the value denoted by the functional expression e, and all connectives are given their standard classical (or constructive) meanings. Since no quantifiers appear in functional languages, no quantifiers are considered to be in the logic, although, as is normal in functional languages, all free variables in all statements are assumed to be universally quantified. Existential quantification may be expressed in a fashion similar to that for constructive logic, by demonstrating a particular value for which the statement holds, which suggests that this logic may be closer to constructive than classical logic.

The choice made for interpreting the set of truth values and connectives was to use the functional connectives directly, defining them by appealing to an interpretation into classical or constructive logic. No fixed set of axioms or proof rules is defined — rather a mechanism is provided for defining any desired axioms and rules. If these rules and axioms are constructively valid, then the user may be considered to be working in constructive logic; likewise, if they are valid in classical logic, the user may be considered to be working in classical logic. Both of these are likely: rules not valid in classical logic are unlikely to be used; the only classical rules not present in constructive logic concern the excluded middle, which is not generally applicable when computable connectives are being used, and proofs involving existential quantification, which is not available as a functional connective: a statement of the form $\forall x \exists y p(x, y)$ must be expressed as $p(x, f\ x)$ where f is a function which demonstrates a suitable y in a constructively valid fashion. The termination characteristics of the pattern matching may be determined by examining the HESF code generated by the algorithm in Chapter 3, and observing whether any attempt is made to head-

evaluate a non-terminating argument.

Another consequence of this choice concerns proofs about infinite values. One way of proving a result $p(xs)$ when xs is an infinite value such as `[1,2,3,4,5...]` is to prove $p(xs)$ inductively for all its partial approximations $\bot$, `1:`$\bot$, `1:2:`$\bot$, ... and then deduce the result for the limit. Unfortunately there are some statements, such as $\exists n.$`drop n xs`$= \bot$, which may be true for all the approximations, but not for the limit. A statement is called *admissible* if its truth for a sequence of approximations implies its truth for the limit – Paulson ([60], section 7.3.6) gives syntactic rules which ensure that statements are admissible. The avoidance of quantifiers suggested above may simplify this issue by making it possible to construct a system in which all statements are admissible.

The aim of this is not to outlaw the use of quantifiers, which may be added by the incorporation of rules into the system; rather the aim is to investigate how far it is possible to go without them, and whether a simpler and more uniform system can be constructed as a result.

6.2 Proof Presentation Styles

Once the logic has been chosen, the layout of the proof still needs to be determined. This is done by examining various styles and comparing them on three points of information presentation: how references to other statements, upon which a given statement depends, are given; how assumptions are traced through a proof, particularly where they are introduced and discharged; and what details are missing from each of the proof steps — for example, exactly how a rule is being applied.

To aid the presentation of the various styles, each has been used to prove a simple property of binary trees — that reflecting a tree does not alter its size. For this purpose, the following types and functions have been defined. Note that for simplicity, the type has been defined strictly, so there are no partial or infinite values, and that none of the function cases overlap.

```
sdata trees = Nil | Tree trees ints trees

sizeof Nil = 0
sizeof (Tree t1 i t2) =
    1 + sizeof t1 + sizeof t2

mirror Nil = Nil
mirror (Tree t1 i t2) =
    Tree (mirror t2) i (mirror t1)
```

and the property, defined using the keyword `property`, is:

```
property mirror_sizeof t = sizeof (mirror t) = sizeof t
```

6.2.1 Block Structured Proof Style

Here, the proof is simply a sequence of statements. The initial statements are the axioms, deduced from the function definitions, together with axioms from other theories, for example, number theory, and previously proved properties.

The remaining statements are given justifications which state how they may be verified in terms of previous statements. At any point an inner block may be introduced, which permits the introduction of assumptions, which are only in scope inside the block and are discharged when the result of the proof block is used. The notation here is copied from [39]

Justifications are presented to the right of the statements. The statements used in a justification are listed along with the justification — each statement in the proof is given a reference number. The dependence of statements on assumptions is highlighted by the blocks which introduce the assumptions. The justifications lack details on the precise way in which the statement relates to those used to justify it. This could be overcome by making the rules more specific.

```
A1 sizeof Nil = 0                                          AXIOM
A2 sizeof (Tree t1 i t2) = 1 + sizeof t1 + sizeof t2       AXIOM
A3 mirror Nil = Nil                                        AXIOM
A4 mirror (Tree t1 i t2) = Tree (mirror t2) i (mirror t1)  AXIOM
A5 x + y = y + x                                           AXIOM

 1 sizeof (mirror Nil) = 0                                 EQ A1,A3
 2 sizeof (mirror Nil) = sizeof Nil                        EQ A1,1
 3 from h1, h2
    h1 sizeof (mirror t1) = sizeof t1                      HYPOTHESIS
    h2 sizeof (mirror t2) = sizeof t2                      HYPOTHESIS
   3.1 sizeof (mirror (Tree t1 i t2)) =
         sizeof (Tree (mirror t2) i (mirror t1))           APPLY A4
   3.2 sizeof (mirror (Tree t1 i t2)) =
         1 + sizeof (mirror t2) + sizeof (mirror t1)       EQ A2, 3.1
   3.3 sizeof (mirror (Tree t1 i t2)) =
         1 + sizeof (mirror t1) + sizeof (mirror t2)       EQ A5, 3.2
   3.4 sizeof (mirror (Tree t1 i t2)) =
         1 + sizeof t1 + sizeof (mirror t2)                EQ h1, 3.3
   3.5 sizeof (mirror (Tree t1 i t2)) =
         1 + sizeof t1 + sizeof t2                         EQ h2, 3.4
   3.6 sizeof (mirror (Tree t1 i t2)) =
         sizeof (Tree t1 i t2)                             EQ A2, 3.5
    infer 3.6
 4 sizeof (mirror t) = sizeof t                            INDUCT(t) 2,3
```

The block at step 3 proves the sequent:

```
{ sizeof (mirror t1) = sizeof t1, sizeof (mirror t2) = sizeof t2 } ⊢
    sizeof (mirror (Tree t1 i t2)) = sizeof (Tree t1 i t2)
```

and the induction at step 4 discharges these inductive hypotheses.

6.2.2 Tree Structured Proof Style

In the tree-structured style, each statement is placed vertically adjacent to its antecedents: in this way, the connections between the statements involved in the justifications are made clear. The proof is written as tiers of statements, each being deduced from those written directly above. Assumptions are introduced and discharged at various points in the tree, and it is often unclear which

assumptions apply to which statements. Statements which are needed in more than one place must generally be proved on each occasion. Finally, the representation becomes unwieldy beyond a certain size of proof, and a proof needs to be broken down into manageable sections.

```
M N = N    S N = 0
---------------------- EQ
      S (M N) = 0      S N = 0
      -------------------------- EQ
         S (M N) = S N
         ----------------
                φ


          M(T t1 i t2) = T (M t2) i (M t1)
          ------------------------------------ APPLY
          η = S(T (M t2) i (M t1))
          -------------------------- EQ
x+y=y+x η = 1+S(M t2)+S(M t2)
---------------------------------- EQ
           η = 1+S(M t1)+S(M t2)    [S(M t1) = S t1]
           ------------------------------------------------ EQ
[S(M t2) = S t2]  η = 1+(S t1)+S(M t2)
---------------------------------------- EQ
              η = 1+(S t1)+(S t2)     S(T t1 i t2) = 1+(S t1)+(S t2)
              ------------------------------------------------------ EQ
                                        ψ

      φ       ψ
------------------ INDUCTION
  S(M t) = M t
```

The steps in this proof are the same as previously, but arranged differently. Some steps which were arranged sequentially above are now laid out horizontally, while others appear vertically. The steps which previously appeared inside an inner block are no longer set off in any special way. The same level of justification detail is missing from both proofs.

Either of these styles may be presented in an inverted goal-directed fashion, whereby the statement to be proved appears at the top, and is broken down into subsidiary statements — or goals — each of which must be proved separately. In general, this renders the same proofs, except that the statements appear in the reverse order, and restrictions on the style (for example, regarding referencing of statements) are reversed. This goal directed style seems more natural for the creation of some kinds of proof, since the statement to be proved is well known, and may be easily broken down. In practice, both forward and backward construction techniques are used, and this is independent of the style of presentation.

6.2.3 Transformational Proof Style

A system for transforming programs written in an algebraic style is given in [12] with the intent of transforming correct but inefficient programs into efficient

programs. It permits the defining and manipulation of functions and expressions. This suggests itself as the basis of a proof system, since many of the operations performed in functional language proofs come under the description of transforming expressions in this way.

In general, cross-referencing is not an issue — most of the steps involve simply the previous step, and external information, such as function definitions and theorems (or laws). When other references are required, the individual statement to be referenced may be individually named and then referenced when required by the name given. Non-transformational rules (such as induction) tend to be included rather haphazardly, although the proofs could be organised in a more structured fashion by lessening the importance of the ideas of transformation. The rules tend to include most relevant information, except when a mathematical law is being applied, when the way in which the law is to be applied is generally unspecified.

The intention in this proof system is to transform one half on an equation into the other — where this is not practical, the entire equation may be transformed into the expression `True` instead.

```
sizeof (mirror t)                                DEFINE property

sizeof (mirror Nil)                              INSTANCE t/Nil
sizeof Nil                                       UNFOLD mirror
                                                 NAME BASE

sizeof (mirror t)                                USE property
sizeof (mirror (Tree t1 i t2))                   INSTANCE t/Tree t1 i t2
sizeof (Tree (mirror t2) i (mirror t1))          UNFOLD mirror
1 + sizeof (mirror t2) + sizeof (mirror t1)      UNFOLD sizeof
1 + sizeof (mirror t1) + sizeof (mirror t2)      LAW x+y = y+x
1 + sizeof t1 + sizeof (mirror t2)               HYP mirror t1 = t1
1 + sizeof t1 + sizeof t2                        HYP mirror t2 = t2
sizeof (Tree t1 i t2)                            FOLD sizeof
                                                 NAME INDUCTIVE

sizeof (mirror t)                                USE property
sizeof t                                         LAW induction(t)
                                                     BASE, INDUCTIVE
```

This style is clear and easily constructed for simple proofs — in general, the construction requires just the repeated application of a small set of rules. The two main disadvantages are the cavalier approach to the introduction and discharge of assumptions, and the inconsistent approach to incorporating non-transformational rules. Transformations produce only partially correct results and separate proofs of termination have to be added to make the approach rigorous, particularly for recursive definitions.

6.2.4 Functional Proof Style

The desire for a notation easily accessible to functional programmers led to an attempt to produce a notation for proofs in a functional style. The two important parts in the exposition of proofs are the references between statements and the introduction and discharge of assumptions. Proof rules may be expressed as

functions mapping statements and assumptions to a new statement. A complete proof consists of a compound function of this sort, built up from functions representing the individual steps of the proof. The final proof somewhat resembles the tree style of Section 6.2.2, in that the function is tree-shaped. References are handled by "naming" each statement with a suitable function definition. Linear sections of proof may be represented by compound functions of simple rules, without any named functions involved. In the example below, expressions surrounded by { ... } are object-level expressions — other expressions are at the meta-level.

```
sm ({sizeof (mirror t) = sizeof t},[])

sm =
  struct_induct {t} (sbase,sind) [subst {t} {t1}, subst {t} {t2}]

sbase =
  sequent [{mirror Nil = Nil}, {sizeof Nil = 0}]

sind =
  eq {mirror (Tree t1 i t2) = Tree (mirror t2) i (mirror t1)} .
  eq {sizeof (Tree t1 i t2) = 1 + sizeof t1 + sizeof t2} .
  eq {sizeof (mirror t2) + sizeof (mirror t1) =
              sizeof (mirror t1) + sizeof (mirror t2) } .
  eq {sizeof (mirror t1) = sizeof t1} .
  eq {sizeof (mirror t2) = sizeof t2} .
  eq {1 + sizeof t1 + sizeof t2 = sizeof (Tree t1 i t2)}
```

The first line of this proof expresses the property as a proof rule applied to the property expression and no assumptions. Before the function is evaluated, the statements known to be true from the object program's function definitions are added to this list of assumptions. The function `sm` is defined as a partial application of the rule for structural induction. An argument to `sm` is passed directly to the function `struct_induct`. The induction rule takes an object argument, a pair of functions to handle the base and inductive cases respectively, and a list of assumptions for the inductive case. These are both introduced and discharged in the induction rule, and are applicable only to the statements proved by the function handling the inductive case. They are checked by the induction rule to ensure that they represent valid inductive hypotheses. The assumptions are curried functions to be applied to the argument to the structural induction function. In general, as here, they will be substitutions for the inductive variable.

The base case is proved by a simple sequent — replacing appropriate terms in the expression produces a decidably true statement. The elements of the list presented as the first argument to sequent must be found in the list of assumptions to the function sequent. The inductive case is proved, as in the block structured example, by a chain of equality substitutions.

There are two major problems with this approach: firstly, although the representation is functional, it is not a clear presentation of the proof; secondly, although evaluation of the meta-program would seem sufficient, it is non-trivial to design an algorithm which can automatically verify this proof in a guaranteed

sound fashion since, for example, a rule could easily be defined not to use the passed arguments, but rather to prove a different property to the one desired.

6.2.5 Proof by Evaluation

One other method of proof studied deserves mention in addition to the others described above, which is important in the development of the ideas in this section. `Resolution` is an automatic method of proof used for establishing the truth of relations in the programming language PROLOG. It was found to be unsuitable for proving properties of functional programs, mainly because of its automatic nature — it cannot deal with free variables much better than can the standard functional language evaluation algorithm. An attempt was made to extend the idea of resolution to allow greater control of the process, but this was abandoned when it became clear that it would be more sensible to attempt the extension to the standard functional evaluation algorithm.

This presents a new view of proof — the idea that proof is just a controlled evaluation of a property. This extends to give additional means of encouraging programmers to engage in proof. A range of tools is provided in the workbench for use by the programmer. Initially, the programmer constructs a program and executes it, testing it in the usual way. A desired property of the program is expressed, and this too is tried on a range of arguments. If the property is determined to be false, changes are made to the program or to the property, until it appears to be true. A debugger may be used to assist in this process. Once the property is considered likely to be true, a proof may be attempted, based on a trace of the execution of the property obtained from the debugger: the step-by-step evaluation of a functional program resembles the desired structure of the proof, with changes being made to enable free variables to appear in the proof rather than more concrete values. Specifically, pattern matching needs to be replaced with an argument which covers all possible cases; recursion needs to be replaced with an inductive argument; and some of the steps rearranged and regrouped to make for a more logical and structured presentation (something lazy evaluation rarely is). The evaluation technique used depends on the object language. It will be assumed here that the technique used is that described in Chapter 3.

The presentation of this style is goal-directed and block-structured. References between one statement and another are made by explicitly writing the referenced statement. Its truth may be determined by looking in a list of facts known to be true, and assumptions currently in force. Inner blocks may be used either to provide proofs for subsidiary goals, or to add facts to the list of those known to be true, in preparation for a subsequent step. Assumptions may be introduced at the start of a block, but if so must be discharged when the result is used (this is equivalent to the deduction theorem).

The proof itself consists of the "evaluation" steps to be performed. Goal-directed proof was chosen since this more naturally reflects what a programmer expects from evaluation — that a more complex expression is reduced into one or more less-complex expressions. The rules give sufficient information to transform the expression into a simpler one, but only at the beginning is the property actually cited. Most of the evaluation steps are reversible, so proofs of equality follow in both directions.

```
mirror_sizeof t <==
  sizeof (mirror t) = sizeof t

  block_to_prove
    sizeof (mirror Nil) = sizeof Nil <==

    unfold (mirror Nil) by (mirror Nil = Nil)

  block_to_prove
    sizeof (mirror (Tree t1 i t2)) = sizeof (Tree t1 i t2) <==

    unfold (mirror (Tree t1 i t2)) by
        (mirror (Tree t1 i t2) = Tree (mirror t2) i (mirror t1))
    unfold (sizeof (Tree (mirror t2) i (mirror t1))) by
        (sizeof (Tree t1 i t2) = 1 + sizeof t1 + sizeof t2)
    commute_addition (sizeof (mirror t2) + sizeof (mirror t1))
    use_equality (sizeof (mirror t1) = sizeof t1) in
        (sizeof (mirror t1))
    use_equality (sizeof (mirror t2) = sizeof t2) in
        (sizeof (mirror t2))
    fold (1 + sizeof t1 + sizeof t2) by
        (sizeof (Tree t1 i t2) = 1 + sizeof t1 + sizeof t2)

  structural induction over t :: trees, using Nil, Tree t1 i t2
```

This method of proof is examined in greater detail in the next chapter.

Chapter 7

The Language of Proofs

Chapter 2 introduced the functional language Flair as a simple subset of the language Haskell. We assume here that programs written in Flair have been processed in such a way that the defining equations of the functions have been made separate (as explained in Chapter 3 and Section 5.7), that the standard Hindley-Milner type algorithm has been applied to deduce the types of identifiers in the program, and to ensure that it is type-correct, and that any processing involving the extended types system of Chapter 5 has been performed.

This chapter describes two further extensions to the language: Sections 7.1 and 7.2 introduce a mechanism which allows proofs to be represented and stored alongside programs; while Section 7.3 introduces a mechanism allowing the definition of rules to be used in these proofs. Together these two extensions permit the representation of a completely rigorous proof system applicable to functional language programs.

This chapter details mechanisms for representing, storing, and automatically checking proofs: it does not discuss how such proofs may be created or examined by a user or programmer. Considerations for interfacing this proof system with the environment described in Chapter 4, enabling user interaction with proofs is detailed in Chapter 8. Chapter 9 covers the meaning of proofs, particularly concerning the validity of proof rules as defined in Section 7.3.

7.1 The Structure of Proofs

Chapter 6 introduced the ideas of forward and goal-directed proof. In a forward proof, each statement in a sequence follows, according to some specified proof rule, from its antecedents, that is, the statements coming before it in the sequence. Ultimately, the justification of the earliest statements depends either on their obvious truth, or their truth being determined externally to the proof (for example, obtained as axioms or previously proved properties). A goal-directed proof consists similarly of a sequence of statements, each of which is a goal to be proved and is reduced to sub-goals, again according to some proof rule. Each statement thus follows from subsequent statements, and the objective is to reduce goals to statements which are clearly true: either directly, or asserted by some external theory.

The mechanism presented in this chapter for the storage and manipulation

of proofs does not contain an explicit statement of which proof rules may be used for a particular application, but tries to provide a scheme within which a wide variety of proof rules, couched in either forward or goal-directed styles, may be provided for use, and indeed, many rules may be provided, if desired, in both forward and goal-directed styles.

This scheme is provided by defining a proof to consist of a sequence of steps, and introducing the notion of a current context between each pair of steps in a proof. This context consists of a current goal and a database of known facts. Each proof step may be either goal-directed, in which case the current goal is reduced to a simpler one using the current database of known facts, or else, in the forward style, a step may add a fact to the current database, usually without changing the goal. Conceivably, a rule could simultaneously change the goal and the known facts, but no such rules are considered here.

For a given step to be valid, it must be possible to show that a truth-preserving relationship exists between the context prior to the step and the context after it. Denoting the prior context by `(g,F)` where `g` is the prior goal and `F` the prior database of known facts, and the consequent context by `(g',F')`, the two basic cases may be considered. For a goal-directed step, such as that shown in Figure 7.1, the consequent goal, `g'`, together with the known facts (which do not change), must imply the prior goal `g`.

A forward style proof step is valid, provided that the new facts can be deduced from the existing facts. Assuming that the new database `F' = F*f`, where `*` is the operation of extending the fact database, and that the goal `g'` is identical to the prior goal `g`, we have that `F⊢f`, as illustrated in Figure 7.2.

The transformation style of proof, described in Section 6.2.3, in which an equality may be proved by the successive transformation of one side into the other is provided by allowing a sequence of (reversible) steps to transform one goal into another (equal) goal. Such a transformation step may be pictured as in Figure 7.3.

Any step may have a subproof (or sequence of subproofs) attached to it as one or more inner blocks. A subproof is used in establishing the correctness of a step, and allows assumptions to be added temporarily to assist the proof, thus allowing the "natural deduction" style of proof. Such a subproof may be added to any of the above steps, but Figure 7.4 illustrates a subproof attached to a forward step, assisting in the proof of a new fact.

With such a range of mechanisms with which to structure proofs, there is often a choice as to how any particular kind of proof rule should be presented. For example, consider the rule to prove a property by induction over the natural numbers, `s(n)`. This must first be established for the base case, `s(0)`, and then for the inductive case that `s(k)` → `s(k+1)`. For this, two subproofs are required: one for each of the cases. The induction rule may be presented either in the forward style, where each case is proved by a forward step together with a subproof, and then the induction step can extract both from the database, as illustrated in Figure 7.5, or in the goal-directed style, in which the main goal is initially reduced to its two component subgoals, each of which is then proved in an inner block, as shown in Figure 7.6. This choice of styles for proof rules is analogous to the choice to be made when writing a functional program between separating out support functions and giving them top-level definitions, or including them with the function definition where they are used inside a `let` or `where` construct.

```
(g,F)
    STEP   valid if  F, g' ⊢ g
(g',F'=F)
```

Figure 7.1: Goal-directed proof step

```
(g,F)
    STEP   valid if  F ⊢ f
(g'=g,F'=F*f)
```

Figure 7.2: Forward style proof step

```
(g,F)
    STEP   valid if  F ⊢ g = g'
(g',F'=F)
```

Figure 7.3: Transformation style proof

```
(g,F)
    STEP, Proof p    valid if p a proof that F ⊢ f
(g'=g,F'=F*f)
```

Figure 7.4: Forward Step with Subproof

```
(g,F)
    FORWARD STEP with subproof p, p ⊢ s(0)
(g,F*s(0))
    FORWARD STEP with subproof q, q ⊢ s(k)=>s(k+1)
(g,F*s(0)*s(k)=>s(k+1))
    INDUCTION STEP
(g,F*s(0)*s(k)=>s(k+1)*s(n))
```

Figure 7.5: Natural number induction in the forward style

```
(s(n),F)
    INDUCTION STEP, subproofs p,q
           p ⊢ s(0)
           q ⊢ s(k) => s(k+1)
(True,F)
```

Figure 7.6: Natural number induction in goal-directed style

```
property mirror_sizeof t = sizeof t == sizeof (mirror t)

mirror_sizeof t <==

{ g = sizeof t == sizeof (mirror t), F = program facts }
```

Figure 7.7: Statement and Start of Proof of Property

One indirect advantage of this approach, combining both forward and goal-directed styles, is that a partial proof, in which some forward steps, and some goal simplifications, have been done, can simply be represented as an initial segment of a proof — reducing the need for proofs with "holes" in them.

7.2 The Representation of Proofs

This section presents a syntax for the textual representation of proofs using the mechanisms given above, specifically for the structures of proof steps and subproofs. This syntax is not intended to be particularly readable, since it is assumed that examination and manipulation of proofs will take place via a suitable interface such as that described in Chapter 8. Rather, this syntax is supposed to allow the proof to be represented in a reasonably compact form, which may be easily stored, transmitted, and processed by computer.

7.2.1 Representing a Proof Step

Each proof step represents a transformation from one context to another. The initial context is defined at the start of a proof by establishing the property to be proved, which is the initial goal, and the fact database is extracted from definitions and declarations in the program, together with previously proved properties, and axioms from other theories. The fact database is never stated explicitly in the proof, and the goal is only stated initially: each proof step describes explicitly the transformation between the context prior to the step and the context after the step. In this way, each proof rule appears as a partially applied function, still to be applied to its prior context in order to generate its subsequent context.

Each proof is introduced by giving the name of the property to be proved, together with its free variables, followed by the special symbol `<==`. The initial goal is then the property referred to; the initial fact database is all the appropriate information as described above. This gives rise to the situation as in Figure 7.7.

Simple rules allow for a variety of proof operations to be carried out, for example, the substitution of a free variable in the goal. These are written as the rule name followed by appropriate arguments, for example, the variable to be substituted, and the expression to be substituted for it. The expression for substituting `Nil` for `t` in the goal of Figure 7.7 is illustrated in Figure 7.8.

When representing a proof in this way, two types of symbols occur: object-level symbols such as `t` and `Nil`, and proof-level symbols such as substitute. Confusion between these must be avoided in a fashion consistent with the object

```
{ g = sizeof t == sizeof (mirror t), F = program facts }
  substitute ‘t ‘Nil
{ g = sizeof Nil == sizeof (mirror Nil), F unchanged }
```

Figure 7.8: The Proof Rule Substitute

```
{ F contains x=0, 0=y }
  equality (‘x ‘= ‘0) (‘y ‘= ‘0)
{ F’ = F*x=y }
```

Figure 7.9: The Forward Proof Rule Equality

language. Here a backquote is used to identify all object-level symbols. When proof rules are being defined (see Section 7.3) a greater distinction is made between the two, but this is less clear and more verbose than quoting. Such distinctions are not however necessary for the user, who should understand what is meant by each proof rule, and so the quoting need not be apparent when the proof is displayed to the user for examination or modification.

A forward proof rule may be expressed in much the same way as the goal-directed rule substitute above. The proof rule equality takes two facts, each of which is an equality with the same right hand side, and, if it can find the facts in the current database, adds the fact that the two left hand sides are equal. This is illustrated in Figure 7.9. The ordering of the equalities in the fact database is not relevant to the working of the rule.

In addition to storing expressions, the fact database contains two further collections of information: all the information gathered about the types in the program, including the definitions of types and the types of the various function and variable identifiers; and a list of identifier names indicating which identifiers are free for substitution. Most of the type information is stored in a fashion identical to that for storing facts, that is, as parsed expressions. The type information is accessed as needed by the proof rules, generally in a supporting role, for example, to check that an argument by cases has indeed covered all the cases of a variable. The type information, in particular the very detailed type information gained from the use of extended types as described in Chapter 5, may play a more central role in the proofs, for example, it is possible to prove that a function terminates by showing that its result is always in a strict type. If it is possible to show that whenever the arguments to a function are of a strict type, then so is the result, it may be shown that the function terminates on all finite, total, data values. The information on variables free for substitution is used in two ways: it enables free variables to be distinguished from constants (such as defined functions); and it permits variables to be fixed in inner blocks when a free variable takes on a specific notional value.

How this information is stored in the database is not described here: the database is viewed as an object of an abstract type, accessed only through functions such as “add a fact to the database” and “check whether this variable is free”. It is assumed that functions exist which make it possible to check whether any given datum is, or is not, in the database; to recover some or all of the data of a given kind; and to identify which facts are defining equations of

```
{ g = sizeof Nil == sizeof (mirror Nil) }

      unfold @r1

{ g = sizeof Nil == sizeof Nil }
```

Figure 7.10: Unfolding a term of the current goal

functions, and which are just properties. In general, all the details of access to the fact database are obscured behind the generalities of the proof rules, which concentrate on the important points of each step, that is, which equations and variables are needed to prove a particular step.

Goal-directed proof steps sometimes act on the entire expression, as in the case of substitute above. However, it is often the case that a rule will be desired to act on only a part of the goal, for example, expanding a term of the current goal according to an equation in the facts database. For example, in order to replace `mirror Nil` by `Nil` in the resultant goal of Figure 7.8, it is necessary to specify that `mirror Nil = Nil` is the appropriate expression to apply to the first argument of the expression to the right hand side of the top level expression. To do this in a consistent way, any subterm of an expression may be referenced by an "expression specifier" which is essentially a method of navigating the expression tree.

All expression specifiers begin with the symbol `@`. On its own, this represents the entire current goal. If followed by a single-digit number, this represents an argument position in the expression, where `0` is the function symbol, `1` is the first argument, and so on. If an operator is written infix, `0` is the operator, `1` is the leftmost argument, etc. For binary operators, the letters `l` and `r` may be used for the left and right hand arguments of the operator respectively: these are in fact, exactly identical to `1` and `2`. If a function should have more than nine arguments, the subsequent arguments may be referenced by writing the appropriate multi-digit number in angle brackets, e.g. `<13>`. Again, the proof assistant should provide adequate means for correctly identifying any desired sub-expression, thus relieving the user of dealing with this notation directly.

Figure 7.10 illustrates the use of the unfold rule on the resultant goal of Figure 7.8 to replace `mirror Nil` with `Nil`. `unfold` takes the left hand side of a defining fact, and recovers the right hand side from the facts database.

7.2.2 Cross-Referencing

In traditional forward style proof, such as the block-structured style of Section 6.2.1, each statement is numbered and the justification of each step refers to the numbers of those statements from which it follows. This is appropriate for short, self-contained proofs, but is less appropriate for longer proofs, and is inappropriate for the kinds of proofs described here, in which statements need to refer not only to other statements, but to information abstracted from the program under consideration, together with a database of properties, for example, from theories appertaining to the data being considered.

In general, facts are stored in a database and may be extracted using one of several ways of searching to match one or more facts against a description. This

description will often be a complete statement of the fact, exactly as it appears in the database, for example, the fact `x=0` from Figure 7.9. Alternatively, it may be approximately that given in the database, for example, the fact `y=0` in Figure 7.9. Another possibility is that only a small amount of information is given, but the search is greatly restricted — for example, limited only to defining equations — so that the information given is sufficient to recover the entire fact from the database. Previously proved properties may be recovered by citing the name of the property rather than the expression of the property, for example, using the name `mirror_sizeof t` in place of a full statement of the property.

It is not always necessary for the user to provide the facts to be checked in the database. In a complex proof rule the statements to be checked may be automatically generated by the proof rule, and these checked for inclusion in the database. For example, the forward version of natural number induction, `nat-induct s(n)`, can automatically generate the two statements `s(0)` and `s(k)=>s(k+1)` required for valid natural number induction, and check for their presence in the database of known facts before adding the new fact `s(n)`.

7.2.3 Inner Blocks

Inner blocks are attached to individual proof steps and contain subproofs. They are introduced with a syntax analogous to that for introducing top-level proofs of properties, that is, a rule together with arguments is presented, followed by the special symbol `<==`, and then subsequent lines contain an indented inner proof. If more than one subproof is to be attached to a single proof step, then the symbol `<==` must appear between the subproofs.

The way in which an inner block is used depends upon the rule to which it is attached. The rule provides the inner block with its own distinct (and usually different) context: frequently assumptions may be added for the duration of the inner block, which are placed in the inner fact database, but not in the outer. Once a subproof is complete, the initial and final inner contexts may be used to modify the outer context. For example, a skeleton proof of goal-directed natural number induction is presented in Figure 7.11. The initial goal `s(n)` is proved using natural number induction over `n`, which has two cases, and one subproof corresponding to each case. The first is the proof for the base case, `s(0)`; the second is for the inductive case. In the inductive case, the assumption `s(k)` is added to the database of facts, `k` is fixed so it may not be used in substitution, and the goal is set to be `s(k+1)`. If both goals are successfully reduced to `True`, then the outer goal may be likewise reduced by the induction law.

7.2.4 Extended Example

Section 6.2.5 gave an outline proof of the property `mirror_sizeof t` in this style. This is reworked here in detail, and the complete proof appears in Figure 7.12. This proof uses the forward structural induction rule over the type `trees`. This requires all the cases for the type `trees` to be covered, and suitable facts to be included in the database. The two subproofs prove the cases `Nil` and `Tree t1 i t2` respectively. Since `trees` is a strict type, a case proving $\perp$ is not required. Each subproof consists of a sequence of reversible steps for translating the expression `sizeof (mirror t)` into `sizeof t`, after the appropriate

```
{ g = s(n) }
  nat-induction `n <==
    { g' = s(0) }
      ...
    { g'' = True }

      <==
    { F' = F*s(k) }
    { F'' = F*fix k }
    { g' = s(k+1) }
      ...
    { g'' = True }

{ g' = True }
```

Figure 7.11: Goal-directed natural number induction

substitution for `t`. The second block makes use of the two inductive hypotheses for the non-empty case, which are stored along with the resulting equivalence in the fact database, and are discharged when the induction rule is applied. The final step of the outer proof checks that the current goal is `True`.

The rule `unfold`, as described above, replaces the left hand side of the appropriate definition with its right hand side, operating on the specified subexpression of the current goal. The rule `add_comm`, applied to a sub-expression of the current goal, checks that this sub-expression is of the form `x+y`, and replaces it with the expression `y+x`. The rule for equality substitution, `eq`, finds an equation in the fact database, `l=r`, one of whose sides matches the cited sub-expression, and then replaces the subexpression with the other side of the equation. The rule `def_fold` takes two arguments: the sub-expression to be considered, and the left-hand side of a definition, whose right hand side corresponds to the given sub-expression, that is, if the second argument to `def_fold` were to be unfolded, the result would be the cited sub-expression.

The rule for structural induction takes an expression and a variable in the expression over which to induct. It examines the fact database for the appropriate type definition, and then checks that all the cases have been covered, generating inductive hypotheses as necessary to discharge assumptions in the inner blocks. If valid, it adds the equivalence of the initial and result expressions to the fact database.

The rule `comp_id` checks that two halves of an equality expression are computably identical: to do this, both sub-expressions have to be the same, and the fact database must contain sufficient type information to deduce that the expressions to be used will terminate. Here, this is guaranteed by the assertion that the trees are finite.

7.3 Defining Proof Rules

The proof rules are defined as context-transforming functions. Each is passed arguments, most of which are determined by the needs of the function: it is

```
mirror_sizeof t <==

    block_to_prove (‘sizeof (‘mirror ‘Nil) ‘== ‘sizeof ‘Nil) <==
        unfold @l1
        unfold @l
        unfold @r
        comp_id

    block_to_prove_hyp
        (‘sizeof (‘mirror (‘Tree ‘t1 ‘i ‘t2))
              == (‘sizeof (‘Tree ‘t1 ‘i ‘t2)))
        [  (‘sizeof (‘mirror ‘t1) ‘== ‘sizeof ‘t1),
           (‘sizeof (‘mirror ‘t2) ‘== ‘sizeof ‘t2) ] <==

        unfold @l1
        unfold @l
        add_comm @lr
        eq @lrl (‘sizeof (‘mirror ‘t1) ‘== ‘sizeof ‘t1)
        eq @lrr (‘sizeof (‘mirror ‘t2) ‘== ‘sizeof ‘t2)
        def_fold @l (‘sizeof (‘Tree ‘t1 ‘i ‘t2))
        comp_id

    struct_induct @ [‘t]
    assert
```

Figure 7.12: Complete Proof of `mirror_sizeof t`

```
P <==
  A p q
  B r s t
  C v
  D x y

proof = D x y [] (C v [] (B r s t [] (A p q [] (g,F))))
```

Figure 7.13: Proof as a Compound Function

these arguments which have been described above for each rule. However, each is also passed two additional arguments, not expressly indicated in the proof step. The first of these is a list of inner subproofs, which is usually empty; and the second is the context before the rule is executed. The result of a proof rule function is a new context. An entire proof can be viewed as each proof step applied to the resultant context of the previous step as pictured in Figure 7.13 (which is similar to the idea of functional proof of Section 6.2.4). Checking a proof thus consists of creating an initial state and applying the appropriate compound function, representing the entire proof, to this state, then checking that the final state is satisfactory. The language used to define these proof rules is intended to be similar to the object language being used, although definitions of proof rules are kept separate from other parts of an object program. Some special types, functions and conventions are included in the language as required by the proof rules.

It is assumed that any proof rules included in the proof system will be sound. This is discussed fully in Chapter 9. It is assumed further that the design of proof rules is a specialised activity (partly because of the necessity of checking soundness) and that, in general, programmers will not write their own proof rules, but will have access to a large library provided by specialists.

7.3.1 The Language of Proof Rules

Proof rules are defined in a functional language, and so the standard object language needs to be enriched by the addition of some data types, functions and conventions. The most important data types are those to handle expressions, literals, specifiers, proofs and contexts. `Expressions` are represented as trees of literals, where a literal is an identifier, number, or symbol in the object language (preceded by a backquote in textual proofs). `Specifiers` consist of a list of numbers after being parsed, each number indicating the argument position of the desired sub-expression in the top-level expression. `Proofs` are a function type from contexts to contexts. A context is a pair consisting of a goal expression and a fact database. Fact databases are an abstract type. `Contexts` are defined as a pair: the goal and database of known facts.

```
data expr       = Expr literal [exprs]
data literal    = Lit [char]
data specifier  = Spec [int]
type proof      = context->context
data context    = Context expr factdb
```

```
abstract factdb = ...
```

Proof rules may be mixed with other supporting functions (for example, standard functions such as `take` and `map`), but may be distinguished by their type signatures. A proof rule must have arity at least two, and the type must end `[proofs]->contexts->contexts`. Proof rules may freely call other proof rules or support functions as necessary. A subproof may be used by extracting the appropriate function from the list passed in and and applying it to a context defined within the rule. The result of this application is a new context which may be subsequently examined by the rule.

Only one extension is added to the underlying language in this design: this relates to the pattern-matching of expressions. To avoid complications in defining proof rules, whenever an argument of type `specifiers` is passed to a rule expecting an argument of type `exprs` (or if nested inside a larger expression), the appropriate sub-expression is recovered from the goal passed in with the current context and used instead. In fact, this can be arranged during compilation and type checking, so that only the actual extraction takes place at run-time, needing no modifications to the run-time code: of course, this requires modifications to be made to the compile-time code. The reverse is not permitted: if a specifier is required by a function or rule an arbitrary expression is not allowed. A function exists for the recovery of any desired sub-expression of any desired expression for use within a rule.

The fact database is an abstract type. The interface to it is through a number of functions for performing specific operations, for example, recovering a definition which matches a particular left-hand side, checking whether a variable is free for substitution, or recovering all the patterns used to define a type. These functions are always passed the fact database as their final argument, preceded by the arguments necessary for performing the operation. The database may be updated by using appropriate functions, for example, adding a new fact, which return a copy of the new database. Of course, both old and new databases may be stored alongside each other, and indeed, nested proofs usually take advantage of having two databases: one includes temporary assumptions; the other is only updated when the subproof is finished.

7.3.2 Examples

A few examples of the definition of proof rules will be presented here. These are some of the rules used in the proof of `mirror_sizeof t` in Section 7.2.4. The simplest of these is the rule `add_comm`, which takes one argument, apart from the (empty) list of sub-proofs and the initial context, which identifies the sub-expression to be considered.

```
add_comm (Spec s) [] (Context g f) =
  Context (replace_subexp g s (Expr (Lit "+") [r,l]))  f

  (Expr (Lit "+") [l,r]) = extract_subexp g s
```

This example, like the others in this section, is not presented entirely in the sublanguage: the more convenient notations for lists (such as displays) are used in place of constructor notation. The two functions `extract_subexp` and `replace_subexp`, find and replace specified parts of an expression. A failure in

the evaluation of a rule (for example, the error missing case) causes the proof to be abandoned as incorrect. Assuming that the desired sub-expression may be found, and is of the right form, the commuted form is then substituted for it in the goal in the resulting context. The fact database `f` is neither used nor modified in this rule.

The equality rule takes two arguments apart from the standard ones. The first is a specifier indicating which sub-expression is to be acted upon, the second is an equality fact to be used in modifying the sub-expression. The abstract function `check_fact` is used to ensure that the specified fact is indeed in the fact database. The function `assert` checks that its first argument is `True`, then returns its second argument.

```
eq (Spec s) (Expr (Lit "==") [l,r]) [] (Context g f) =
  assert chk (Context (replace_subexp s (Expr r)) f)

  chk =
    extract_subexp g s == l & check_fact (Expr (Lit "==") [l,r])
```

The `unfold` rule searches the fact database for a fact which is a definition and matches the given definition. It then replaces it with the matching right hand side, instantiated to correspond to the left hand side. The abstract function `get_lhsdefn` which searches the fact database returns a left and right hand side, together with an appropriate substitution. The support function `subst` performs the substitution.

```
unfold (Spec s) [] (Context g f) =
  Context (replace_subexp g s (subst r sub) f

  (l,r,sub) = get_lhsdefn (extract_subexp g s) f
```

The rule `block_to_unfold_hyp` uses a sub-proof to establish its desired equality. A new context is created for this inner block, incorporating the given hypotheses and the specified goal, and the resulting context extracted after the proof has been followed. A new fact is then added to the original fact database, and this returned with the original goal as the resultant context.

```
block_to_prove_hyp ex hs [p] (Context g f) =
  assert (innf == Expr (Lit "True") [])
         (Context g (add_fact newf f))

  (Context innf innf) = p (Context ex (add_fact_list hs f))
  newf = Expr (Lit "_deduct") (hs++[ex])
```

Chapter 8

Proof Assistant

Kernighan and Plauger [43] state that a major problem with software development is controlling complexity — minimising side-effects between modules, and restricting the detail from any one portion of the program from affecting other parts of the program. Without machine assistance, it is unlikely that programmers would regularly attempt proofs of program properties — the development of such proofs is not particularly difficult, but it is tedious — mainly because the proof, once complete, must be verified to be of any value. This is another time consuming and error-prone task, but one which can be easily machine supported.

Chapter 4 introduced the idea of a functional program development environment, and how it could be organised into a coordinated collection of cooperating tools. These tools support the spectrum of activities involved in developing functional programs, from design, through coding, testing, and debugging, to program verification and documentation. Chapter 4 only presented four tools within this environment; many more are possible. This chapter introduces a tool to assist with the verification, examination and creation of proofs of program properties. The creation of program properties assumes the presence of a debugger with the capacity to generate outline proofs from a trace of execution of a property: no such debugger is detailed here, but indications are given in Chapter 10.

Chapters 6 and 7 have presented means of proving program properties in a functional language. Specifically, Chapter 7 has given a mechanism within which proofs might be represented as a sequence of steps, each such step corresponding to the application of a proof rule to some arguments, together with an implied context at each step. A method of checking a single proof, given a suitable initial context was outlined.

This chapter attempts to draw together the ideas of Chapters 4, 6 and 7 by presenting the design for a tool within the environment to assist in the automatic verification of proofs once constructed; the examination of proofs, by presenting them in a fashion most easily understood by the user; and the creation of proofs by concentrating the user's attention on the steps to be taken, rather than on the specific details of a rule, or on the copying of information.

It is assumed that the coordinating module of the central tool is made aware of the presence of the proof assistant, and that suitable messages (for example, `check_proof`) are added to support the functions described here. Internally,

the proof assistant itself is programmed using an executor similar to that found within the coordinating module, interpreting a functional control program. This is used both to evaluate proofs, and to program the user interface. Like the coordinating module, it is assumed that the control program in the executor of the proof assistant is available for modification and configuration by the user, and that the most frequent customisations are most readily performed.

The assistant provides help in three areas: validating proofs presented by the user; displaying proofs in a form most easily comprehensible ot the user; and assisting the user in creating and extending proofs. Sections 8.1 to 8.3 discuss these activities. Sections 8.4 to 8.8 describe the design for the proof assistant.

8.1 Proof Validation

Each proof consists of a sequence of steps, each of which is a function mapping contexts to contexts. A proof may be verified by applying each step in turn to the context produced by the applications of the previous steps; the first step must be applied to a context which is generated from the program definitions. Since each proof step is defined by a function in a functional programming language, this checking can take place automatically by writing the proof as a single compound function from the initial context of the proof to the final context. This merely leaves the question of the initial context, and verification of the final context.

When given a proof for consideration, the proof assistant is also given a list of all the declarations and definitions which are associated with that proof. The proof assistant also maintains a list of previously established or asserted properties, all of which are assumed to be independent of the property being proved. The property to be proved is found, and this is made the initial goal. The proof function is then applied to this initial context.

Once this has completed, the final context is examined, and the goal extracted. If this goal is the expression `True`, then the proof has succeeded, otherwise it has failed. If the proof succeeds, the property may be added to the list of successful properties, and may be used in proofs of later properties.

The assistant provides an interactive environment for the checking of proofs. The user repeatedly specifies a property to be checked, and then the assistant attempts to check the corresponding proof. If the proof fails, the error message generally reveals what was wrong with a proof: for example, a fact could not be found, a goal was not of the correct form, etc. The examination facilities of the next section may help to trace the cause of the problem.

Once a property has been verified, it may then be kept as both a guarantee of the correctness of the program fragments involved in the property, and also as a property of the functions for use in developing other proofs. In this way, theories of various data structures and their corresponding functions may be built up hierarchically. A property and its proof, if well presented, also serves as documentation for the data types and functions involved.

8.2 Examining Proofs

The step-by-step form of proofs as described in Chapter 7 is not very clear — the current goal is only made explicit at the beginning of the proof. The rest of the current context is never made clear. The purpose of any step or sequence of steps cannot be easily determined by the user without reference to this information. But, as explained above, presenting all of this information would rapidly overwhelm any programmer by its sheer volume. Two measures are taken to make proofs more presentable.

The current context may be traced through the proof in exactly the same way as when a proof is verified. The successive application of steps produces a sequence of contexts, and any information required for the sensible display of the proof may be extracted from the context current at the beginning of any appropriate proof step.

Each step of the proof is represented by one or more lines in the displayed version of the proof. What is displayed depends on two factors: firstly, the proof rule is associated with a particular display form inside the assistant, which lays out where the various components (the rule name, arguments, goal, etc) are to go, and what words and symbols are to be inserted to make the proof read well (for example, compare the proof of Section 6.2.5 to that of 7.2.4); and secondly, special annotations within the proof allow a user to control the display of a proof. For example, if four or five steps concentrate on the transformation of one specific subexpression of the goal, then the rest of the goal need not be displayed, and an annotation may be introduced to the effect that only that subexpression should appear wherever normally the entire goal would appear. Other similar annotations might control how expressions are presented: in their normal, minimally bracketed form, in a fully bracketed form, or as a more directly parsed form.

The proof resulting from this layout procedure may then be copied into the appropriate temporary buffer inside the editor module, from whence it may be included in the editor view. This operation consists of the transmission of one or more edit messages to the coordination module — these messages being of the asynchronous variety.

8.3 Proof Development

The above sections have indicated how a proof, once constructed, may be automatically verified, and how it may be examined. Both of these features are important as quality assurance and documentation measures. However, they do not provide much help with the construction of the proof. There are three areas in which machine assistance may help with the construction of a proof: removing details from the user, reducing the risk of trivial errors; providing a possible outline of the proof from an execution trace; and by suggesting rules which might be applicable at any given step.

The most important assistance the machine may provide is taking over details of book-keeping. The user starts with a property to prove, and selects a rule to use, and notifies the machine of this choice. The machine responds with a form, requesting the arguments required for this rule, while providing access to the current context. The user fills out the form, copying data from

the context where appropriate. The machine then encodes this information into the proof as a step, issues appropriate instructions to the editor to modify the buffer, and redisplays the affected portions of the proof. The form to be used, and the means for translating the information on the form into a proof step, are stored in the proof assistant's control program, along with the functions for displaying the proof steps.

As has been suggested previously, the environment within which the proof assistant exists contains a rich set of tools. One of these might be a functional language debugger, capable of producing a trace of the execution of a functional program, for any given program — for example a property. This could be taken as an outline for a proof by evaluation, giving an indication of which rules would be appropriate at each step. The detailed proof could then be generated in exactly the manner described above. It seems unlikely that this approach could be extended to make the proof of properties automatic, but it certainly seems capable of directing the user as to the way in which a proof could be constructed.

With goal-directed proof, a goal is broken down into sub-goals. Any particular goal can only be broken down in a limited number of ways, and some of these are more obviously useful than others. A suitable matching algorithm could be used to compare the apparent relevance of various rules for any given step in the proof. Once selected, a step may be completed using the form-filling method described above.

In any proof, some steps are harder than others, and these are often deferred until others have been completed; also steps may depend on subproofs, which are not particularly relevant when encountered — it is very useful to be able to defer these until the main proof has been completed. To assist in this `holes` may be left in the proof. These consist of rules which transform the context in any desired way, but are currently unjustified. These holes in the proof must be attended to later, before the proof can be finally verified, but in the interim, they may be assumed true while attempting to prove the larger goal.

8.4 Objectives

The previous sections have laid out the general principles underlying the work that we want the assistant to carry out. This section details these tasks as the assistant sees them, and gives ideas as to the methods it will employ to satisfy them.

The first objective of the proof assistant is to allow a proof system to be established which is amenable to automatic checking. This system is described in the previous chapter, and is based around the principle of the evaluation of a property. The central feature of this aspect of the assistant is a knowledge base, or information store, containing information derived from the program, properties, and previously proved proofs.

The second objective is that the assistant should be able to discuss the proof 'knowledgeably' with the user. That is, the user should be able to ask questions about a proof, and have them answered, and also to manipulate a proof. In particular, this means that the assistant should be able to selectively interrogate the knowledge base and present the results to the user in any form the user desires, and to interactively present the user with more detailed information

about sections of the proof.

The third objective is for the assistant to be able to advise on the construction of proofs; that is, given a knowledge base and an immediate context within a proof, it should be able to assist the user on possible steps to take in order to progress with the proof.

Apart from these tactical objectives, the proof assistant must meet some non-functional criteria. It must facilitate making proof an integral part of development; if this does not happen, then the key objectives of increasing quality, and hence productivity will not be met. In order to do this, it must be well-integrated into the system, easy-to-use, and must provide real and obvious benefits to the user. Ideally, the assistant should also provide feedback within the development cycle by giving developers greater insights into their own — and each other's — code, by allowing them to access properties — known to be true — about the functions, as well as documentation, interfaces, and source. A browsing facility should be responsible for the presentation of all this information in a structured form to the user.

In many development teams and systems, project managers attempt to measure development progress: but it is hard to know exactly what to measure. Lines of code (usually in units of 1000 or 1000000) is probably the most common, often with qualifiers such as 'debugged' or 'fully documented'. With this system, another qualifier may be added - 'which have been shown to meet their specifications' - and if the project has been specified in advance, including a complete set of properties, the list of properties already proved could be the entire measure. Indeed, this would add incentive to prove properties, since this would be the measure of programmer productivity.

8.5 Architecture

The proof assistant is a tool within the environment described in Chapter 4. It has a number of connections with the other tools in that environment: the UI module provides it with an interface to the user; the compiler parses the user's code and generates proof axioms from it; the debugger (see Chapter 10) can be used to produce proof outlines.

To meet its objectives, the proof assistant must be able to access these other tools, create and maintain a database of facts about the program, acquire and store a set of rules to be used in proofs, and accept, store, display and modify proofs.

Figure 8.1 shows the architecture of the proof assistant which is similar to that of the complete environment. The control program accepts requests from the environment and controls the behaviour of the complete tool. The interface allows the control program to view the services of the rest of the environment as if they were in fact internal to the assistant. The knowledge base, used for the storage of information relating to the program, and of previously proved properties, is pictured here as being treated similarly. The proof rules are represented externally as a functional style program as described in Section 7.3, and internally as a parsed program.

Internally, the assistant maintains a collection of partial proofs. These may be referenced or accessed from within the environment by passing suitable messages to the assistant's control program. The proof engine may be used in two

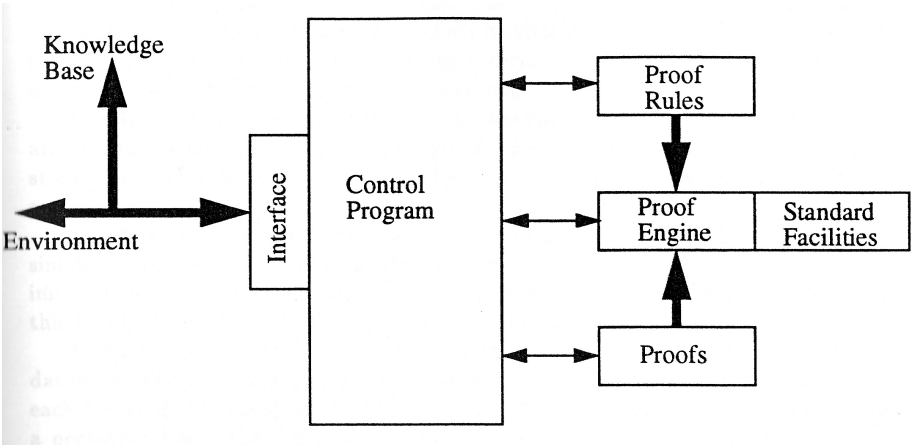

Figure 8.1: Organisation of Proof Assistant

rôles — either to validate a proof or to suggest possible rules that could be used to extend the proof. In doing so, it has access to all of the proof rules, the proof in question and the knowledge base. Some standard facilities are provided to the proof engine to handle certain theories, for example number theory and evaluation, without requiring all the rules to be programmed individually.

Externally, the assistant provides services as described in the previous section. It must accept requests to check, display or modify a proof. Its responses include indications that a particular task has been done, as well as interacting with other elements of the environment (e.g. the editor module) directly to produce the desired effects.

8.6 The Information Store

The functionality of the assistant may be broken down into three categories: the storage and manipulation of information about programs; deductions about the programs based upon this information; and interaction with the environment. This section looks at the information to be stored, and techniques that can be used to do this.

While parsing, the compiler picks up two sorts of information about programs: axioms are deduced from function definitions as described in Section 5.7, and (extended) type information is gathered directly from the source.

The proof rules are created by an "expert" when the system is set up. These are presented in the form of a functional program, which must be parsed in order to load it into the assistant.

The properties associated with a program, and their proofs, are a further two types of information which must be stored. The properties are simply expressions in the native functional language, while the proofs are sequences of steps in the language of proof rules. The properties and their proofs, while very closely related, should be kept separate for two reasons. The property may well be used in other proofs, for which the exact means in which it itself was proved will not be relevant (other than that it was sound); while any individual

property may have more than one proof associated with it (some of which may be unfinished or unprovable).

The final category of information to be handled are the proof contexts. These are needed transiently while work on a proof is progressing, but also it is desirable to store them, along with other transient information (e.g. window contexts) between sessions.

A wide variety of choices exist as to how this information could be stored. The simplest method is to load in the programs, proof rules, properties and proofs while initialising, and then to attempt each of the proofs. A "project" file could assist in this by maintaining a list of files to be loaded for any session.

A more sophisticated method for storing the information would be to use a database. The data is stored internally as structures using cross-references to access each other. If this could be stored directly on the disk using a database resembling a persistent heap, then this would be the most effective way of reinitialising. Alternatively, the data could be massaged to enable it to fit into a more traditional database model.

The database model of the information store is so useful, that the other tools in the environment would probably also wish to use it. As a consequence, this design views a persistent heap as a separate tool within the environment, which is capable of taking a data structure and storing it in a database as required.

8.7 Protocols

The three main functions that the proof assistant will be called upon to perform have been outlined in Section 8.4 above. In order to do this, each needs to be given a requests and responses interface, which together form the proof assistant's protocol. In addition, the knowledge base, being treated as a separate tool, must be given a similar protocol. The sections below outline the protocols for: the knowledge base; checking proofs; viewing proofs; and modifying proofs.

8.7.1 Queries

In order to access the knowledge base effectively from within the assistant (or elsewhere), the knowledge base must have a defined protocol, in terms of requests that it understands, and responses that it can make. In general, these requests will take the form of queries about particular names, either unspecific, or else qualified to take the scope into account. Figure 8.7.1 gives an outline of this query sub-protocol.

These queries allow for names (represented by a list of characters) to be used as a key to look up entries in each of the categories. Its interpretation in looking up axioms and types is obvious, since these are essentially definitions associated with names in scopes; in the case of a property or a proof it refers to any property or proof which contains any of the symbols. Rules are found like axioms, only in a completely different scope. Contexts are named specifically and retrieved only by their given names. The result of a query is always a list of pairs, each consisting of a (fully qualified) name and an item, where the item is an appropriate data structure. If no matches occured, the result is an empty list.

```
data kb_query_0.0_req =
  QueryRequest query

data kb_query_0.0_resp =
  QueryResult [(name,item)]

data query =
  Query    [category] [name] |
  QueryAnd [query] |
  QueryOr  [query]

data category =
  Axiom | EType | Propty | Proof | Contexts | Rule

type name = [char]

data item =
  ItemAx ... | ItemET ... | ItemPy ... |
  ...
```

Figure 8.2: Querying the Knowledge Base

```
data kb_update_0.0_req =
  UpdateRequest category name item
  UpdateDelete category name

data kb_update_0.0_resp =
  UpdateNew |
  UpdateReplaced |
  UpdateDeleted |
  UpdateFailed
```

Figure 8.3: Updating the Knowledge Base

It is also possible to update the knowledge base. In general,[1] this is not useful for axioms, types or rules, since the information for these is deduced directly from source programs at the initialisation stage; it is more useful for the proofs and contexts, which are generally modified and updated by the proof assistant. Properties are somewhere in between, in that they will generally be specified in designs (and hence appear in source files). The requests to update the database are presented in Figure 8.7.1. The possibilities are to add a new item in a particular category with a given fully-qualified name; or to delete a item from a particular category with a fully qualified name.

[1]The obvious exception to this is when the respective source files are loaded into the environment

```
data pa_check_0.0_req =
  AttemptProof proofid

data pa_check_0.0_resp =
  AttemptingProof sync-id |
  Successful_Proof sync-id |          -- asynchronous
  Proof_Error sync-id [char] [char]  -- asynchronous
```

Figure 8.4: Protocol for Checking Proofs

8.7.2 Checking Proofs

The procedure for checking a proof is fairly simple. The environment sends a message to the assistant, citing a proof which it wishes proved. The assistant then starts at the top of the proof, creating an initial context, and then applies each of the rules in the proof in turn, updating the context as it goes. The proof succeeds if and only if the expression in the final context is `True`.

The initial context consists of the initial property, an initially empty set of hypotheses, and an initially empty sequence of 'fixed' variables — those for which no substitution may be made. This environment is used as an extra argument to each proof rule, and a new context is returned.

At the end of the proof, the assistant sends an asynchronous message back to the environment saying either that the proof was successful, or else giving an error consisting of a message and the line of the proof that failed. Figure 8.7.2 gives the protocol for these messages; `sync-id` is a special synchronisation ID, to allow the environment and assistant to stay in sync, in spite of each working on separate actions.

8.7.3 Displaying Proofs

The normal form for proofs, as set out in Section 7.2 is not entirely clear, as is shown by the example proofs in Sections 7.2.4 and 6.2.5. The assistant can help by supporting a protocol to generate more readable forms for a given proof. This generally takes two forms: massaging the text of the proof to add in information which is to be found in the context, and then storing the information in an editor buffer, thus making it available to the user; and presenting some form of graphical overview or closeup of the proof.

There are two means to getting enhanced representations of the proof. The first, and simplest, is for the environment to support user interaction to control the display of the proof, in particular the use of dialogs to set the preferences for the display of individual steps of the proof. The alternative is to add pseudo-rules into the body of the proof, which likewise control the format of the proof.

Graphical representations of, for example, an overview of the proof, or an exploded view of one of the expressions or rules within the proof, may be requested interactively from within the environment. As above, a proof may be annotated with pseudo-rules to permanently attach particular graphical views either to a complete proof or to a step of a proof.

Acheiving the desired effect requires close cooperation between the assistant and the environment; the environment needs to be able to select parts of the

```
data pa_display_0.0_req =
  DisplayProof proof-id format
  DisplayProofStatement proof-id statement-id format
  AnnotateProof proof-id proof-annote
  AnnotateProofStatement proof-id statement-id state-annote

data pa_display_0.0_resp =
```

Figure 8.5: Protocol for Displaying Proofs

```
data format =
  Plain buffer |
  Expressions buffer bool |
  Outline
```

Figure 8.6: Formats for Displaying Proofs

proof and then encode the user's intended action into a message to send to the assistant. The basic protocol for this is presented in Figure 8.7.3. There are no specific responses to these messages from the assistant; instead it just generates suitable messages to send to the editor and user interface modules.

The `DisplayProof` and `DisplayProofStatement` messages instruct the assistant to construct a particular view of a proof, or an individual statement. The `proof-id` identifies which proof, and the format determines how the proof is to be displayed. The `statement-id` specifies a statement within the proof. The `AnnotateProof` and `AnnotateProofStatement` messages add annotations to an entire proof and to a proof statement respectively. Annotations are discussed in Section 8.8. Three formats are specified here for displaying proofs, which also apply to individual statements, and an extra format is provided just for the individual statements. These are summarised in Figure 8.7.3.

The `Plain` format just displays the proof rules as given in Section 7.2.4; the `Expressions` format gives the goal expression of the context at each step, applying annotations attached to the proof if the boolean is `True`; and the `Outline` format generates a graphical outline of the proof, giving each proof subblock as a box, and showing what the logical effect of each block is, as a deduction.

8.7.4 Manipulating Proofs

So far, proofs have been treated as static objects to be manipulated. In order to gain real benefit from having the assistant as part of the environment, it must be possible to extend, and indeed create proofs with assistance.

Four levels of assistance in this activity are envisaged: the user may request to type in a new proof step at a particular point in the proof, in effect simply editing the raw proof; the user may fill out an assistant-supplied dialog box, possibly copying information from elsewhere on the screen; the assistant may suggest a range of alternatives, and then assist the user in narrowing down what is needed; and finally, the assistant may do most of the work automatically, just

```
data pa_manip_0.0_req =
  CreateProof property |
  AddStep proof-id statement-id proof-step |
  DialogAddStep proof-id statement-id |
  DialogSuggest proof-id statement-id |
  SuggestAssist sync-id partial-proof-step |
  AutomateProof proof-id method

data pa_manip_0.0_resp =
  SuggestSync sync-id
```

Figure 8.7: Protocol for Manipulating Proofs

involving the user when it can progress no further.

The protocol for manipulating proofs is given in Figure 8.7.4. Proofs may be created from scratch by sending the `CreateProof` message, along with a property to be proved. The `AddStep` message adds a new step to a specified proof after a specified step. `DialogAddStep` specifies a proof and a step to insert a new step after, but depends upon a series of user interactions based around a dialog box to decide which proof rule to use and what arguments to give it. The `DialogSuggest` message works analogously to the `DialogAddStep` message, except that the assistant does more work, evaluating the possibilities, and trying to guide the user in the right direction. When the user has progressed some way, a button on the dialog box sends a `SuggestAssist` message which resynchronises the assistant with the user, so that the assistant knows more exactly where the user is going and can customise choices more precisely. Finally, the `AutomateProof` message attempts to generate a proof of a property given an existing proof (which it may ignore). No automatic proof methods are included in this design; however, a possibility for extracting an outline proof from a property using the debugger is discussed in Chapter 10.

Other ways in which the proof assistant can assist the user in the construction and maintenance of proofs are legion. For example, inserting an extra proof rule half way through a proof could subtly change the format of the expression, and misdirect all subsequent references to a particular term. The assistant adjusting all subsequent lines of the proof to take account of this would be invaluable.

8.8 Annotating the Language of Proof Rules

The language of proofs presented in Section 7.2 describes adequately the meaning of a proof, and is sufficient to check a proof of a property. However, it is insufficient to say how it should be laid out to be comprehensible to humans, or to be aesthetically pleasing. In order to do this, it is necessary to extend the language of proof rules (see Section 7.3) by adding some annotations in the form of pseudo-rules.

There are two basic sorts of these annotations: they may either apply to the entire proof (or some part of it); or they may just apply to the immediately following statement. Some annotations may be used in either way. Three special proof rules are used to handle annotations.

An annotation which just applies to the next line is introduced by the keyword `annotate`, followed by the annotation type, and the appropriate arguments. An annotation which applies to a section of a proof is introduced by `begin_annotate` and ended by `end_annotate`. The first argument to `begin`, and the only argument to `end`, is an identifier which is used to handle the pairing of `begin`s and `end`s since these do not need to nest, but can be interleaved. Additionally, it is not necessary to have an `end` for every `begin`: often annotations will be used for the entire proof, and so will be started at the start of the proof, and never ended.

Example annotations include: replacing every occurence of an expression (or a pattern) with another expression; replacing a complex subexpression with an ellipsis; highlighting a term which moves dramatically around an expression from step to step; adding extra information from the context; and adding graphical annotations to particular steps.

Chapter 9

The Meaning of Proofs

Chapter 7 presented a detailed exposition of a proof system for functional languages, designed to be intuitive for use by functional programmers. Chapter 8 illustrated how the proof system might be incorporated into a functional program development environment, by presenting the design for a proof assistant. The only remaining task is to demonstrate that the proofs constructed using this proof system are sound. This is done here by interpreting the proof system in terms of classical or constructive logic together with domain theory.

A proof in the proof system consists of an initial context and a series of steps. Each step uses a current context, and produces a new context as its result. The initial context is used by the first step, and subsequent steps use the context resulting from the previous step. The final step produces a final result context. A proof succeeds if this final context has `True` as its current expression. A proof is sound if it succeeds and every step in the proof is sound.

It will be recalled from Chapter 7 that any given step of the proof will do one of two things: either it will transform the current expression in some way, generally making it simpler, or it will add information to the fact database. It is possible for a single rule to do both, but such rules are not considered here. As the proof progresses, the fact database will increase in size, each added datum being a logical consequence of those already known; and the current expression will become simpler, each being a logical consequence of the fact database taken together with the resultant expression after the next step. In this way, the proof is a mixture of forward and backward reasoning: the known facts increase in a forwards fashion, starting with the facts which are direct consequences of the program, while the current expression evolves backwards, ending with a statement of absolute truth. The four general cases of proof steps are illustrated in Figures 7.1 to 7.4 in Chapter 7.

Figure 9.1 further illustrates this mixture of extending the fact database while reducing the current expression, by giving a two-step proof and showing the cumulative implications of the steps.

To demonstrate that a proof is sound, it must be shown that any fact added to the database is a logical consequence of the existing facts, and that any reductions performed on the current expression guarantee that the prior expression is a logical consequence of the facts database and the resultant expression. It may be shown that each of the proof rules is valid for all possible arguments, and that the structures used to bind the proof are truth-preserving. Then any

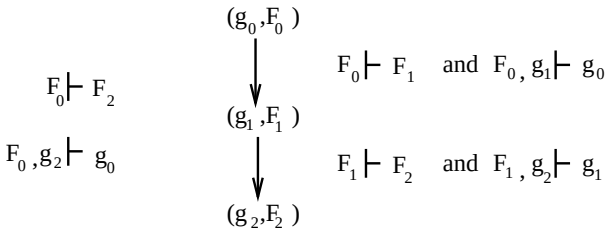

Figure 9.1: Cumulative proof steps

proof constructed using these rules must be sound.

As explained in Chapter 7 no explicit statement of which rules are supported, or how they are to be implemented, or what style they should take, is made here; rather, all that is described is a mechanism for checking that any given proof rule is sound with respect to the appropriate logic.

9.1 Translation of Expressions

Within a proof, statements such as goals to be proved or facts in the database are written in the object-level language. Within the proof system, these are not viewed as parts of the object program, but rather as statements about the object program. For example, an equation such as `sizeof Nil = 0` is taken to mean that the expressions on either side have the same value. A boolean expression such as `member l 42` standing alone is taken to mean that the expression has value `True`, i.e. that if it were evaluated for any suitable instantiations of its free variables, the result would be `True`. In this chapter, this interpretation of object language statements is made precise by providing a direct translation into first order logic.

The value of an expression can be given precisely via the value semantics discussed in Chapter 2. The value of an object level expression e will be represented as $\|e\|$, and all values will be expressed in the form $\|e\|$ for suitable expressions e, except that $\bot$ will be used as a synonym for $\|\texttt{Bottom}\|$. The value of a type expression is the set of values associated with that type, as in Chapter 2. In addition, boolean expressions and connectives in the object language have an interpretation as logical statements and connectives. In order to make this interpretation clear, two sets of symbols are used in this chapter. The symbols of the object language are shown on the left and the roughly equivalent symbols of first order logic are shown on the right in Figure 9.2.

It is important to note that boolean expressions in the object language can have three values, $\|\texttt{True}\|$, $\|\texttt{False}\|$ and $\bot$. An expression e is only interpreted as *true* if it always evaluates to $\|\texttt{True}\|$, and is interpreted as *false* if it ever evaluates to $\|\texttt{False}\|$ or to $\bot$.

The computable equality `==` only evaluates to *true* if both arguments represent the same finite, total data value. It is therefore necessary to test that both values are finite, total data values as well as testing their equality. The non-computable function `isftd` is used to test whether or not a value is finite, total and data. The interpretations of `&` and `~` correspond to the way in which

`True`	*true*
`False`	*false*
`~`	$\neg$
`&`	$\wedge$
`|`	$\vee$
`imp`	$\rightarrow$
`=  ==`	$\equiv$
`::`	$\in$

Figure 9.2: Rough comparison of computable and logical connectives

`‖ ~x ‖`	`->`	`‖x‖` $\equiv$ `‖False‖`
`‖ x & y ‖`	`->`	`‖x‖` $\equiv$ `‖True‖` $\wedge$ `‖y‖` $\equiv$ `‖True‖`
`‖ x | y ‖`	`->`	`‖x‖` $\equiv$ `‖True‖` $\vee$ (`‖x‖` $\equiv$ `‖False‖` $\wedge$ `‖y‖` $\equiv$ `‖True‖`)
`‖ imp x y ‖`	`->`	`‖x‖` $\equiv$ `‖False‖` $\vee$ (`‖x‖` $\equiv$ `‖True‖` $\wedge$ `‖y‖` $\equiv$ `‖True‖`)
`‖ x == y ‖`	`->`	*isftd* `‖x‖` $\wedge$ *isftd* `‖y‖` $\wedge$ `‖x‖` $\equiv$ `‖y‖`
`lhs = rhs`	`->`	`lhs` $\equiv$ `rhs`
`‖ x :: t ‖`	`->`	`‖x‖` $\in$ `‖t‖`

Figure 9.3: Translation of Boolean Operators

they are evaluated in the object language, which takes into account that $\perp$ must be interpreted as *false*. The interpretation of `|` includes the anticipated evaluation order of the functional operator: the second argument may be `Bottom`, but only provided that the first argument is `True`; the first argument may never be `Bottom`; and the second argument may not be `Bottom` if the first argument is `False`. The interpretation for `imp` is analogous, except that where `|` allows the second argument to be `Bottom` when the first is `True`, `imp` allows the second argument to be `Bottom` when the first is `False`.

9.2 Justification of Proof Rules

The previous section indicated how a statement, written in the object-level language, could be interpreted as a statement about the object program, by translation into classical or constructive logic. It is now necessary to show a means by which one such statement may be derived from one or more others in a sound fashion. It will be recalled from Chapter 7 that there are essentially four styles of proof rule: goal-directed steps, forward steps, transformational steps, and steps with subproofs, as illustrated in Figures refgoal-step to 7.4. The transformational steps do not differ greatly from the goal-directed steps, so only the other three will be considered here.

Each proof rule, of whatever style, provides a mapping from one context to another, possibly with the assistance of one or more subproofs. A proof rule may be unpacked so as to give a definition of its operation in terms of its prior and resultant contexts, together with the subproofs it uses. Taken together, this gives a classical statement of the meaning of the rule, which may then be checked for validity by a separate classical proof.

For example, the proof rule for commuting addition in Chapter 7 was given as:

```
add_comm (Spec s) [] (Context g f sb rb) =
  Context (replace_subexp g s (Expr (Lit "+") [r,l]))  f sb rb

  (Expr (Lit "+") [l,r]) = extract_subexp g s
```

This is a goal-directed step, so it has the validity condition given in Figure 7.1, that

```
F, g' ⊢ g
```

where g is represented by the free variable g in the context, while g' is given by the replacement of one subexpression for another in the original goal g. The database of known facts F, represented by the free variable f from the context, is not used in this proof rule. The rule acts only on a subexpression of the goal g, which subexpression is denoted by s. If s is not a valid subexpression of g, or if it is not of the appropriate form (the addition of two sub-expressions), then the proof rule fails. Assuming it succeeds, then the goal g must have at the indicated point a subexpression l+r, so there is an expression $X[v]$ where the free variable v occurs exactly once in X and not in g or g', which has the properties:

```
g = X[v/(l+r)]
g' = X[v/(r+l)]
```

The first of these states that substituting l+r for the variable v in X produces the original goal, so X denotes all of the goal which is not specified in the rule, while the variable v represents the part of the goal specified by v. For example, in the expression `18-(2+3) == 13`, the expression X would be $X[v]$ =`18-v == 13`. The subexpression replacement in the rule is equivalent to substituting for the variable v in X, and is the second property of the expression X given above, namely that the resultant goal is the result of substituting r+l for v in X. So the complete condition for this step to be valid, given that a suitable X may be found, is:

```
X[v/(r+l)] ⊢ X[v/(l+r)]
```

which may be easily proved by appealing to the law of substitution of equals for equals, then demonstrating that `r+l` and `l+r` are equal, which is given by the property of commutativity of addition from number theory.

The rule `block_to_unfold_hyp` consists of a forward step combined with a single subproof. Its definition will be recalled from Chapter 7:

```
block_to_prove_hyp ex hs [p] (Context g f) =
  assert (ff == Expr (Lit "True") [])
         (Context g (add_fact newf f))

  (Context ff fr) = p (Context ex (add_fact_list hs f))
  newf = Expr (Lit "_deduct") (hs++[ex])
```

The validity condition for a forward step with a single subproof is:

```
⊢p t1
F,t1 ⊢ f
F' = F*f
```

The condition for the inner proof (which is inductively assumed to be valid), may be stated as:

```
F,hs ⊢p ex
```

By the deduction theorem, this may be rewritten to:

```
F ⊢ hs ⇒ ex
```

And so the condition for the complete rule may be written as:

```
F ⊢ hs ⇒ ex          (1)
F' = F*(hs ⇒ ex)     (2)
```

Condition (2) is simply satisfied by the addition of the new fact to the facts database. Condition (1) is satisfied since if the proof p is sound then `ex` follows under the hypotheses `hs`, and by the deduction theorem this is equivalent to expression (1), and the deduction theorem is used on `hs` and the expression `ex` to generate the new fact.

To enable rules to check that subproofs meet certain soundness conditions, one or more boolean flags may be added to the actual definition of the context. The holes in the proof suggested at the end of Chapter 8 may be implemented by adding a boolean flag which is set initially `True`, but may be later set to `False` if a rule which leaves a hole is used. A rule which does not leave a hole in the proof passes this flag on unchanged. Transformation arguments, where one expression is transformed into another in the style of Section 6.2.3, requires a check that all the proof steps are reversible. A boolean flag working on exactly the same principle as for the holes accomplishes this.

Chapter 10

Conclusions and Future Work

The objective of this research has been to provide the design for an environment in which functional programs might be developed, properties of them defined, and then these properties might be proved correct. Two constraints were placed on the achievement of this objective: that the features of the environment and the proof system should be intuitive to functional programmers; and that the design of the environment should lead the programmer naturally from the design of a program, through the development process and encourage the proof of properties as a guarantee of software correctness.

The design of the environment began with the definition of a functional language, a sublanguage of the standard functional language Haskell, and for which an evaluation mechanism was provided. This sublanguage was given the name Flair, and was described in Chapters 2 and 3. It fulfills the requirements of a functional language, as intended in the objective, while avoiding many of the complications in Haskell (such as arrays, classes, and the interaction of pattern matching and guarded equations).

Chapter 4 presented the framework for the environment, permitting the construction of a collection of tools, coordinated by a single module based around the execution of a user-configurable functional program, written in the same functional sublanguage, Flair. Three other elementary tools — a compiler, an executor, and an interface to the non-functional filestore were described. All of the modules were made sufficiently general that users could configure the environment to their own preferences, and, at the same time, new tools could be introduced into the system with a minimum of complications. This provided a design for the first part of the objective — an environment within which functional programs could be developed — and did so under the first constraint, that the environment should be intuitive to functional programmers.

To meet the rest of the objective, it was necessary to develop a proof system, and specifically one that catered for the intuitions of functional programmers. Chapter 5 started this process by refining the type system, to produce a means of making explicit statements about types as sets of values, and to introduce the ideas of subtypes and overlapping types. These ideas, together with the pattern matching algorithm of Chapter 3, allowed function definitions, compris-

ing overlapping defining equations, to be separated into a set of independent equations, each individually governed by an expression which tests whether or not the equation is applicable to any given set of values. This separation into independent equations allows the building of a set of axioms which are immediately provable from the source program, and form an essential component of any proof system.

The next step was to find a suitable logic, and a suitable means of presenting the proofs. Again, the overriding concern was that each should reflect the intuitions of functional programmers. Chapter 6 reviewed some of the options, and eventually selected a form for statements that closely resembled that for expressions in the object language, with the proviso that these could be understood to be statements about the object language, by translations into a more familiar representation, for example, classical or constructive logic. The presentation of the proof decided upon was also functional: it consisted of a compound function from an initial context to a final context, but written sequentially, rather than as nested applications.

Chapter 7 presented the selected proof system in detail, indicating the nature of the context, the various types of proof step which are supported, and the precise representation of proofs. The concluding section of the chapter described how proof rules themselves could be written in the same functional sublanguage as the object program, and that was to be used elsewhere in the environment.

The achievement of the remainder of the objective — integrating proof into the development of functional programs — and the second constraint — providing a natural progression from design to proof — was the subject of Chapter 8, which described the facilities provided by a specialised tool — the proof assistant — for the verification, examination, and manipulation of proofs. The use of proofs was encouraged by enabling them to be mechanically checked, thus giving real guarantees of correctness. The facilities for the manipulation of proofs, particularly the proof outline obtained from a debugger such as that described below, assist in the construction of proofs in a way which flows naturally from the normal process of testing programs using a debugger — thus satisfying the second constraint. The ability to examine a proof presented in a clear, intelligent fashion provides valuable documentation, both of the proof, and, because a property proof is so closely related to the function definitions, of the functions involved in the property, and their relations with each other.

Chapter 9 finished the discussion on the proof system, justifying its correctness by appealing to classical or constructive logic, and enabling each proof rule to be checked.

Two issues emerge immediately as still requiring attention. Firstly, this is a design, not an implementation. An implementation would provide both a useful tool for functional program development, and a means of testing many of the ideas presented here, and their applications with real software development. Secondly, the features of Haskell not in the sublanguage need to be addressed (particularly the class system), and any extensions made to the types system of Chapter 5, and the proof system described in Chapters 7, 8 and 9 None of the other material is likely to be affected.

Three additional tools have been envisaged as current or future developments within the context of the environment described here. At the time of writing, one of these is currently under research at Bristol. These three tools flesh out the environment, which, even with the proof assistant, is not really capable of

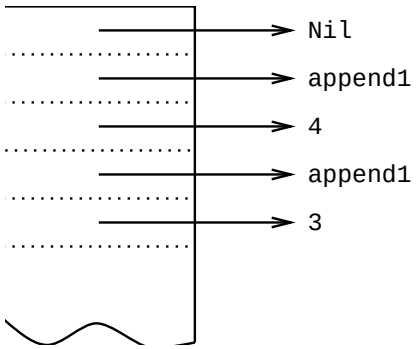

Figure 10.1: Stack while executing `reverse (Cons 3 (Cons 4 Nil))`

covering all the aspects of the development process.

A functional language debugger, providing a range of facilities for debugging programs, would be an invaluable aid, firstly to testing and understanding programs, and secondly as a stepping stone from program development to program proof. In the G-Code model, it is possible at the start of any block to automatically determine the current expression to be evaluated (although special information needs to be attached to each block, apart from the code, in order to do this). The expression may be determined by looking at the current block, the current stack, and the heap nodes pointed to, either by pointers on the stack, or within other heap nodes. This expression, once obtained, is the most telling piece of information relating to the current state of execution of a program. For example, the `reverse` function, for which G-Code is given in Chapter 3 (see Figures 3.5 to 3.8), might be used to evaluate the expression:

```
reverse (Cons 3 (Cons 4 Nil))
```

When the recursion has been fully unwound, `reverse` is entered with one argument on the stack (`Nil`) and two previous stack frames maintained by the code for `append`. The stack is as given in Figure 10.1. `append1` is the `CONTINUE` block of the function `append`, to which control returns after evaluating its first argument, which decides between the cases `Nil` and `Cons`. The expression represented by being in the block `reverse` is the function `reverse` applied to the top element of the stack, that is,

```
reverse Nil
```

Each lower frame on the stack may be recovered in turn. In this case, each is in the block `append1`, which represents the expression `append` applied first to the result of an inner evaluation (which is held in the special register `NODE` during evaluation, but here is the expression previously recovered), and secondly to the new top element of the stack (after the previous frame and return address have been removed). So the expression for the second stack frame is

```
append (reverse Nil) 4
```

and that for the whole expression is

```
append (append (reverse Nil) 4) 3
```

which can indeed be determined to be the value of the expression at that point, and equivalent to the original expression, and the final value

```
Cons 4 (Cons 3 Nil)
```

An outline proof of a property is obtained by tracing the evaluation of a property, and observing which sub-expressions are unfolded at each step. This is the functional language equivalent of a normal execution trace of a sequential program. The result, if properly formatted and collated to remove unnecessary laziness, can provide a guide for a proof, encouraging programmers to move from program development to proof. Two important points which must still be dealt with by the user are the substitution of induction for recursion (although in general the outline proof will include both base and inductive cases); and the handling of non-terminating expressions (which are beyond the abilities of a debugger).

As with environments, most work in the field of debugging has been done with imperative languages, such as C and PASCAL. PROVIDE [57] is a debugging environment for a subset of the C language. It provides a graphical environment which gives the user easy access to the program source code. The primary objective of PROVIDE is to give the user a pictorial view, and graphical control of the state of program execution at any point. PROVIDE is based significantly on BALSE [10], a teaching aid used for the animation of processes and observation of the data structures being manipulated. BALSE is programmed at compile-time from within the program being visualised. PROVIDE is programmed interactively while the program is being executed, and the graphical display can be bound at any time. The data is updated after every transaction. Generally, a transaction consists of one line of source code, but may be defined by the user to be any number of lines. This approach has two advantages: firstly, it allows the user to specify the granularity of the investigation, and secondly, it allows the user to bind interrelated statements into a single atomic operation. Additionally, the system allows the states of the program to be stored and then later played back for repeated inspection of target areas of the code.

PICT [25] differs from systems such as PROVIDE and Tinkertoy [19] which concentrate on the relationships between the procedures and their arguments. Instead, PICT is a pictorial programming language which focuses on making the data flows clear, rather than the relationships between the functions and their arguments. PICT is based around a very simple PASCAL-like imperative language which is entirely graphical. The system works in colour: data flows between functions are colour-coded, the colours indicating the variables being used. The functions and procedures themselves are represented by colour icons, the patterns on the icons indicating both the name and the type signature of the procedure. When the procedure has been constructed it is possible to observe its execution in exactly the same graphical form. The debugging cycle is then rapidly and easily integrated with the main edit and compile operations.

Such traditional methods of debugging (tracing data and code flows), which work well for imperative, state-based languages, are not really practical for the debugging and analysis of non-strict functional programming languages, since these are not state-based, and do not have a definite, step-by-step evaluation order.

Instead, existing functional environments place great emphasis on reuseable code modules and the testing of individual features and functions away from the main body of the program. The biggest problem with this is that in order to test behaviour of real programs, it is necessary to retype structures created by a producer in order to feed them to a consumer.

One solution is to allow side-effects in the evaluation – within an environment – but to treat these as separate from the value of the program itself. This can be done in a number of ways. One possibility is just to introduce printing statements as a function applied to a value whose value is simply the original value. An alternative is to return an annotated pair from a function, whose annotated portion may be printed, or manipulated, from within the context of the calling function. Another problem to be encountered here is that, with lazy evaluation, the generation and printing of these values are interleaved, and so the printing of this output can be very confusing.

A more comprehensive solution is presented by Hall and O'Donnell [31], who present a language DAISY, and a transformation environment whereby standard schemes or 'templates' may be applied to functions to produce debugging functions. These allow for user interaction, variable display and execution, as well as redefining of functions, when certain conditions within the executing code are encountered. Since the transformations are automatic (although applied manually), the source code for the program is always completely declarative.

A tool which is found useful by many programmers is MAKE [21], which keeps programs under development up to date in such a way that the minimum amount of work needs to be done. This idea is akin to that of lazy evaluation, and it is possible to design a functional tool to perform this task. A collection of definitions indicate the dependencies between various definitions and modules: if a module has not been recompiled since its last change, and is needed, then it is recompiled. This can also be used as an optimisation: if a result has been evaluated and stored, then the stored version may be used if nothing it depends upon has changed. It also ties in with the ideas of automatic recomputation in spreadsheets. Finally, it can provide a fully functional interface to the operating system.

This research originated as the first step towards the semi-automatic construction of functional programs. Turner [81] addresses the issue of extracting programs from constructive proofs; but this is somewhat unsatisfactory for two reasons: the proofs are somewhat hard to construct, possibly harder in general than the programs; and the proof must still be generated by hand.

The next step towards the goal of more automatic construction of programs from the work presented here would appear to be allowing machine assistance in the generation of programs, similar to that permitted for proofs. A specification for the function is laid down, then the proof begins by attempting to demonstrate that a function exists to match the given specification. Turner used various sorting functions as an example of his approach to this problem. The following example attempts to illustrate an alternative derivation of the insert sort by using an interactive method. Like Turner, it will be assumed that the functions perm and order are well known, as are their properties (see Figure 10.2). This example is given using the standard Haskell notation for lists and guarded equations.

The first step is to define the property to be proved. This is written as an expression, and a property that is to hold. The intent is to demonstrate a

```
ord [] = True
ord [x] = True
ord (x:y:rxs) = x <= y & ord (y:rxs)

perm [] [] = True
perm (x:rxs) ys =
  stest & perm rxs rys
  where
    (stest,rys) = split x ys

split x [] = (False,[])
split x (y:rys)
= (True,rys), x == y
= (stest,y:rys1), otherwise
  where
    (stest,rys1) = split x rys
```

Figure 10.2: The functions perm and order

function interactively which matches the specification.

```
* sort xs :: ord (sort xs) & perm xs (sort xs)
```

The asterisk * indicates a user prompt, at which the specification is typed. The next step is to decide how the function will treat its arguments. In this case a recursion will be chosen, which handles the two cases of empty lists and non-empty lists.

```
* induction(sort,xs::{[],(x:rxs)})

  sort [] = _1
  sort (x:rxs) = _2 x (sort rxs)

  _1 :: ord (_1) & perm [] (_1)
  _2 x ys :: ord ys => ord (_2 x ys) & perm (x:ys) (_2 x ys)
```

This produces a definition for sort in terms of two subsidiary expressions, each of which is automatically given a specification similar to that originally laid down for the expression `sort xs`. The new functions are lifted out to the top level and so must be given any free variables appearing in the patterns of the previously defined function. For simplicity, only those arguments which are needed are included here, and the specification here (and elsewhere) has been simplified. To avoid cyclic arguments, once a function has been defined (such as `sort` here) it may not be used later in the definition.

The next step is simply to suggest a possible expression for `_1`. The answer is suggested by its specification, since the only permutation of the empty list is the empty list. This completes the definition of the first case of the function sort, and gives rise to a proof obligation, which is simply the specification of `_1` instantiated with the suggested expression.

```
* suggest(_1,[])

  sort [] = []

  ProofOblig : ord [] & perm [] []
```

The next step is to give a more meaningful name to _2. The purpose of _2 is to insert the element x into the appropriate place in the sorted list of remaining elements.

```
* name(_2,insert)

  sort (x:rxs) = insert x (sort rxs)

  insert x ys :: ord ys => ord (_2 x ys) & perm (x:ys) (_2 x ys)
```

The next step is to begin the definition of insert. Like sort, this is a recursive function, so the first step is to unfold the recursion into the two cases, empty and non-empty lists.

```
* induction(insert,ys::{[],(y:rys)})

  insert x [] = _3
  insert x (y:rys) = _4 x y rys (insert x rys)

  _3 :: ord [] => ord _3 & perm (x:[]) (_3)
  _4 x y rys zs ::
     ord zs & perm (x:rys) zs & ord (y:rys) =>
       ord (_4 x y zs) & perm (x:(y:rys)) (_4 x y zs)
```

The case of insert for _3 may be easily cleared up, as for the base case of sort above: the only permutation of a singleton list is the singleton list.

```
* suggest(_3,[x])

  insert x [] = [x]

  ProofOblig : ord [] => ord ([x]) & perm (x:[]) ([x])
```

This gives rise to the appropriate proof obligation, that if the argument to insert is sorted, then so will the result.

The non-empty case of insert, _4, in fact consists of two cases: when the element x is to be placed on the front of the remaining list, and when it is to be inserted later in the list. This requires the use of a test, here represented by a guarded equation.

```
* guard_eqn(_4,{x<=y,otherwise})

  insert x (y:rys)
  = _5 x y rys (insert x rys), x<=y
  = _6 x y rys (insert x rys), otherwise
```

```
_5 x y rys zs ::
   ord zs & perm (x:rys) zs & ord (y:rys) & x <= y =>
     ord (_5 x y rys zs) & perm (x:(y:rys)) (_5 x y rys zs))
_6 x y rys zs ::
   ord zs & perm (x:rys) zs & ord (y:rys) & x > y =>
     ord (_6 x y rys zs) & perm (x:(y:rys)) (_6 x y rys zs))
```

Since it is known what are to be the two actions in these two cases they may now be stated as suggestions, and then the code is finished, just leaving the proof obligations, which may be tackled with the aid of the proof assistant.

```
* suggest(_5,(x:y:rys))

  insert x (y:rys)
  = (x:y:rys), x<=y
  ...

  ProofOblig :
    ord (insert x rys) & perm (x:rys) & ord (y:rys) & x <= y =>
      ord (x:y:rys) & perm (x:(y:rys)) (x:y:rys))

* suggest(_6,(y:insert x rys))

  insert x (y:rys)
  ...
  = (y:insert x rys), otherwise

  ProofOblig :
    ord (insert x rys) & perm (x:rys) & ord (y:rys) & x > y =>
      ord (y:insert x rys) & perm (x:(y:rys)) (y:insert x rys))

* done

  Proof Obligations:

  ord [] & perm [] []
  ord [] => ord ([x]) & perm (x:[]) ([x])
  ord (insert x rys) & perm (x:rys) & ord (y:rys) & x <= y =>
    ord (x:y:rys) & perm (x:(y:rys)) (x:y:rys))
  ord (insert x rys) & perm (x:rys) & ord (y:rys) & x > y =>
    ord (y:insert x rys) & perm (x:(y:rys)) (y:insert x rys))
```

Bibliography

[1] Allman. An introduction to the source code control system. In *UNIX Programmer's Manual*. University of California (Berkeley), 1986.

[2] Altmann, Hawke, and Martin. An integrated programming environment based on multiple concurrent views. *Australian Computer Journal*, 20(2):65–72, May 1988.

[3] Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[4] Bahlke, Moritz, and Snelting. A generator for language-specific debugging systems. *SIGPLAN Notices*, 22(7):92–101, July 1987.

[5] Beaudouin-Lafon and Gresse. CATY: A programming environment for graphic and interactive program construction. *Technology and Science Information*, 3(4):257–262, May 1985.

[6] Bird and Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[7] Blair, Malik, Nicol, and Walpole. A synthesis of object-oriented and functional ideas in the design of a distributed software engineering environment. *Software Engineering Journal*, 5(3):193–204, May 1990.

[8] Bourne. An introduction to the UNIX shell. In *UNIX Programmer's Manual*, volume 2A. Bell Laboratories, 1978.

[9] Boyer and Moore. *A Computational Logic*. Academic Press, 1979.

[10] Brown and Sedgewick. A system for algorithm animation. In *Proc. SIGGRAPH*, pages 177–186, July 1984.

[11] Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[12] Burstall and Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[13] Clement. GIPE: Generation of interactive programming environments. *Technical Science Information*, 9(2):157–165, 1990.

[14] Darlington. Program transformation. In Darlington, Henderson, and Turner, editors, *Functional Programming and Its Applications*, pages 193–217. Cambridge University Press, 1982.

[15] Darlington and Horns. A functional programming environment supporting execution, debugging and transformation. In *Lecture Notes in Computer Science*, volume 365, page 286. Springer-Verlag, 1989.

[16] Delisle, Menicasy, and Schwartz. Viewing a programming environment as a single tool. *SIGPLAN Notices*, 19(5):49–56, 1984.

[17] Donzeau-Gouge, Dubois, Facon, and Jean. Development of a programming environment for SETL. In *ESEC 87: 1st European Software Engineering Conference*, September 1987.

[18] Dwelly. Synchronizing the I/O behaviour of functional programs with feedback. *Information Processing Letters*, 28(1):45–51, May 1988.

[19] Edel. The Tinkertoy graphical programming environment. *IEEE Transactions on Software Engineering*, 14(8):1110–1115, August 1988.

[20] Eisenbach. *Functional Programming: Language Tools and Architectures.* Ellis Horword, 1987.

[21] Feldman. Make – a package for maintaining computer programs. In *UNIX Programmer's Manual.* University of California (Berkeley), 1986.

[22] Field and Harrison. *Functional Programming.* Addison Wesley, 1988.

[23] Galen. GNOME: An introductory programming environment based on a family of structure editors. *SIGPLAN Notices*, 19(5):65, 1984.

[24] Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper and Row, 1986.

[25] Glinert and Tanimoto. PICT – an interactive graphical programming environment. *Computer*, 17(11):7–25, November 1984.

[26] Goldberg and Robson. *SMALLTALK-80.* Addison-Wesley, 1983-84.

[27] Goswami and Patnaik. Foundations of software technology and theoretical computer science. In *Fourth Conference Proceedings*, pages 44–59, Bangalore, India, 1984.

[28] Goswami and Patnaik. A functional style of programming with CSP-like communication mechanisms. *New Generation Computing*, 7(4):341–364, 1990.

[29] Habermann and Notkin. The Gandalf software development environment. Technical report, Carnegie-Mellon University, January 1982.

[30] Hailpern, Huynh, and Reversz. Comparing two functional programming systems. *IEEE Transactions on Software Engineering*, 15(5):532–542, May 1989.

[31] Hall and O'Donnell. Debugging in a side effect free programming environment. *Sigplan Notices*, 20(7):60–68, July 1985.

[32] Hayes. *Specification Case Studies*. Prentice Hall, 1987.

[33] Heering and Klint. Towards monolingual programming environments. *ACM Transactions on Programming Languages and Systems*, 7(2):183–213, April 1985.

[34] Henderson. Purely functional operating systems. In Darlington, Henderson, and Turner, editors, *Functional Programming and Its Applications*. Cambridge University Press, 1982.

[35] Henderson and Notkin. Integrated design and programming environments. *Computer*, 20(11):12–16, November 1987.

[36] Holyer. Types and sets in functional languages. Technical Report CS-88-11, University of Bristol, Dept. of Computer Science, 1988.

[37] Hudak and Bloss. The aggregate update problem in functional programming systems. In *12th ACM Symposium on the Principles of Programming Languages*, pages 300–314, 1985.

[38] Hudak, Wadler, et al. The Haskell report. Technical report, University of Glasgow, Dept. of Computer Science, 1990.

[39] Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.

[40] Jones and Sinclair. Functional programming and operating systems. *Computer Journal*, 32(2):162, 1989.

[41] Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1), 1979.

[42] Kahn. Implementation of arrays. *Prolog Digest*, 2(4), 1984.

[43] Kernighan and Plauger. *The Elements of Programming Style*. McGraw-Hill, 1978.

[44] Kernighan and Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[45] Knuth. Literate programming. *Computer Journal*, 27(2):97–111, 1984.

[46] Knuth. *The TeX Book*. Addison Wesley, 1986.

[47] Kolb, Sommer, and Stadel. Programming environments. *Computational Physics Communications*, 41(2–3):227–244, August 1986.

[48] Koopman. Interactive programs in a functional language: a functional implementation of an editor. *Software — Practice and Experience*, 17(9):609–672, September 1987.

[49] Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–164, 1966.

[50] Leilt. Top-down design in a functional environment. *Software — Practice and Experience*, 14(10):921, 1984.

[51] Lindsay. A survey of mechanical support for formal reasoning. *IEEE Software Engineering Journal*, 3(1):3–25, 1988.

[52] Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[53] Lopriore. A user interface specification for a programming debugging and measuring environment. *Software — Practice and Experience*, 19(5):437–460, May 1989.

[54] Manna and Waldinger. Synthesis: Dreams ⇒ programs. *IEEE Software Engineering*, 5(4):294–328, 1978.

[55] Miller, Ambreas, Cagan, and Kendzierski. The unified programming environment — unobtrusive support. In *Advanced Programming Environments*, pages 507–518. Trondheim, Norway, June 1986.

[56] Milner. Type polymorphism. *Journal of Computer and System Sciences*, 1978.

[57] Moher. PROVIDE: A process visualisation and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[58] Nobel. Remarks on SASL and verification of functional programming languages. In *Lecture Notes in Computer Science*, volume 270, page 265. Springer-Verlag, 1987.

[59] O'Donnell. Dialogues: a basis for constructing programming environments. *SIGPLAN Notices*, 20(7):19–27, July 1985.

[60] Paulson. *Logic and Computation: interactive proof with Cambridge LCF.* University Press, Cambridge, 1987.

[61] Peyton Jones. Functional programming languages as software engineering tools. In D. Ince, editor, *Software Engineering: the decade of change*, pages 124–153. 1986.

[62] Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[63] Peyton Jones and Salkild. The spineless, tagless g-machine. In *Functional Programming and Computer Architecture.* Addison Wesley, 1989.

[64] Poncet. SADL: a software development environment for software specification, design and programming. In *ESEC 87: 1st European Software Engineering Conference*, September 1987.

[65] Reeskaug and Skaar. An environment for literate SmallTalk programming. *SIGPLAN Notices*, 24(10):337–345, 1989.

[66] Reppy and Gansner. A foundation for programming environments. *SIGPLAN Notices*, 22(1):218–227, January 1987.

[67] Robin. Compiler aspects of an environment for programming by demonstration. In *Lecture Notes in Computer Science*, volume 282, page 199. Springer-Verlag, 1987.

[68] Sanders. An evaluation of functional programming for the commercial envrionment. *British Telecom Technology Journal*, 7(3):25–33, July 1989.

[69] Sanders, Johnson, and Tinker. From Z specification to functional implementations. *British Telecom Technology Journal*, 7(4):47–63, October 1989.

[70] Schmidt. *Denotational Semantics: a Methodology for Program Development*. Allyn and Bacon, 1988.

[71] Skedzielewski, Yates, and Oldehoeft. DI: an interactive debugging interpreter for applicative languages. *SIGPLAN Notices*, 22(7):102–112, July 1987.

[72] Sufrin and Woodcock. Towards the formal specification of a simple programming support environment. *Software Engineering Journal*, 2(4):86–94, July 1987.

[73] Sweet. The MESA programming environment. *SIGPLAN Notices*, 20(7):216–229, July 1985.

[74] Teitelbaum and Reps. The Cornell program synthesizer: A syntax directed programming environment. *CACM*, 24(9), September 1981.

[75] Teitelman and Masinter. The INTERLISP programming environment. In Barstow, Shrobe, and Sandewall, editors, *Interactive Programming Environments*. McGraw-Hill, 1984.

[76] Thompson. A logic for Miranda. Technical Report 56, University of Kent at Canterbury, Computing Laboratory, 1989.

[77] Thompson. Lawful functions and program verification in Miranda. *Science of Computer Programming*, 13, 1990.

[78] David Turner. Recursion equations as a programming language. In Darlington, Henderson, and Turner, editors, *Functional Programming and Its Applications*, pages 1–29. Cambridge University Press, 1982.

[79] David Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*. Springer-Verlag, 1985.

[80] David Turner. Functional programming and communicating processes. In *Lecture Notes in Computer Science*, volume 259, page 54. Springer-Verlag, 1987.

[81] Raymond Turner. *Constructive Foundations for Functional Languages*. McGraw-Hill, 1991.

[82] van der Linden and Wilson. An interactive debugging environment. *IEEE Micro*, 5(4):18–31, August 1985.

[83] Wadler. Listlessness is better than laziness. *ACM Conference on Lisp and Functional Programming*, 1984.

[84] Wadler. How to replace failure by a list of successes. In *Lecture Notes on Computer Science*, volume 201, pages 113–128. Springer-Verlag, 1985.

[85] Wegner. Dimensions of object-based language design. *SIGPLAN Notices*, 22(12):168–182, 1987.

Appendix A

Syntax and Semantics of FLAIR

This appendix describes the understood interpretations of FLAIR programs presented in the main part of this thesis. Section 1 addresses the lexical issues in parsing programs in the language; Section 2 provides a formal syntax for the language; and Section 3 presents an informal semantics for the language, including some consistency checks placed upon programs in the language.

Extensions to the language, as used in some places in the text are next presented. Section 4 describes extensions needed to take advantage of the system of extended types in Chapter 5; the syntax used for proofs in given in Section 5; and Section 6 concludes the appendix by giving details on extensions used for simplicity in certain places in the text.

A.1 Lexical Analysis

The lexical analyser breaks the input into tokens and passes these to the syntax analyser. Any lines starting in the leftmost column are considered comments, and discarded before further analysis is done, leaving a blank line. If the comment starts with the characters `--` then the blank line is discarded as well. In any other column, the characters `--` introduce a comment, in which case all the characters from the beginning of the `--` to the character immediately preceding the next newline is discarded. Comments formed by starting in the leftmost column, but without an initial `--` must be surrounded by blank lines. A blank line is one which consists entirely of non-printing characters.

The remaining input is broken into tokens in a fairly simple way. Printable characters are grouped into three classes: alphanumerics (upper and lower case letters, numbers, `.`, and `_`), symbols (`+`, `-`, `/`, `*`, `&`, `|`, ...), and punctuation (`(`, `)`, `,`, `[`, `]`, `{` and `}`). Tokens are formed by any string of: consecutive alphanumerics; consecutive symbols; or single punctuation characters. These tokens may therefore be described by the three regular expressions:

```
[A-Za-z0-9._]+
[+/*&|-...]+
[()[]]
```

Additionally, the characters ' and " are used in pairs to introduce character and string constants respectively. These are not allowed to extend beyond the end of a line. The character \ is treated specially: it is not permitted outside of string or character constants; inside it is used to introduce special (generally non-printable) characters, by starting a multi-character sequence.

Non-printing characters are generally discarded. The exceptions are the newline, space and TAB characters. These are processed in the following way:

- between tokens on the same line, newline characters do not appear, and space and TAB characters are ignored.
- at the beginning of a file, or after a newline character, space and TAB characters are counted to generate an indentation for the next line. The contribution that TAB characters make to this count is implementation-dependent. The *previous line* and *previous indent* refer to the line, and indentation of the line, preceding the newline. Four outcomes are possible:
  - if the new line has the same indent as the previous line, a `BREAK-LINE` token is generated.
  - if the new line has greater indent than the previous line, then it is considered a continuation line, no token is generated, and subsequently this line is considered to have the same indent as the previous line.
  - if the new line has less indent than the previous line, then entries on the indent stack are popped until an entry is found with a matching indent level. An error results if no matching entry is found. As each entry is popped, a pair of tokens, `BREAK-LINE`, `END-IND` are generated.
  - if the new line is blank, then the indentation for the next non-blank line is calculated. If the indentation is the same or less than the previous indent, then the actions are the same as if no blank line had appeared. If the new line is indented further than the previous line, then the previous indent is pushed on the indent stack (creating it if necessary) and the token `BEGIN-IND` is generated.

At the beginning of the file, a `BEGIN-IND` token is generated, and at the end, the indent stack is unwound, if necessary an extra `BREAK-LINE` token is generated, and finally a `END-IND` token is generated.

A.2 Syntax

A program in the FLAIR language consists of declarations, grouped according to the layout described in the section above. There are three types of declarations: types, parameters, and functions. Here the syntax `[...]` is used to indicate 0 or more repetitions of `...`, `{...}` represents 1 or more repetitions and `...|...` represents alternatives. A character in quotes `' '` is a single punctuation character, tokens in bold are specific symbols or identifiers, and subscript annotations to identifiers indicate the intended use of the name in this scope.

prog `= BEGIN-IND [` *decl* `BREAK-LINE ] END-IND`

```
decl        = type-decl
            | fn-decl
type-decl   = data id_type-name [ type-var ] =
                    id_cons [ type-expr ] [ | id_cons [ type-expr ]]
            | type id_type-name [ type-var ] = type-expr
type-var    = id_poly-var
type-expr   = id_type-name
            | '(' id_type-name [ type-expr | type-var ] ')'
fn-decl     = tok_fn [ patt ] = expr opt-prog
patt        = id_parm | id_cons | '(' id_cons [ patt ] ')'
expr        = tok | const | '(' [ expr ] ')'
opt-prog    =
            | prog
tok         = id | sym
```

A.3 Semantics

The declarations constituting a FLAIR program generate two sets of information: the type information relates to the construction and grouping of data values and the function information relates to function values, and thus the relationships between data values.

A *data* declaration introduces new values, and an forms a grouping for all the defined values. For each element of the right side of the declaration (separated by `|`), the introduced values are of the form `constructor args` where constructor is a new constructor identifier $\texttt{id}_{cons}$ and `args` are values of each of the types `type-expr` respectively. Any values so generated and all the approximations to them, are included in the data type identified by the data type name on the left hand side of the definition. If any polymorphic variables are specified after the type name, these may appear in the type expressions on the right hand side of the declaration. Any given type name, and any given constructor name, may only be introduced once in any scope in this way, and these names may not be used in any other form of declaration in the same scope.

A *type* declaration introduces a new collection of values, as a set combination of the sets of values as described in Chapter 5. As with *data* declarations, any polymorphic variables introduced on the left hand side may appear in the type expressions on the right hand side, although care must be taken not to violate the rules laid down in Section 5.3. Any given type name may only be introduced once in any scope in this way, and this name may not be used in any other form of declaration in the same scope.

A *type expression* is a type name, as defined by a data or type declaration, together with polymorphic variables or type expressions, which correspond to the number of variables cited in the declaration. Any cited type variables must have understood meanings at the point where the expression occurs.

A *function declaration* specifies that a particular symbol in the given scope is to be associated with a function value. The right hand side is an expression which may contain variables bound by their appearance in the patterns of this (or any enclosing) definitions, as well as constants defined by other declarations. The constants available to a function declaration may be extended by enclosing more declarations in an inner block which has the same syntax and semantics as

the program. Nested scopes may redefine function names by redeclaring them in an inner scope. In this case, the definition used is the innermost one currently in scope. This includes the definitions in the block associated with the current expression, and all of the blocks nested outside the current expression, but not any other blocks nested inside.

If two definitions are given for the same name in the same scope, then they are treated as defining the function over different domains of values, and the defined function is the intersection of the declared functions. If the definitions overlap, then the value of the function in the overlapping regions is implementation-dependent, but will be one of the defined values, or an error. If no definition is given for a particular value, then the function is undefined on that value.

A *pattern* consists of a template value: it is defined as a constructor applied to arguments. The number and type of the arguments must match the number and type of the arguments defined for the constructor in the data declaration. The arguments may be patterns, or they may be parameters.

An *expression* may be a constant, a parameter, or a function value applied to arguments. In the latter case, a value is identified by isolating the element of the function range corresponding to the appropriate elements of the domain. By currying, the application may result in a function value, a data value, or an error value, depending on the number of arguments supplied.

A.3.1 Pattern Matching

Pattern matching is performed upon entering a function. There are three objectives in pattern matching: to select one of several equations to use to define the value of the function; to ensure that this equation completely matches the values being passed in to the function; and to uncover the bindings for all the variables in the patterns so that they may be used in the expression on the right hand side of the selected equation.

In order to do this, it is necessary to evaluate the arguments to the function so that the structure may be seen down to the same level as is visible in the patterns, to ensure they match. This guarantees the second objective. In doing this it is of course necessary to determine which is the correct pattern by evaluating each of the arguments sufficiently far enough to decide which case applies, thus fulfilling the first objective. Because the patterns are defined as a constructor applied either directly to variables, or recursively to other patterns, fulfilling the second objective automatically fulfills the third objective: once the structure has been verified, the expressions have been evaluated sufficiently far to extract all the variable bindings.

Flair does not lay down a particular method of selection beyond this, neither does it specify a particular order in which the arguments and sub-arguments should be evaluated. Chapter 3 however, does give a guide on how a generally optimal solution may be generated, and gives an intermediate code for specifying how pattern matching may proceed. We use this intermediate code to produce a semantics for pattern matched functions.

The technique we use is to define a set of non-overlapping equations for the function, based around the HESF definition for pattern matching as described in Chapter 3. In place of the formal arguments or patterns to the function, we write special symbols, `_` for anything, $\perp$ for bottom, `x'` (where `x` is a variable) to indicate any value which has not already appeared (which range of values

can be made precise using the extended types of Chapter 5), and patterns such as `(Cons _ Nil)` to represent sets of values.

The equations are generated from the HESF by examining this code. It is in general, as has been noted, very repetitive, consisting of head-evaluations followed by switches, with nested head-evaluations, ending in equation selection. The translation proceeds by observing that each possible value (including ⊥) can be inserted in place of the relevant argument during the switch. Unswitched arguments are replaced by `_`, and `*` is replaced by a suitable variable `x'`. ⊥ is always substituted for first, and will always produce the result of ⊥.

As an example, consider the HESF code for `mink` (see Figure 3.14), we can construct the following equations. For brevity, E_n is written in place of the full expression with appropriate substitutions.

```
mink ⊥ _ = ⊥
mink Zero _ = E1
mink (Succ _) ⊥ = ⊥
mink (Succ _) Nil = E2
mink (Succ _) (Cons _ ⊥) = ⊥
mink (Succ _) (Cons _ Nil) = E3
mink (Succ ⊥) (Cons _ (Cons _ _)) = ⊥
mink (Succ Zero) (Cons _ (Cons _ _)) = E4
mink (Succ n') (Cons _ (Cons _ _)) = E5
mink (Succ ⊥) l' = ⊥
mink (Succ Zero) l' = E4
mink n' ⊥ = ⊥
mink n' Nil = E2
mink n' (Cons _ ⊥) = ⊥
mink n' (Cons _ Nil) = ⊥
```

These equations give a complete specification of the function `mink` for each of the possible elements of the input domains.

A.4 Extended Type Extensions

In order to take advantage of the extended type notations of Chapter 5, it is necessary to add extra cases to the definitions of type declarations (for `edata`, `sdata`, `etype` and `stype`), and declarations (for the parameter declarations). An extra production, for extended type expressions, is also added.

A *parameter* declaration declares the type of variables to be used in the function definitions. This says that one or more variables are restricted to holding values belonging to the specified type. Because no polymorphic variables are globally bound, they may not be used in the type expression[1]. Any given parameter name may only be introduced once in any scope in this way, and this name may not be used in any other form of declaration in the same scope.

decl = ...
| *parm-decl*

[1] Although see Section 5.8.2 on a possible interpretation of this, and further extensions to the language

type-decl = ...
| edata $\texttt{id}_{type-name}$ [*type-var*] =
$\texttt{id}_{cons}$ [*type-expr*] [| $\texttt{id}_{cons}$ [*type-expr*]]
| sdata $\texttt{id}_{type-name}$ [*type-var*] =
$\texttt{id}_{cons}$ [*type-expr*] [| $\texttt{id}_{cons}$ [*type-expr*]]
| etype $\texttt{id}_{type-name}$ [*type-var*] = *ext-type-expr*
| stype $\texttt{id}_{type-name}$ [*type-var*] = *ext-type-expr*

parm-decl = $\texttt{id}_{parm}$ [',' $\texttt{id}_{parm}$] :: *type-expr*

ext-type-expr = *type-expr*
| '(' *ext-type-expr* ')'
| *ext-type-expr* '|' *ext-type-expr*
| *ext-type-expr* '&' *ext-type-expr*
| *ext-type-expr* '-' *ext-type-expr*

A.5 Syntax of Proofs

The lexical analysis of proofs is much like the analysis of FLAIR programs. However, the symbol ‘ is valid as a character in identifiers. The objective is to allow for qualified names such as `‘trees‘++` to be included in proofs. These qualified names remove the normal dependency on and restrictions of the scoping rules in FLAIR programs. The symbol `@` introduces the notation describing references to subexpressions in proofs. The behaviour of white space is also different: an inner block starts after the symbol `<==`, which must always be followed by a newline; blank lines are no longer relevant. The symbol `<==` by itself on a line separates two inner blocks with the same parent. This is necessary to separate two sub-proofs. Both sub-proofs and the `<==` symbol must have the same indentation.

```
proof      = flair-expr <== BREAK-LINE sp
step       = id_rule [ arg ] opt-rsp
arg        = '(' [ arg ] ')'
           | id
           | ref
           | '[' ']'
           | '[' arg [ ',' arg ] ']'
opt-rsp    = BREAK-LINE
           | <== BREAK-LINE rsp
rsp        = sp <== rsp
           | sp
sp         = BEGIN-IND [ step ] END-IND
```

A.6 Extensions

For simplicity at some points in the text extensions to the basic FLAIR language as described have been used.

Lists are not always quoted as applications of the constructor Cons to arguments. Instead, display lists, ranges and ZF notation as found in Miranda [79] are used.

Another extension used in the section on proof rules is conformal definitions. In Flair, these are not supported; instead, the alternative of using first, second, etc. functions in possible. However, while this simplifies the language, it makes writing clear programs more complex, so in the text conformal definitions have been used with their understood meaning being a translation back into FLAIR as described above. For example, the definition

```
(x,y) = f w
```

would be transformed to

```
v = f w
x = first v
y = second v
```

Appendix B

Simple Parser for the Flair Language

This appendix contains a simple parser for the functional language Flair. Its intent is not as part of a full compiler, but merely as a simple means of entering programs for the algorithms in Appendices C and D. Some features of this parser which are specifically not covered are:

- Robust syntax checking: some syntax checking is done, when not to do it would either leave the parser confused, or would leave another function acting after the parse confused.
- User-friendly error messages: again, the error messages provided indicate the line which caused the error; in general the actual cause of the error is not indicated.
- Type inference, type checking, undefined variables etc, are not handled.

Furthermore, some restrictions are placed on the Flair language, and implemented by the parser:

- it is illegal to use the same parameter in two nested definitions (this is because it is non-trivial to uniquely rename the parameters when lifting).
- it is only possible to define type information at the top level.
- polymorphism is currently ignored in the type dictionary

In these appendices, a package of sets and maps is assumed to be present. These contain the usual functions associated with finite sets and finite maps (for example, see [39]). The functions associated with sets are:

```
emptyset      -- the empty set
insert    x s -- insert x into s
remove    x s -- remove x from s
union     A B -- form the union of A and B
intersect A B -- form the intersection of A and B
setdiff   A B -- form the set whose elements are in A, but not B
subset    A B -- is A a subset of B ?
in        x s -- is x a member of the set s ?
```

```
listtoset l   -- convert the list l into a set
settolist s   -- convert the set s into a list
setsize   s   -- find the cardinality of the set s
```

The functions associated with maps are:

```
emptymap           -- the empty map
apply        m k   -- the entry in m corresponding to key k
defaultapply v m k -- as apply, but return v if k not in m
update       m k v -- produce new map, like m except key k has value v
updatelist   m l   -- update m using list of pairs l
owr          M N   -- overwrite entries in M with those in N
dom          m     -- recover set of keys in m
rng          m     -- recover set of values in m
domres       m s   -- restrict m to those keys in s
domsub       m s   -- subtract from m the keys in s
rngres       m s   -- restrict m to those values in s
rngsub       m s   -- subtract from m the values in s
maptolist    m     -- convert the map m to a list
listtomap    l     -- convert the list l into a map
mapsize      m     -- find the number of entries in the map m
```

The file `fl.lit.m` contains definitions concerned with the parsed, lifted representation of a functional language.

```
>   %include <local/maps>
>   %include <local/oldsets>

In this version of the compiler, virtually all objects are tree structured.
The definition of trees is an identifier at the node, together with zero or
more subnodes. Identifiers are character strings. The null string is special
and reserved for internal use.

>   ids == [char]
>   trees ::= Tree ids [trees]

Programs are defined as a list of function definitions. This representation
assumes that all functions have been lifted to the outer level.  Each function
definition is represented by one or more cases, each of which is an expression
in the form of an equality between the function name applied to zero or more
patterns and an expression representing the result of that case being applied.
A collected set of patterns is a list of lists of patterns, where each list of
patterns is the same length (the arity of the function). Such a collected set
of patterns consists of the patterns associated with each case of a function
definition.

>   programs == [functions]
>   functions == [cases]
>   cases == exprs
>   exprs == trees
>   patts == [[trees]]
```

The file `ptypes.lit.m` contains definitions used by the parser while parsing and lifting the program.

```
>   %include <local/oldsets>
>   %include <local/maps>

>   %include "fl.lit"

A parsed program (before and during lifting) is represented as a list of
```

```
blocks, each block consisting of a statement (a defining equation, a
declaration, or a type definition) and any associated inner definitions.
renames defines a type used for renaming functions when lifting: the key is the
original name, while the value is a pair consisting of the new name and the
list of arguments (old free variables). ctypes defines the grouping of
characters which happens during lexical analysis, and also the state of the
lexical analyser at any point.

>   blocks ::= Block stats [blocks]
>   stats ::= DefEqn exprs | Decl exprs | TypeDefn exprs
>   renames == maps ids (ids,[ids])
>   ctypes ::= None | Alpha | Symbol | Punc

The following functions are used to identify the type of any given statement.

>   isdef (DefEqn s) = True
>   isdef x = False
>   isdecl (Decl s) = True
>   isdecl x = False
>   istdef (TypeDefn s) = True
>   istdef x = False
```

The file `types.lit.m` provides definitions for the type dictionary.

```
>   %include <local/maps>
>   %include <local/oldsets>
>   %include "fl.lit"

Type data comes in five varieties: constructors (giving their arity; type
definitions (as a list of simple patterns); and parameters (together with an
associated typename); type synonyms; and function declarations.  The simple
patterns in a type definition consist of a constructor name, together with a
list of typenames. The list of typenames should be the length of the arity of
the constructor. The type dictionary is a map of names to data.

>   spatts ::= Simple ids [ids]
>   typedata ::= Cons num | PattDef [spatts] | Parm ids |
>                Type exprs | TDecl exprs
>   typedicts == maps ids typedata

iscons checks whether a given name is recognised as a constructor in the type
dictionary passed in.

>   iscons :: typedicts->ids->bool
>   iscons types n =
>     check td
>     where
>       td = defaultapply (Parm "") types n
>       check (Cons n) = True
>       check t = False

isparm checks whether a given name is recognised as a parameter in the
specified type dictionary

>   isparm :: typedicts->ids->bool
>   isparm types s =
>     check td
>     where
>       td = defaultapply (Cons 0) types s
>       check (Parm v) = True
>       check t = False

lazyparm tests whether a parameter s is of a type which contains the value
```

```
bottom (i.e. may not be evaluated for pattern-matching).

>   lazyparm :: typedicts->ids->bool
>   lazyparm types s
>   = False,                              iscons types s
>   = member sps (Simple "bottom" []),    isparm types s
>   = error (s++" is not cons or parm"), otherwise
>     where
>       (PattDef sps) = apply types tname
>       (Parm tname) = apply types s

arity returns the arity of a given constructor c

>   arity :: typedicts->ids->num
>   arity types c =
>     n
>     where
>       (Cons n) = defaultapply (Cons 0) types c
```

The file `flair.lit.m` contains the top-level definition of the Flair compiler.

```
>   %include <local/oldsets>
>   %include <local/maps>

>   %include "fl.lit"
>   %include "ptypes.lit"
>   %include "types.lit"

>   %include "gcode.lit"
>   %include "hesf.lit"
>   %include "tables.lit"

>   %include "parse.lit"
>   %include "pm.lit"
>   %include "presel.lit"

>   %include "tdata.lit.m"

flair compiles a program to produce G-Code.

>   flair f =
>     print ++ chars ++ gcodepr gcode
>     where
>       (fns,td) = parse f
>       gcode = compile pre sel (settolist (rng fns)) td
```

The file `parse.lit.m` handles the job of parsing the program. It uses the subsidiary files `lexer.lit.m` and `lift.lit.m` to do this.

```
>   %include <local/oldsets>
>   %include <local/maps>

>   %include "fl.lit"
>   %include "ptypes.lit"
>   %include "types.lit"

>   %include "lexer.lit"
>   %include "lift.lit"
>   %include "support.lit"

parse produces a list of functions and a type dictionary from a single string,
representing the user program. The first step is to break the string down into
```

lines, then to apply the lexer function to each of the lines. Each of the resulting parsed lines consists of an indent token together with the remaining tokens on the line. The indent token may be used to extract the block structure, and simultaneously the lines may be grouped together and parsed. This results in the block structured program b1. The type information may be extracted, and statements in the program associated with it disposed of. This gives rise to the type dictionary td, and a block-structuted program consisting entirely of defining equations, b2. This program may then be lifted to give a list of defining equations, and then the functions may be grouped together into a map from function names to definitions.

```
>   parse f =
>     (fns,extr_parms (dom fns) td)
>     where
>       b1 = extr_blocks (map lexer (lines f))
>       (td,b2) = extr_typeinfo b1
>       b3 = lift td b2
>       fns = extr_fns b3
```

extr_blocks checks that the returned value from block is accurate -- that means that no characters remain to be processed, and the rules on inner blocks have been observed.

```
>   extr_blocks f
>   = error "blocking incorrect",        r ~= []
>   = error "illegal nested statements", ~ and (map (defblkonly True) bl)
>   = bl,                                otherwise
>     where
>       (bl,r) = block f
```

block takes a list of lines of tokens, disposes of all those which are either blank or comments (a comment is any line which starts in column 1). If no lines remain, then no blocks may be formed. Otherwise, the remaining lines are grouped according to indentation, and finally any lines not grouped are passed to a recursive call of block. Any lines with less indentation than the first line of this block are returned as unprocessed.

```
>   block ls
>   = ([],[]), indls = []
>   = (b:bs,remls), otherwise
>     where
>       indls = dropwhile ind0 ls
>       k = ind (hd indls)
>       thesels = [ l | l<-indls; ind l = 0 \/ ind l >=k]
>       remls = drop (#thesels) indls
>       (b,rem1) = group k thesels
>       (bs,empty) = block rem1
```

group groups lines according to the following simple system: any lines with greater indent that the first, and immediately adjacent (not separated by blank lines or comments) are taken to be a continuation of the first line. Any lines with greater indent that the first but separated by a blank line or a comment from it, form an inner block, processed recursively by block. Any lines of equal indent to this one are returned as unprocessed.

```
>   group k ls =
>     (Block (parse_expr (concat conts)) indbs,rem2)
>     where
>       conts = map tl (hd ls:takewhile iscont (tl ls))
>       rem1 = drop (#conts) ls
>       inds = takewhile isind rem1
>       rem2 = drop (#inds) rem1
```

```
>       iscont l = ind l > k
>       isind l = ind l = 0 \/ ind l > k
>       (indbs,remind) = block inds
```

extr_typeinfo recovers all the type information from a block. This consists of finding all declarations and type definitions and generating the appropriate structures. All the definitions must appear at the top level. The various cases are identified and collated, and any type information statements are removed from the new block which is generated, leaving just the defining equations. The final step in collecting the type information is to extract the constructor definitions from the type definitions by patterns.

```
>   extr_typeinfo b =
>     (extr_cons td,nb)
>     where
>       (td,nb) = collate (listtomap []) [] b
>       extr_cons m =
>         updatelist m (get_cons [simps | (t,PattDef simps) <- maptolist m])
>       get_cons l = mkset [(c,Cons (#a)) | (Simple c a) <- concat l]
>       collate typ rest [] = (typ,reverse rest)
>       collate typ rest (Block s ibs:bs) =
>         collate newtyp newrest bs
>         where
>           newtyp
>           = typ, isdef s
>           = updatelist typ (zip2 vids (repeat (TDecl t))), isdecl s
>           = update typ f (PattDef simps), istdef s
>             where
>               (Decl (Tree "::" [Tree "," vs,t])) = s
>               vids = [v | (Tree v []) <- vs]
>               (TypeDefn (Tree "=" [Tree f as,Tree "|" patts])) = s
>               simps = [ Simple f [v|(Tree v [])<-as] | (Tree f as) <- patts]
>           newrest
>           = Block s ibs:rest, isdef s
>           = rest, otherwise
```

extr_parms converts declarations which appear to be of function types, but for which no function is defined into parameter declarations.

```
>   extr_parms s td =
>     listtomap (
>       [ (v, Parm t) | (v,TDecl (Tree t [])) <- tdl; ~in t s]++
>       [ (v, te)     | (v,te) <- tdl; ~ist te \/ ~in (gt te) s])
>       where
>         tdl = maptolist td
>         ist (TDecl (Tree t [])) = True
>         ist te = False
>         gt (TDecl (Tree t [])) = t
```

extr_fns takes a lifted block containing only function defining equations, and collates them to form a map from function names to a list of defining equations.

```
>   extr_fns b =
>     collate (listtomap []) b
>     where
>       collate fns [] = fns
>       collate fns (Block s []:bs) =
>         collate newfns bs
>         where
>           newfns
>           = update fns f (defs++[Tree "=" [Tree f as,t]]), isdef s
```

```
>           = fns, otherwise
>             where
>               (DefEqn (Tree "=" [Tree f as,t])) = s
>               defs = defaultapply [] fns f
```

parse_expr takes a list of symbols, identifies which case applies, and then
attempts to parse the symbols into an internal structure.

```
>   parse_expr syms
>   = error "no tokens on line", syms = []
>   = pdata, member datas (hd syms)
>   = pdecl, member syms declsym
>   = pdefn, member syms defnsym
>     where
>       datas = ["data","type"]
>       declsym = "::"
>       defnsym = "="
```

The case for data declarations is

data type parm* = Cons types { | Cons types } *

```
>       pdata =
>         TypeDefn (Tree "=" [Tree (syms!1) [Tree p []|p<-ps],Tree "|" patts])
>         where
>           ps = takewhile (~= "=") (drop 2 syms)
>           patts = gtypes (drop (#ps+3) syms)
>           gtypes [] = []
>           gtypes ss
>           = serror syms, hd ss = "|" \/ rs ~= []
>           = e:es,        otherwise
>             where
>               (e,rs) = ptype vs
>               vs = takewhile (~="|") ss
>               es = gtypes (drop (#vs+1) ss)
```

The case for function and parameter declarations is

var {, var} * :: type {type}* {-> type {type}* }*

```
>       pdecl =
>         Decl (Tree "::" [Tree "," [Tree v [] | v<-vars],tname])
>         where
>           vars
>           = serror syms, #vs mod 2 = 0 \/ or [v ~= "," | v<-evens vs]
>           = odds vs, otherwise
>             where vs = takewhile (~="::") syms
>           tname = pftype (drop (2*#vars) syms)
```

The case for defining equations is

f arg * = fbex

[where arg = fbex ; fbex = term term * ; term = name | '(' fbex ')']

```
>       pdefn =
>         DefEqn (Tree "=" [lhs,rhs])
>         where
>           lhssyms = (takewhile (~="=") syms)
>           rhssyms = drop (#lhssyms+1) syms
>           lhs = pfbex lhssyms
>           rhs = pfbex rhssyms
```

```
>           pfbex ss
>           = serror syms, rs ~= []
>           = Tree h as,   otherwise
>             where
>               (es,rs) = fbex ss
>               (h,as) = despine es
>           fbex ss
>           = ([],ss),    hd ss = ")"
>           = ([e],[]),   rs = []
>           = (e:es,rs1), otherwise
>             where
>               (e,rs) = term ss
>            (es,rs1) = fbex rs
>           term ss
>           = (Tree (hd ss) [],tl ss), hd ss ~= "(" & hd ss ~= ")"
>           = (Tree h as,tl rs),       hd ss  = "(" & rs ~= [] & hd rs  = ")"
>           = serror syms,             hd ss  = "(" & (rs = [] \/ hd rs ~= ")")
>             where
>               (es,rs) = fbex (tl ss)
>               (h,as) = despine es
>           despine (Tree h []:ts) = (h,ts)
>           despine (Tree h es:ts) = (h,es++ts)
```

The case for a type declaration is handled by the two definitions

```
t1 ::= t2 {-> t2}*
t2 ::= typename typename * | '(' t1 ')'
```

```
>       pftype ss
>       = serror syms, rs ~= []
>       = t,           otherwise
>         where (t,rs) = ptype ss
>       ptype ss
>       = serror syms,       #ts < 1
>       = (uniq ts,rs),      #ts = 1
>       = (Tree "->" ts,rs), #ts > 1
>         where (ts,rs) = ptypes ss
>       ptypes ss
>       = ([e],[]),   rs1 = []
>       = (e:es,rs2), hd rs1 = "->"
>       = ([e],rs1),  hd rs1 = ")"
>         where
>           (e,rs1) = ptype2 ss
>           (es,rs2) = ptypes (tl rs1)
>       ptype2 ss
>       = (e,tl rs),                    hd ss = "(" & rs ~= [] & hd rs = ")"
>       = (Tree (tns!0) [Tree t [] | t<-tl tns],drop (#tns) ss), otherwise
>         where
>           (e,rs) = ptype (tl ss)
>           tns = takewhile isname ss
>           isname s = s~="->" & s~=")" & s~= "("
```

defblkonly checks that only defining equations are allowed to have or appear in inner blocks.

```
>   defblkonly f (Block (DefEqn ex) bs) = and (map (defblkonly False) bs)
>   defblkonly f (Block s []) = f
>   defblkonly f (Block s bs) = False
```

ind0 tests whether a line starts in column 1.
ind returns the indent of a line.

```
>   ind0 t = #t = 1 \/ #(hd t) = 0
>   ind t = #(hd t)
```

The file `lexer.lit.m` handles the lexical analysis of Flair programs.

```
>   %include <local/oldsets>
>   %include <local/maps>

>   %include "fl.lit"
>   %include "ptypes.lit"

lexer analyses a string (assumed to be a single line of source text) and
produces a list of tokens, each token being simply a string of characters.
The first token is always the initial space; any remaining spaces on the line
are ignored.

>   lexer s = takewhile (=' ') s:lex "" None (dropwhile (=' ') s)

lex produces the current token by extending the token w until no more
characters may be added. The end comes when there are no more characters in the
string, or when a character of different type (e.g Symbol vs Alpha) is
encountered, or when any character of type Punc or None is encountered.

>   lex w t []
>   = [reverse w], w ~= ""
>   = [],          otherwise
>   lex w t (c:s)
>   = reverse w:lex "" newt s,  newt = None & w~=""
>   = reverse w:lex [c] newt s, (t ~= newt \/ t = Punc) & w~=""
>   = lex "" newt s,            newt = None
>   = lex [c] newt s,           (newt ~= t \/ t = Punc)
>   = lex (c:w) t s,            otherwise
>     where
>       newt = stype c

stype analyses a single character and reports on its type. It does so by
consulting an array which gives the type for each ASCII character. Backquote is
used internally so is disallowed. Any disallowed character (. = None) is
assumed to be equivalent to a space character (except when it occurs at the
beginning of a line).

>   stype c =
>     ctype!(code c)
>     where
>       ctype = map ccode cct
>       ccode '.' = None
>       ccode 's' = Symbol
>       ccode 'p' = Punc
>       ccode 'a' = Alpha
>||
>||          !"#$%&'()*+,-./0123456789:;<=>?
>||         @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
>||         `abcdefghijklmnopqrstuvwxyz{|}~
>       cct =
>         "................................" ++
>         ".sssssssppsspsasaaaaaaaaaassssss" ++
>         ".aaaaaaaaaaaaaaaaaaaaaaaaaapsps." ++
>         ".aaaaaaaaaaaaaaaaaaaaaaaaaapsps."

serror produces a syntax error.

>   serror syms = error ("Syntax Error: line"++concat [" " ++ s | s<-syms])
```

The file `lift.lit.m` is responsible for the lifting of any inner blocks, together with any associated renaming and adding parameters to function definitions.

```
>   %include <local/oldsets>
>   %include <local/maps>

>   %include "fl.lit"
>   %include "ptypes.lit"
>   %include "types.lit"
```

lift takes a list of blocks, together with their inner blocks and lifts all the defining equations in the inner blocks (nothing else is permitted there) to the enclosing block. At the same time, all the lifted definitions are renamed by prepending the name of the enclosing function and a backquote. The free variables are collected and added to the definition, and all the uses of the function are renamed and reworked to include the new arguments.

```
>   lift td [] = []
>   lift td (Block s bs:rbs) =
>     Block (renstat m s) []:lift1 m bs1++lift td rbs
>     where
>       bs1 = lift td bs
>       (DefEqn (Tree "=" [Tree f as,ex])) = s
>       m = renmap f (fvs td as) bs1
>       fvs td as =
>         [ v | (Tree v [])<-as; ~iscons td v]++
>         concat [ fvs td ras | (Tree c ras)<-as]
```

renmap builds the renaming map for all the functions in a block. It takes the name of the enclosing function and the free variables in the patterns of the defining equation. It uses this to build a map from names to new names and additional arguments. There should be a dependency analysis in here so that only the needed names are included.

```
>   renmap f vs [] = listtomap []
>   renmap f vs (Block (DefEqn (Tree "=" [Tree g as,ex])) []:bs) =
>     update m g (f++'`':g,vs)
>     where
>       m = renmap f vs bs
```

renstat renames the defining equation which introduces the inner block

```
>   renstat m (DefEqn (Tree "=" [Tree f as,ex])) =
>     (DefEqn (Tree "=" [Tree f as,renex m ex]))
```

renex renames a right hand side expression

```
>   renex m (Tree h as) =
>     Tree newh ([Tree a [] | a<-newas]++renas)
>     where
>       (newh,newas) = defaultapply (h,[]) m h
>       renas = [ renex m a | a<-as ]
```

rendef renames an entire inner definition.

```
>   rendef m (DefEqn (Tree "=" [Tree f as,ex])) =
>     (DefEqn (Tree "=" [Tree g (newas++as),renex m ex]))
>     where
>       (g,exas) = defaultapply (f,[]) m f
>       newas = [Tree a [] | a<-exas]
```

lift1 renames and extracts all the definitions from an inner block.

```
>   lift1 m [] = []
>   lift1 m (Block s bs:rbs) =
>     Block (rendef m s) bs:lift1 m rbs
```

The file `tdata.lit.m` contains test data for the parser, and for the other appendices.

```
>   %include "tables.lit"

Test Program: this program supplies the functions mirror, reverse, and mink,
along with the required data types and supporting functions.

>   prog = "\
>    data trees k = Empty | Tree k ints k\n\
>    data lists = Nil | Cons ints lists\n\
>    data ints = Zero | Succ ints\n\
>\n\
>    t1, t2 :: trees ints\n\
>    i,a,b,k :: ints\n\
>    l :: lists\n\
>\n\
>    resexpr =\n\
>       show (mink (Succ (Succ (Succ Zero)))\n\
>              (Cons Zero (Cons (Succ (Succ Zero)) (Cons (Succ Zero) Nil))))\n\
>\n\
>    mirror :: (trees ints)->(trees ints)\n\
>    mirror (Tree t1 i t2) = Tree (mirror t2) i (mirror t1)\n\
>\n\
>    reverse Nil = Nil\n\
>    reverse (Cons i l) = append (reverse l) i\n\
>\n\
>    mink Zero l = Zero\n\
>    mink k Nil = Zero\n\
>    mink k (Cons a Nil)= a\n\
>    mink (Succ Zero) l =\n\
>        hd l\n\
>    mink (Succ k) (Cons a (Cons b l)) =\n\
>        mink k (if (<= a b) a b) l\n\
>\n\
>    <= Zero k = True\n\
>    <= k Zero = False\n\
>    <= (Succ k) (Succ l) = <= k l\n\
>\n\
>    if True t f  = t\n\
>    if False t f = f\n\
>\n\
>    hd (Cons h l) = h\n\
>\n\
>    show Zero = Cons zchar Nil\n\
>    show (Succ k) = append (show k) qchar\n\
>\n\
>    append Nil c = Cons c Nil\n\
>    append (Cons c1 s) c =\n\
>        Cons c1 (append s c)\n\
>"

The print function is used to output a list of characters to the terminal. It's
use of the write instruction (which involves a side-effect) makes it
non-functional.

>   print = "\
>FUNCTION result\n\
```

```
>\theap 2\n\
>\tstack 1\n\
>\tnew _p\n\
>\tfill APPLY0\n\
>\tfill resexpr\n\
>\tpush _p\n\
>\tjump print\n\
>\n\
>FUNCTION print\n\
>\theap 2\n\
>\tstack 1\n\
>\targs 1\n\
>\tpop p1\n\
>\tpush print1\n\
>\tenter p1\n\
>\n\
>\tdata 1 0\n\
>RETURN print1\n\
>\tpop print1\n\
>\ttry _tNil\n\
>\tswitch print2\n\
>\tuntry _tNil\n\
>\ttry _tCons\n\
>\tswitch print3\n\
>\tuntry _tCons\n\
>\tjump _misscase\n\
>\n\
>CASE print2\n\
>\twrite 10\n\
>\tstop 0\n\
>\n\
>CASE print3\n\
>\tget _tmp p1 p2\n\
>\tpush p2 print4\n\
>\tenter p1\n\
>\n\
>\tdata 2 2\n\
>RETURN print4\n\
>\tpop print4 p2\n\
>\tget _tmp d1\n\
>\twrite d1\n\
>\tpush p2\n\
>\tjump print\n\
>\n\
>\tdata 2 0\n\
>LABEL APPLY0\n\
>\tget APPLY0 d1\n\
>\tjump d1\n\
>\n\
>\tdata 3 4\n\
>LABEL APPLY1\n\
>\tstack 1\n\
>\tget APPLY1 d1 p1\n\
>\tpush p1\n\
>\tjump d1\n\
>\n\
>\tdata 4 12\n\
>LABEL APPLY2\n\
>\tstack 2\n\
>\tget APPLY2 d1 p1 p2\n\
>\tpush p2 p1\n\
>\tjump d1\n\
```

```
>\n\
>\tdata 2 2\n\
>LABEL INDIRECT\n\
>\tget INDIRECT p1\n\
>\tenter p1\n\
>\n"
```

Characters are supported directly, by use of two constructor functions, zchar and qchar, to represent the characters 0 and ' respectively.

```
>   chars = "\
>EQUATE _tchar 999\n\
>\n\
>\tdata 2 0\n\
>LABEL _char\n\
>\treturn _tchar\n\
>\n\
>FUNCTION zchar\n\
>\tnew _p\n\
>\tfill _char\n\
>\tfill 48\n\
>\tenter _p\n\
>\n\
>FUNCTION qchar\n\
>\tnew _p\n\
>\tfill _char\n\
>\tfill 39\n\
>\tenter _p\n\
>\n"
```

The tables gentab and seltab are the generated and selected tables for the function mink.

```
>   gentab =
>     Table [(["Zero",""],Eqns [1]),(["Zero","Nil"],Eqns [1,2]),
>     (["Zero","Cons"],Table [(["",""],Eqns [1]),(["","Nil"],Eqns [1,3])]),
>     (["",""],Eqns []),(["","Nil"],Eqns [2]),(["","Cons"],Table [(["","Nil"],
>     Eqns [3])]),(["Succ",""],Table [(["Zero"],Eqns [4])]),(["Succ","Nil"],
>     Table [([""],Eqns [2]),(["Zero"],Eqns [2,4])]),(["Succ","Cons"],
>     Table [(["","","Nil"],Eqns [3]),(["","",""],Eqns []),(["","","Cons"],
>     Table [(["",""],Eqns [5])]),(["Zero","","Nil"],Eqns [3,4]),
>     (["Zero","",""],Eqns [4]),(["Zero","","Cons"],Table [(["",""],
>     Eqns [4,5])])])]

>   seltab =
>     Table [(["Zero",""],Eqns [1]),(["Zero","Nil"],Eqns [1]),(["Zero","Cons"],
>     Table [(["",""],Eqns [1]),(["","Nil"],Eqns [1])]),(["",""],Eqns []),
>     (["","Nil"],Eqns [2]),(["","Cons"],Table [(["","Nil"],Eqns [3])]),
>     (["Succ",""],Table [(["Zero"],Eqns [4])]),(["Succ","Nil"],
>     Table [([""],Eqns [2]),(["Zero"],Eqns [2])]),(["Succ","Cons"],
>     Table [(["","","Nil"],Eqns [3]),(["","",""],Eqns []),(["","","Cons"],
>     Table [(["",""],Eqns [5])]),(["Zero","","Nil"],Eqns [3]),(["Zero","",""],
>     Eqns [4]),(["Zero","","Cons"],Table [(["",""],Eqns [4])])])]
```

basetd is some sample test data for the function fbasetypes.

```
>   basetd = "\
>    data int = Z | S nat | P nonpos\n\
>    data nat = Z | S nat\n\
>    data nonpos = Z | P nonpos\n\
>    i :: int\n\
>    n :: nat\n\
```

```
>    np :: nonpos\n\
>"

inftd is some sample test data for the function testcycle

>   inftd = "\
>    data p = Bottom | A x q\n\
>    data q = B r y\n\
>    data r = C p\n\
>\n\
>    data s = A x t\n\
>    data t = Bottom | B t y | C s\n\
>\n\
>    data x = Z | S x\n\
>    data y = N | L x y\n\
>"
```

Appendix C

Code Generation for Flair

This appendix provides code to compile a parsed, lifted program, stored as a list of function definitions and a type dictionary, into G-Code. It handles the separation of patterns, the generation of pattern-matching code, and the generation of code to evaluate expressions.

G-Code is represented as a Miranda algebraic type in the file `gcode.lit.m`, together with a function `showhesf` for printing it (and thus preparing it for input to the interpreter of Appendix E.

```
>   gcode ::=
>       Function [char] | Label [char] |
>       Continue [char] | Case [char] |
>       Data num num | Heap num | Stack num | Args num |
>       Storenode [char] | Get [char] | Put [char] |
>       New [char] | Fill [char] |
>       Pop [char] | Push [char] |
>       Try [char] | Swi [char] | Untry [char] |
>       Jump [char] | Enter [char] | Return [char] |
>       Equate [char] num

>   gcodepr gcs =
>     lay (filter (~="") (map gcpr gcs))
>     where
>       gcpr (Function s) = "\nFUNCTION "++s
>       gcpr (Case s) = "\nCASE "++s
>       gcpr (Data s l) = "\n\tdata "++show s++" "++show l
>       gcpr (Label s) = "LABEL "++s
>       gcpr (Continue s) = "CONTINUE "++s
>       gcpr (Heap k)
>       = "\theap "++show k, k>0
>       = "",                otherwise
>       gcpr (Stack k)
>       = "\tstack "++show k, k>0
>       = "",                 otherwise
>       gcpr (Args k)
>       = "\targs "++show k, k>0
>       = "",                otherwise
>       gcpr (Storenode s) = "\tsavenode "++s
>       gcpr (Get s) = "\tget "++s
>       gcpr (Put s) = "\tput "++s
>       gcpr (New s) = "\tnew "++s
>       gcpr (Fill s) = "\tfill "++s
>       gcpr (Pop s) = "\tpop "++s
```

```
>       gcpr (Push s) = "\tpush "++s
>       gcpr (Try s) = "\ttry "++s
>       gcpr (Swi s) = "\tswitch "++s
>       gcpr (Untry s) = "\tuntry "++s
>       gcpr (Jump s) = "\tjump "++s
>       gcpr (Enter s) = "\tenter "++s
>       gcpr (Return s) = "\treturn "++s
>       gcpr (Equate s i) = "\nEQUATE "++s++" "++show i
```

The file `hesf.lit.m` contains the definition of HESF intermediate code, and a suitable display function `showhesf`.

```
>   %include <local/maps>
>   %include <local/oldsets>
>   %include "fl.lit"
>   %include "support.lit"

HESF code is a compact means for representing pattern matching code. There
are a total of four instructions, although generally only three are used.
Head head evaluates a given sub-expression of the argument list. Eqn
specifies that pattern matching has finished, and a particular case of the
function has been selected for execution. Switch handles the distribution of
cases after an evaluation, the code to be executed depending on the head
constructor of the evaluated expression. Full fully evaluates a given
sub-expression of the argument list. The sub-expression is identified by a
list of numbers, indicating which element of the list of sub-expressions is
to be chosen at each stage.

>   posn == [num]
>   hesf ::= Head posn hesf | Eqn num | Switch [(ids,hesf)] | Full posn hesf

showhesf pretty-prints HESF code

>   showhesf h    = lay (sh h)
>   sh (Eqn r)    = ['E':show r]
>   sh (Head p h) = ('H':show p):sh h
>   sh (Full p h) = ('F':show p):sh h
>   sh (Switch cs) =
>     concat [indent 4 ((prstar i ++ " : " ++ hd s):tl s) |
>                                              (i,h) <- cs; s <- [sh h]]
```

The file `tables.lit.m` support the tables used for resolving overlapping equations.

```
>   %include <local/maps>
>   %include <local/oldsets>
>   %include "fl.lit"
>   %include "support.lit"

Tables are used to represent the coverage of patterns over equations. There are
two possibilities : there are no constructors to match against, in which case
all possible equations are represented by a list; or constructors may be
matched, which is represented by a table whose indices are the possible
constructor combinations, together with a table representing
the possible matching of the arguments to the constructors.

>   tables ::= Eqns [num] | Table [([ids],tables)]

showtable displays a table, using the sub-function showtab.

>   showtable tab = lay (showtab tab)
```

```
>   showtab (Eqns ks) = [show ks]
>   showtab (Table vs) =
>     concat [(concsp (map prstar c)++":"):(indent 4 (showtab t)) | (c,t)<-vs]
```

The file pm.lit.m generates the code and resolves the pattern matching.

```
>   %include <local/maps>
>   %include <local/oldsets>
>   %include "restrict.lit"
>   %include "fl.lit"
>   %include "types.lit"
>   %include "hesf.lit"
>   %include "gcode.lit"
>   %include "tables.lit"
>   %include "support.lit"

Compile compiles an entire program, given a program represented as a list of
lists of defining equations, an appropriate type dictionary, and two functions
(pre and sel) used to massage the patterns to be in standard form, and then to
reduce the generated table to produce the selected table.

>   compile :: (typedicts->patts->patts)->(tables->tables)->
>                                       programs->typedicts->[gcode]
>   compile pre sel prog types =
>     compdata types ++
>              concat [compfunc pre sel types i p | (i,p)<-zip2 [1..] prog]

Compdata compiles all data type definitions to produce three items: a tag
value, a code pointer, and a constructor function.

>   compdata types =
>     concat [compdt c n i |
>            ((c,n),i)<-zip2 [(c,n) | (c,Cons n)<-maptolist types] [0..]]
>     where
>       compdt c n i =
>         tag:code++cons
>         where
>           tag = Equate ("_t"++c) i
>           code = [mkdata (n+1),Label ("_p"++c),Return ("_t"++c)]
>           cons =
>             [Function c,Args n,Heap (n+1),New "_tmp1",Fill ("_p"++c)]++
>             concat [[Pop "_tmp2",Fill "_tmp2"] | z<-[1..n]]++
>             [Enter "_tmp1"]

Compfunc compiles one function in a program. It first asserts that the function
definition and type dictionary are consistent and valid, and if not, an error
is produced.  Otherwise, a function header is produced, followed by code to
perform pattern matching, and code to evaluate each of the expressions.

The pattern matching code is produced in two stages.
First, an intermediate form HESF code is produced based on the pattern
intersections, together with type restrictions (which are discarded).
Secondly, the HESF code is translated into G-code.

pdmap is a mapping from positions in the patterns to G-Code variable names.
vmaps is a list of mappings (one per defining equation) from source variable
      names to G-Code variable names.
funcno is a function number used in the creation of unique entry point names.
eps, args, and fvs are lists of entry point names, argument variabel names, and
      free variables in the patterns.

>   compfunc :: (typedicts->patts->patts)->(tables->tables)->typedicts
>                                     ->num->functions->[gcode]
```

```
>   compfunc pre sel types funcno func
>   = error "Illegal Function Definition", ~ assert
>   = intro ++ pattcode ++ eqncode,          otherwise
>     where
>       assert = True
>       intro = [Function f,Args nargs,Stack estack]++pops
>       (pattcode,pstack,pheap,eps1) =
>         genpattcode hlist types funcno pdmap [] hec eps args
>       (eqncode,estack,hlist,eps2) =
>         geneqncode fvs (zip3 [mke funcno e | e<-[1..]] eqs vmaps) eps1
>       (hec,res) = comppatt pre sel types ps

>       (Tree "=" [Tree f p1,eq1]) = hd func
>       ps = [ p | Tree "=" [Tree f p,eq] <- func]
>       eqs = [ eq | Tree "=" [Tree f p,eq] <- func]
>       nargs = #(hd ps)
>       pops = [Pop arg | arg <- args]
>       eps = [mke funcno e | e<-[#eqs+1..]]
>       args = [mkv v | v<-[1..nargs]]
>       fvs = [mkv v | v<-[nargs+(mapsize pdmap)+1..]]
>       pdmap = bpdmap ps nargs
>       vmaps = [bvmap types pdmap p | p<-ps]
```

comppatt takes two functions, a type dictionary, and a two-dimensional array of patterns, and produces HESF code to select an appropriate equation, together with restrictions on the types of the free variables in the equations. The restrictions are represented as an expression. The first function taken by comppatt, pre, is a function to be applied to the patterns before they are processed. This allows language-specific irregularities in the patterns (for example, underapplied constructors) to be removed before pattern matching takes place. The function buildtab is used on the resulting patterns to build a table representing the coverage of the patterns across the equations. The second argument function, sel, is then used to select the desired coverage for the table. In general, the equations will overlap somewhat, and so two equations will appear to apply to the same set of values. The table resulting from the selection function must be the same shape as the input table, but no entry may have more than one equation applying to a single set of values. However, the relationships between values and equations is not otherwise limited: that is, the resulting equation did not need to be in the original list for that set of values, and it is possible to select no equation where one or more were originally present.

A list of parameter/equation references is generated (called lparms) by examining each position of each pattern in turn and deciding whether or not it contains a lazy parameter. The result is a list of (equation,position) pairs where the position is a list of argument positions.

```
>   comppatt :: (typedicts->patts->patts)->(tables->tables)->
>               typedicts->patts->(hesf,[exprs])
>   comppatt pre sel types ps
>   = error "Illegal selection function",        ~ assert
>   = (  genhesf types ps1 lparms [[i] | i<-[1..#(hd ps1)]] table2,
>        genrestr types table1 table2 ps1       ), otherwise
>     where
>       ps1 = pre types ps
>       table1 = buildtab types [] [1..#ps1] ps1
>       table2 = sel table1
>       assert = checkstruct table1 table2
>       lparms = [(e,pn) | (e,p)<-zip2 [1..] ps1; pn<-idlp [] p]
>       idlp r p = concat [id1 (r++[i]) p1 | (i,p1)<-zip2 [1..] p]
>       id1 r (Tree h as)
>       = [r],       lazyparm types h
```

```
>       = idlp r as, otherwise
```

checkstruct checks that the structure of two tables is identical, that
the second table never allows more than one equation to be selected by the
same set of values, and that any equation selected by table 2 must also have
been selected by table 1.

```
>   checkstruct :: tables->tables->bool
>   checkstruct (Eqns ks) (Eqns ls) = (#ls <= 1) & ((ls--ks) = [])
>   checkstruct (Table us) (Table vs) =
>     and [ ui = vi & checkstruct ut vt | ((ui,ut),(vi,vt)) <- zip2 us vs]
>   checkstruct tab1 tab2 = False
```

buildtab constructs a table of equation coverage over the patterns. Firstly,
the head constructors for each of the cases for each argument position are
collected together in a list of lists, idl, and the coverage determined by
taking all the possible combinations of constructors. Parameters are not
included directly in this process, but if any are used in a given argument
position, the special constructor "" is added to produce the coverage
(performed by exparm). If no constructors appear anywhere in any of the
patterns, then all equations apply to all cases of the patterns. These
equation numbers are given in the list ks. The list es holds equation numbers
for which it has already been decided that they hold over all equations with
the given coverage (initially ks is all equations; es is no equations).

If at least one constructor appears in at least one argument position,
buildtab is called recursively with new arguments. These arguments are
calculated in the function rbtab based on the individual elements of the
coverage. The coverage and inner tables are then combined to produce the
final result table. The length of the lists ks and ps are always the same,
since one is a list of equation numbers, and the other the list of patterns
for the equations to be matched.

The new list of universal equations (es) is the old list, extended by all
those which involve no constructors in their patterns (all their arguments are
free variables). The new list of considered equations (ks) is the old list,
with the exception of those, which, in one of their arguments, have a head
constructor which differs from that given in the same position of the
coverage. The new list of patterns is the concatentation of the arguments of
the patterns for the equations which are to be used.

```
>   buildtab :: typedicts->[num]->[num]->patts->tables
>   buildtab types es ks ps
>   = Eqns (es++ks), #idl = 0
>   = Table [(rep (#idl) "",Eqns (es++ks))], and [ i = [""] | i <- idl ]
>   = Table [(cov,rbtab cov) | cov<-coverage], otherwise
>     where
>       idl = [ mkset [exparm h | (Tree h as) <- l] | l <- transpose ps ]
>       exparm h
>       = h,  iscons types h
>       = "", otherwise
>       coverage = xprod idl
>       rbtab cov =
>         buildtab types (es++newes) newks newps
>         where
>           newes = [ e | (e,p) <- zip2 ks ps;
>                         and [ ~iscons types h | Tree h as <- p ]]
>           newks = [ k | (k,p) <- zip2 ks ps; and [ ~iscons types h \/ h = c |
>                            (c,Tree h as)<-zip2 cov p]]  --  newes
>           newps = [ concat [ exargs c t | (c,t) <- zip2 cov p ] |
>                                        (k,p) <- zip2 ks ps; member newks k]
>       exargs c (Tree h as)
```

```
>       = as,                                iscons types h
>       = [],                                c = ""
>       = rep (arity types c) (Tree "" []), otherwise
```

genhesf generates HESF code from a table. If the table represents a single equation and there are no further free variables which need to be extracted from that equation (and it is certain that that equation matches), then that equation is selected, otherwise an argument is evaluated, and a switch performed depending on the resulting constructor. Which column to switch upon is determined by calling choosecol, and the appropriate column position is easily determined by looking up the list of positions passed in as an argument. The list of cases for the HESF switch statement is a list of pairs, each possible constructor for that argument position, together with HESF code generated by recursively calling genhesf with new position and table information. The new position data is calculated by throwing away the item at the selected position and replacing it with one new position name for each argument to the constructor. The new table is calculated by splitting up the old table by constructor, and discarding the old coverage entry for the selected argument position. If the entry for that position is an equation, nothing more is done. However, if the entry is a table, then the entries in the table corresponding to the selected constructor (which may be zero if a parameter or constant constructor was selected) are promoted to this level, and the lower table is split across the new coverages of the upper table, ‘‘staying with’’ the constructors they were previously associated with. extract performs the operations of splitting and promoting by extracting a list of triples from the lower coverage: the constructors to be promoted, those not being promoted, and the attendant entries in the lower table. These are collated by promote to produce the new inner tables. Collate works by using a list of all promoted patterns generated by alltus to collect together all the entries in the list of triples which have the same promoted entry. The list of constructors and code pairs for a switch is ordered so that the default case always appears at the end.

isvp is used in the test to see whether we can just return an Eqn or if we need a complete table: isvp takes a list of patterns (corresponding to an equation definition), and tests whether any constructors appear as the heads of the patterns, and if so, returns False. If no constructors appear in any of the heads, there’s nothing left to switch on, so just return the equation.
gppos is used to get the head of a pattern at a given position.

```
>   genhesf :: typedicts->patts->[(num,posn)]->[posn]->tables->hesf
>   genhesf types ps lparms poss (Eqns ks) = Eqn (uniq ks)
>   genhesf types ps lparms poss (Table vs)
>   = Eqn (uniq allks),       conss = [] \/ and [isvp (ps!(k-1)) | k<-allks]
>   = Head pw (Switch cases), otherwise
>     where
>       conss = mkset [c | (cs,e) <- vs; c<-cs; c~=""]
>       allks = getcases (Table vs)
>       which =
>         choosecol (Table vs)
>                   [and [p~=pn | (e,p)<-lparms; member allks e] | pn<-poss]
>       cases =
>         order [(c,genhesf types ps lparms (np c) (ns c st)) |
>                         (c,st)<-split vs which; getcases (Table st) ~= []]
>       np c =
>         take (which-1) poss ++ [pw++[i] | i<-[1..ar c]] ++ drop which poss
>       ns c st = Table (concat [ promote (ar c) cov tab | (cov,tab) <- st])
>       promote a cov (Eqns ks) =
>         [(take (which-1) cov ++ (rep a "") ++ drop which cov,Eqns ks)]
>       promote a cov (Table vs1) =
>         collate [(t ++ pc ++ d,trem,e) | (pc,trem,e) <- extract n a vs1 ]
>         where
```

```
>           t = take (which-1) cov
>           d = drop which cov
>           n = sum (map ar t)
>       extract k a vs1 =
>         [(take a (drop k v), take k v ++ drop (k+a) v,e)| (v,e) <- vs1]
>       collate ts =
>         [(tu1,Table [(tl,e) | (tu,tl,e) <- ts; tu = tu1])| tu1 <- alltus ts]
>       alltus ts = mkset [tu | (tu,tl,e) <- ts]
>       ar "" = 0
>       ar c = arity types c
>       pw = poss!(which-1)
>       order [] = []
>       order [ch] = [ch]
>       order ((c,h):ch:rchs)
>       = ch:rchs++[(c,h)],      c = ""
>       = (c,h):order (ch:rchs), otherwise
>       isvp p = and [ ~iscons types (gppos (Tree "" p) r) | r<-poss]
>       gppos (Tree c as) [] = c
>       gppos (Tree c []) ks = c  || Handles * case.
>       gppos (Tree c as) (k:ks) = gppos (as!(k-1)) ks
```

choosecol selects a column of a table to switch upon based upon a metric applied to the separation in cases that will occur were this column to be switched upon. The list of booleans bs determines which columns may be chosen, depending on lazy elements in the types. No column may be chosen which has no interesting constructors.
colmets is a list of metrics, one for each of the columns of the table. The table is divided up into subtables. Each subtable is in fact the entire table, but split up into a list of subtables in such a way that in one particular column in the subtable, only one constructor appears. ncol is the number of columns in the table.

```
>   choosecol :: tables->[bool]->num
>   choosecol (Table vs) bs
>   = error "No possible choice",                              or bs = False
>   = hd [ k | (k,m) <- colmets; m = min (map snd colmets)], otherwise
>     where
>       colmets =
>         [(i,metric [ getcases (Table vs1) | (c,vs1) <- cst]) |
>                                                    (i,cst)<-subtables]
>       subtables =
>         [(i,split vs i) | i <- [1..ncol];bs!(i-1) & constrs!(i-1)~=[]]
>       ncol = #(fst (hd vs))
>       constrs = [filter (~=[]) (mkset l) | l<-transpose [c | (c,t)<-vs]]
```

split collects all the entries in the table entries vs which have a given constructor in a given column i together producing a list of (constructor,entry) pairs. constrs collects the list of constructors which appear in column i in the table.

```
>  split :: [([ids],tables)]->num->[(ids,[([ids],tables)])]
>  split vs i =
>    [(c,[(cov,tab) | (cov,tab) <- vs; cov!(i-1) = c]) | c <- constrs]
>    where
>      constrs = mkset [cov!(i-1) | (cov,tab) <- vs]
```

metric calculates the relative benefits of choosing one subtable over another.

```
>   metric :: [[num]]->num
>   metric [] = 0
>   metric ls = sum (map sum ls) / #ls
```

```
getcases is given a table, and returns the list of all the possible equations
which could be selected by the table.

>   getcases :: tables->[num]
>   getcases (Eqns ks) = ks
>   getcases (Table vs) = mkset (concat [ getcases ts | (c,ts) <- vs ])

genpattcode generates G-Code from HESF code. There are four cases. The
arguments passed in are:
hlist -- a list of heap requirements for each of the destination CASE blocks
         for the rhs expressions
types -- the type dictionary
funcno -- the unique function id for generating entry points
pm    -- map from argument positions to G-Code variables
pd    -- current position being switched on
HESF  -- the HESF instruction to be executed
eps   -- the free entry points to be used
vs    -- the variables to be saved on the stack

>   genpattcode :: [num]->typedicts->num->maps posn ids->
>                   posn->hesf->[ids]->[ids]->([gcode],num,num,[ids])

For head evaluation, the existing context and return address to a new
continuation block must first be saved, then the suspension entered, then the
new continuation block must be created, which first stores the address of the
heap node resulting from the evaluation, then recovers the saved context.
Finally, the remaining HESF code is compiled recursively.

>   genpattcode hlist types funcno pm pd (Head s c) (ep1:reps) vs =
>     (pushes++newblock++pops++rhec,max[#vs+1,rstack],0,reps1)
>     where
>       pushes = [Push vp | vp <- reverse pvs]
>       newblock =
>        [Enter cvar,mkdata (#pops+1),Continue ep1,Heap rh,Storenode cvar]
>       pops = [Pop vp | vp <- pvs]
>       (rhec,rstack,rh,reps1) =
>         genpattcode hlist types funcno pm s c reps (mkset (cvar:vs))
>       pvs = (ep1:vs)--[cvar]
>       cvar = apply pm s

For equations, a simple transfer of control to the appropriate equation entry
point suffices.

>   genpattcode hlist types funcno pm pd (Eqn k) eps vs =
>     ([Jump (mke funcno k)],0,hlist!(k-1),eps)

For a switch, the current block is finished by adding test-and-switch
instruction groups for each listed constructor. If a default case is
supplied, then this is used as a final default branch, otherwise the error
MISSING CASE is used. A series of blocks is then generated, one for each of
the cases, each consisting of an entry point, recovery of the constructor,
then the arguments in a node, and remaining code corresponding to the
constructor.

>   genpattcode hlist types funcno pm pd (Switch chs) eps vs =
>     (tries++dcase:cases,mstack,mh,eps1)
>     where
>       tries = concat [[Try ("_t"++c),Swi ep,Untry ("_t"++c)] |
>                                    ((c,h),ep) <- zip2 chs eps; c ~= ""]
>       dcase
>       = Jump "misscase", defs = []
>       = Jump (hd defs),  otherwise
>       (cases,mstack,mh,eps1) = gccases (zip2 chs eps) (drop (#chs) eps)
```

```
>       defs = [ ep | ((c,h),ep) <- zip2 chs eps; c = ""]
>
>       gccases [] eps2 = ([],0,0,eps2)
>       gccases (((c,h),e):rchs) eps2 =
>         (bcase++chec++rhec,max [cstack,rstack],max [ch,rh],reps), otherwise
>         where
>           bcase = Case e:Get tmp:gnvs
>           nvs
>           = [],                                                    c= ""
>           = [defaultapply tmp pm (pd++[k]) | k<-[1..arity types c]], otherwise
>           gnvs = map Get nvs
>           (chec,cstack,ch,ceps) =
>             genpattcode hlist types funcno pm pd h eps2 (vs++filter (~=tmp) nvs)
>           (rhec,rstack,rh,reps) = gccases rchs ceps
>           tmp = "_tmp"
```

genpattcode fails if the HESF code includes the Full Evaluation instruction
(it is never generated by genhesf).

```
>   genpattcode hlist types funcno pm pd (Full s c) eps vs =
>     error "Full Evaluation not supported"
```

geneqncode generates gcode for all the equations, given a list of triples
(entry,expr,vmap) for each of the equations.

```
>   geneqncode :: [ids]->[(ids,trees,maps ids ids)]->[ids]->
>                                          ([gcode],num,[num],[ids])
>   geneqncode fvs [] eps = ([],0,[],eps)
>   geneqncode fvs ((ep,eq,vmap):eqns) eps =
>     (Case ep:code++rcode,max [stack,rstack],hs:hlist,eps2)
>     where
>       (code,hs,stack,eps1) = geneqn eq vmap fvs eps
>       (rcode,rstack,hlist,eps2) = geneqncode fvs eqns eps1
```

geneqn generates gcode for one equation. The first instruction checks that
there is sufficient heap for this block (the amount is calculated when the
remaining code is generated). Then a list of suspensions is generated and the
values placed on the stack in reverse order. The block is finished by a
transfer instruction, and then blocks to handle each of the suspensions are
generated.

```
>   geneqn :: trees->maps ids ids->[ids]->[ids]->([gcode],num,num,[ids])
>   geneqn (Tree f sts) vmap fvs eps =
>     (suspensions++jump:closures,hsize,stack,eps2)
>     where
>       (suspensions,hsize,stack) = gensusp vmap fvs (reverse zse)
>       (closures,eps2) = genclos vmap fvs zse (drop (#sts) eps)
>       jump
>       = Enter (apply vmap f), in f (dom vmap)
>       = Jump f,               otherwise
>       zse = zip2 sts eps
```

gensusp generates suspensions for arguments to a function call. If the
argument is simply a variable, then this is pushed onto the stack, otherwise
a suspension node is created, filled in with the free variables of the
sub-expression (at least one is required, since each node must be at least
two words long), and then the address of the suspension node is stored on the
stack.

```
>   gensusp :: maps ids ids->[ids]->[(trees,ids)]->([gcode],num,num)
>   gensusp vmap fvs [] = ([],0,0)
>   gensusp vmap fvs ((s,e):ses)
```

```
>   = (push:rclos,hsize,stack+1),       la = [] & in f (dom vmap)
>   = (fill++rclos,hsize+huse,stack+1), otherwise
>     where
>       (Tree f la) = s
>       (rclos,hsize,stack) = gensusp vmap fvs ses
>       push = Push (apply vmap f)
>       fill =
>         New vi:Push vi:Fill e:[Fill (defaultapply e vmap v) | v<-freevars]
>       vi = hd fvs
>       huse = #freevars+1
>       freevars
>       = fvars, fvars ~= []
>       = [e],   otherwise
>         where fvars = mkset (free vmap s)
```

genclos generates code blocks to be used when evaluating a suspension on the stack. This consists of an opening label, the recovery of any free variables (it is not necessary to recover a padding variable), and then the code for the subexpression is generated by recursively calling the function to generate code for an expression above.

```
>   genclos :: maps ids ids->[ids]->[(trees,ids)]->[ids]->([gcode],[ids])
>   genclos vmap fvs [] eps = ([],eps)
>   genclos vmap fvs ((s,e):ses) eps
>   = genclos vmap fvs ses eps,           la = [] & in f (dom vmap)
>   = (label++Stack stack:Heap hs:gets++code++rcode,eps2), otherwise
>     where
>       (Tree f la) = s
>       label = [mkdata (#freevars+1),Label e]
>       gets = Get e:[Get (apply vmap v) | v<-freevars]
>       (code,hs,stack,eps1) = geneqn s vmap fvs eps
>       (rcode,eps2) = genclos vmap fvs ses eps1
>       freevars = mkset (free vmap s)
```

xprod calculates the cross product of a list of lists as follows: if the list is empty, then the result is also empty. If there is one element, v, in the list, then the result is a list of singleton lists, one singleton list for each element in the inner list v. If there is more than one element in the list, v being the first element (a list) and vs being the list of remaining lists, then a new list of lists is produced by combining each element in the first list v, with each list in the cross product of remaining lists vs. The result will be a list of lists, of length equal to the product of the lengths of the initial lists, while each result inner list will have length equal to the length of the original outer list.

```
>   xprod :: [[ids]]->[[ids]]
>   xprod [] = []
>   xprod [v] = [ [vi] | vi <- v]
>   xprod (v:vs) = [ vi:vj | vi<-v ; vj <- xprod vs]
```

free finds all the free variables in a tree expression. The set of all free variables is given by (dom vmap)

```
>   free :: maps ids ids->trees->[ids]
>   free vmap (Tree f sts)
>   = (f:rfree), in f (dom vmap)
>   = rfree,     otherwise
>     where rfree = concat (map (free vmap) sts)
```

bpmap builds a mapping from path descriptors to gcode variables, permitting a consistent naming convention throughout the G-Code.

```
>   bpdmap :: patts->num->maps posn ids
>   bpdmap ps nargs =
>     listtomap (lp1++lp2)
>     where
>       lp1 = [(p,mkv (hd p)) | p<-lv; #p = 1]
>       lp2 = zip2 [reverse p | p<-lv; #p > 1] [mkv v | v <- [nargs+1..]]
>       lv = mkset (bpmap [] ps)
>       bpmap m [] = m
>       bpmap m (p:rps) =
>         bpmap (bp1 m [] 1 p) rps
>         where
>           bp1 m pd k [] = m
>           bp1 m pd k ((Tree v sts):ts) =
>             bp1 (bp1 ((k:pd):m) (k:pd) 1 sts) pd (k+1) ts
```

bvmap builds a mapping from variable names in the patterns to variable names in G-Code for one equation. This requires the use of the path mapping as defined above.

```
>   bvmap :: typedicts->maps posn ids->[trees]->maps ids ids
>   bvmap types pdmap p =
>     listtomap [(v,apply pdmap (reverse pd)) | (v,pd) <- bmap [] [] 1 p]
>     where
>       bmap m pd k [] = m
>       bmap m pd k (t:ts) =
>         bmap (bm1 m (k:pd) t) pd (k+1) ts
>       bm1 m pd (Tree v sts)
>       = bmap m pd 1 sts,          iscons types v
>       = bmap ((v,pd):m) pd 1 sts, otherwise
```

mke builds an entry point given a function number and a series number.
mkv builds a G-Code variable given a series number

```
>   mke :: num->num->[char]
>   mke f e = "f"++shownum f++"e"++shownum e
>   mkv :: num->[char]
>   mkv v = "v"++shownum v
>   mkdata k = Data k (2^k-2)
```

The file `presel.lit.m` contains possible definitions for the functions `pre` and `sel` used in the pattern matching algorithm.

```
>   %include <local/maps>
>   %include <local/oldsets>
>   %include "fl.lit"
>   %include "types.lit"
>   %include "tables.lit"
```

pre performs theoretical preprocessing on the patterns. None is actually done.
sel selects an appropriate equation from the overlaps. This one emulates the Miranda ‘‘choose the textually first’’ rule.

```
>   pre :: typedicts->patts->patts
>   pre types ps = ps

>   sel :: tables->tables
>   sel (Eqns ks) = Eqns (minl ks)
>   sel (Table vs) = Table [ (c,sel t) | (c,t) <- vs]

>   minl [] = []
>   minl l = [min l]
```

The file `support.lit.m` contains miscellaneous functions used by one or more of the other files.

```
uniq returns the element in a singleton list

>   uniq :: [*]->*
>   uniq [x] = x

odds returns all the odd elements in a list

>   odds [] = []
>   odds (x:l) = x:evens l

evens returns all the even elements in a list

>   evens [] = []
>   evens (x:l) = odds l

indent, applied to a number and a list of lines, produces a new list of lines,
each of which is indented by the amount specified by the number.

>   indent k ls = [rep k ' '++l | l<-ls]

concsp concatenates a list of strings, inserting a space between each pair of
strings.

>   concsp [] = ""
>   concsp [x] = x
>   concsp (x:xs) = x++" "++concsp xs

prstar is used in the display of tables and HESF code to print a star in place
of the empty string (which is used internally).

>   prstar "" = "*"
>   prstar s = s
```

Appendix D

Extended Type Algorithms

This appendix contains code for the algorithms described in Chapter 5: the non-polymorphic unique representation algorithm (see Section 5.5, but without the type synonym notation); the infinite cycle checker (see Section 5.6); and the algorithm for generating restrictions to be applied to functions (see Section 5.7). These are compatible with the parser and code generation algorithms presented in Appendices B and C. However, an additional example top-level Miranda script is provided in the file `example.lit.m`. The test data is still to be found in the file `tdata.lit.m`

```
>   %include <local/oldsets>
>   %include <local/maps>

>   %include "fl.lit"
>   %include "ptypes.lit"
>   %include "types.lit"

>   %include "gcode.lit"
>   %include "hesf.lit"
>   %include "tables.lit"

>   %include "parse.lit"
>   %include "pm.lit"
>   %include "presel.lit"
>   %include "restrict.lit"

>   %include "btypes.lit"
>   %include "infcycchk.lit"

>   %include "tdata.lit.m"

This example generates the test cases for mink: the two tables, generated and
selected, the HESF code, G-Code, and the new set of equations (with
restrictions).

>   mink_example =
>     showtable tab1 ++
>     showtable tab2 ++
>     showhesf hec ++
>     gcodepr (compile pre sel [f] td) ++
>     showeqs (gendisjeq td tab1 tab2 f)
>     where
>       (fm,td) = parse prog
```

```
>       f = apply fm "mink"
>       ps = [ p | (Tree "=" [Tree f p,e])<-f]
>       tab1 = buildtab td [] [1..#ps] ps
>       tab2 = sel tab1
>       (hec,res) = comppatt pre sel td ps

showeqs is a functions to display a list of equations with restrictions.

>   showeqs :: [(exprs,exprs,exprs)]->[char]
>   showeqs eqs =
>     lay [s l++" = "++s r++" { "++s q++" }" | (l,r,q)<-eqs]
>     where s = showexprs

showexprs displays an expression stored as a tree

>   showexprs :: exprs->[char]
>   showexprs (Tree f []) = f
>   showexprs (Tree f as) = "("++f++concat [" "++showexprs a | a<-as]++")"
```

The file `btypes.lit.m` contains the unique representation (base type) algorithm.

```
>   %include <local/oldsets>
>   %include <local/maps>

>   %include "fl.lit"
>   %include "types.lit"

>   %include "tables.lit"

basetypes is a list of booleans representing a base type, True indicating
positive intersection, and False indicating negative intersection.

ntypes is included solely for the ease of the rest of the algorithm. Each of
the original types is given a new representation. The constructor name is
replaced by its corresponding (name,arity) pair, and each of the type
variables is replaced by a number pointing up the position in the original
list of the type corresponding to that variable.

>   basetypes ::= Base [bool]
>   ntypes ::= NT ids num [num]

fbasetypes finds all the base types for a given set of original type
definitions. Also provided are mappings of constructors and variables used to
perform the transformations described above. The recursive base types algorithm
is applied to the results of the constant base types algorithm.

>   fbasetypes :: typedicts->sets basetypes
>   fbasetypes data =
>     baser basec
>     where

The constructors are split into two lists, constant and non-constant consK
and consC.

>       consK = [ (n,Cons k) | (n,Cons k) <- maptolist data; k = 0]
>       consC = [ (n,Cons k) | (n,Cons k) <- maptolist data; k ~= 0]

The transformation of types described above is carried out and the result is
placed in nt. srch is used to find the correct number to put in the type.
tns is used to hold the correspondence of number and type name.

>       tns = zip2 [0..] [n | (n,PattDef ps) <- maptolist data]
>       nt = [ [ NT c (arity data c) [ srch tns t | t<-ts ] |
```

```
>              (Simple c ts)<-ps] | (n,PattDef ps) <- maptolist data]
```

srch runs through the list of types, returning the position of a given typename in the list.

```
>       srch ((k,n):kns) v
>       = k,          n = v
>       = srch kns v, otherwise
```

basec is the set of base types which can be determined by examining the membership of just constant values in the types.

```
>       basec =
>         listtoset [Base [member t (NT c k []) | t<-nt] | (c,Cons k) <- consK]
```

baser applies the recursive base types algorithm to an initial set of base types. The algorithm has two cases : no new base types could be generated, in which case the initial set of base types is returned; and some new base types were generated, in which case this set is used as the argument to a recursive call to baser.

```
>       baser base
>       = base,          setsize base = setsize newbase
>       = baser newbase, otherwise
>         where
```

lbase is a list of the initial base types, needed for Miranda, which cannot process a set sequentially.

```
>           lbase = settolist base
```

newbase is the set of base types formed by merging the newly created base types with those which already existed.

```
>           newbase = union base extras
```

extras is the set of newly created base types. It is calculated as follows: for each non-constant constructor, build up all the patterns of (constructor,[basetypes]) for which #[basetypes] = arity(constructor). Then, for each type out of the list, determine whether there is an intersection with each of these patterns. The result is the set of lists produced by doing this. The presence of an intersection is determined by applying the twiddle function to the pattern, together with each element of the type. If any of these twiddle applications returns True, there is an intersection.

```
>           extras =
>             listtoset [ Base [ or [ twiddle (c,v) ti | ti <- t] | t <- nt] |
>                         (c,Cons k) <- consC; v <- prod k lbase]
```

twiddle tests whether a pattern represented as (constructor,[basetypes]) intersects with a pattern represented as (constructor,[type numbers]). This is true if for every corresponding argument pair, the appropriate (determined from the type number) element of the base type is True.

```
>       twiddle (c,v) (NT c k v1) =
>         and [ b!t | (Base b,t) <- zip2 v v1]
>       twiddle x y = False
```

prod creates a k-dimensional cross product of the list of base types as follows. If k=1, then the result is just a list of singleton lists for each element of the base types. If k>1, then the result is a list which is the two dimensional cross product of each element of the list of base types with each

```
element of the k-1-dimensional cross product of the list of base types.

>       prod 1 base = [[b] | b <- base]
>       prod k base =
>         [ b:rbs | b<-base; rbs<-prod (k-1) base]
```

The file `infcycchk.lit.m` contains the infinite value cycle checker.

```
>   %include <local/oldsets>
>   %include <local/maps>
>   %include "fl.lit"
>   %include "types.lit"

transitions is a type used to list all the edges in the graph of types. It
contains an initial type, a constructor/argument position pair, and a
destination type.
goals is a type used to represent goals for the search algorithm. It contains
the initial position in each cycle, the next constructor/argument pair to be
tried, the types that will result from this transition, and a boolean
indicating whether an imbalance of bottoms has yet been found.

>   transitions ::= Trans ids ids num ids
>   goals ::= Goal ids ids ids num ids ids bool

testcycle is the top level function, applied to a type dictionary. It
constructs a list of transitions from the type definitions in the dictionary,
and then calls tcycle to check the cycles. The list of types containing bottom
is also acquired by testing each type to see if the constructor Bottom is
included.

>   testcycle :: typedicts->bool
>   testcycle g =
>     tcycle trs bs
>     where
>       lg = maptolist g
>       trs = ftrans [(t,PattDef ps) | (t,PattDef ps)<- lg]
>       bs = [ t | (t,PattDef ps) <- lg; member ps (Simple "Bottom" [])]

ftrans finds all transitions existing in the graph of types. This is the form
which the cycle testing algorithm uses to form new goals from old.  The
transitions are found by discarding irrelevant transitions from the complete
list of all transitions.

>   ftrans :: [(ids,typedata)]->[transitions]
>   ftrans g =
>     discard trans
>     where

The list of all transitions is found by, for each type, looking at the list of
patterns, and producing one transition from the original type to the argument
type annotated with the constructor and argument position.

>       trans = concat (map falltrans g)
>       falltrans (t,PattDef ps) =
>         [ Trans t c a (args!a) | (Simple c args) <- ps; a <- [0..#args-1]]

The discarding procedure works by repeatedly discarding irrelevant types
until two iterations of the loop produce the same results. If the original
list is finite, this must terminate, since the loop may only continue while
the length of the list is decreasing, and it may only decrease by a finite
amount.

>       discard trs
```

```
>       = trs2,         trs2 = trs
>       = discard trs2, otherwise
>         where
```

Two sorts of entries are discarded: Any entry which proposes a transition into a non-existent type, i.e. one that has an 'nt' field which is not a 't' field for any other transition; and any entry which is inaccessible, i.e. one that has a 't' field which is not an 'nt' field for any other transition.

```
>           ts = mkset [ t |  (Trans t c a nt) <- trs ]
>           trs1 = [ Trans t c a nt | (Trans t c a nt) <- trs ; member ts nt]

>           rts = mkset [ nt | (Trans t c a nt) <- trs ]
>           trs2 = [ Trans t c a nt | (Trans t c a nt) <- trs1 ; member rts t]
```

tcycle does the work of attempting to distinguish cycles. It takes a list of transitions and a list of types containing bottom, establishes a list of possible cycles, and then attempts to investigate these as best it may, by repeatedly producing new goals.

```
>   tcycle :: [transitions]->[ids]->bool
>   tcycle trs bs =
>     test initgoals []
>     where
```

The initial list of goals is produced by taking a cross product of all transitions for which the constructor and argument match, and for which at least one of the initial types contains bottom, and constructing a goal consisting of the source types for the transitions, the constructor and argument position used in both transitions, and the destination types for the transition, and finally the boolean representing whether or not a bottom mismatch occurred in the source types.

```
>       initgoals =
>         [ Goal t1 t2 c a nt1 nt2 (isb t1 ~= isb t2) |
>           (Trans t1 c a nt1) <- trs; (Trans t2 c2 a2 nt2) <- trs;
>           c2 = c; a2 = a; isb t1 \/ isb t2; t1 ~= t2]
```

Test does the loop work in deciding whether or not a goal may cause a problem. It takes two lists of goals, the first is the goals still to be tried, and the second is the goals that have been tried. It returns true if no problem cycles may be found, and false otherwise.

If there are no goals remaining to be tested, this can only be because all suggested goals have been tried unsuccessfully. Therefore the system contains no problem cycles. Otherwise, the first goal is removed from the list of goals still to be tried, and it is tested to see whether it directly represents a problem cycle (if the current state is the same as the original state, and if a bottom mismatch has been seen). If so, false is immediately returned. Otherwise, if the goal has already been seen it is discarded and the remainder are tried. Otherwise, if the cycles have been completed, and bottom has not been seen in both cycles, then the goal is added to the seen list, and the remainder are tried. Finally, in any other case, appropriate new goals are generated (possibly none) and these are tried in turn, with the current goal being added to the list of those seen & tried.

```
>       test [] seen = True
>       test (goal:goals) seen
>       = False,                              bl = True & t1 = nt1 & t2 = nt2
>       = test goals seen,                    member seen goal
>       = test goals (goal:seen),             t1 = nt1 & t2 = nt2
>       = test (newgoals++goals) (goal:seen), otherwise
```

```
>         where
```

```
The list of goals to be added in the default case is generated as follows: all
the transitions are found which have a source type equal to that of the current
state for each cycle (two lists of transitions), for each of these transitions,
the constructors and the argument positions must match. The new goal consists
of the same final destination types, the new matching constructor and argument
position information, and the destination for each of the transitions, together
with whether or not a bottom mismatch has been seen yet.
```

```
>           (Goal t1 t2 c a nt1 nt2 bl) = goal
>           newgoals =
>             [ Goal t1 t2 ac aa ant bnt (bl \/  (isb ant ~= isb bnt)) |
>               (Trans at ac aa ant) <- trs; nt1 = at;
>               (Trans bt bc ba bnt) <- trs; nt2 = bt;
>               ac = bc; aa = ba]
```

```
isb is a function to determine whether or not bottom is in a particular type.
```

```
>       isb = member bs
```

The file `restrict.lit.m` generates disjoint equations from a function definition.

```
>   %include <local/oldsets>
>   %include <local/maps>

>   %include "fl.lit"
>   %include "types.lit"

>   %include "tables.lit"

>   %include "support.lit"
```

```
gendisjeq takes, along with a type dictionary, the generated and selected
tables for a function (as constructed by buildtab), and the function definition
(as a list of cases) and produces a list of triples, which consist of
(left,right,restr) for the function. These are the ‘‘true equations’’
associated with the definition of the functions.

genrestr assists in the construction of these equations by first finding the
‘‘exceptions’’ -- the places where there is an equation entry in the generated
table which is not in the selected table -- and then converting these into
expressions using the ~match operator, and combining such expressions using a
conjunction of disjunctions.

excepts is the list of exceptions in table2 compared to table1, arranged by
equations. The result of getexc is offset from one, not zero, so must be
adjusted by dropping the first element, and may lack elements at the end for
equations with no restrictions, so an infinite amount of ‘‘no restrictions’’
are added to the end of the exceptions.

restr contains the restrictions as a list of lists of expressions involving
~match. The only remaining task (performed by mkrestr) is to convert this into
a single expression represented as a conjunction of disjunctions.
```

```
>   gendisjeq :: typedicts->tables->tables->functions->[(exprs,exprs,exprs)]
>   gendisjeq types tab1 tab2 func =
>     [(l,r,re) | (Tree "=" [l,r],re) <- zip2 func restr]
>     where
>       ps = [ p | (Tree "=" [Tree f p,e]) <- func]
>       restr = genrestr types tab1 tab2 ps

>   genrestr :: typedicts->tables->tables->[[exprs]]->[exprs]
```

```
>   genrestr types tab1 tab2 ps =
>     map mkrestr restr
>     where
>       restr =
>         [[concat[finddiff types p1 c1 | (p1,c1)<-zip2 pt c] |
>                   (Tree "@" c) <- ct] | (pt,ct) <- zip2 ps excepts]
>       excepts = tl (getexc types tab1 tab2 ++ repeat [])
```

getexc finds the exception values by comparing the tables tab1 and tab2 to see where equations have been eliminated in tab2 from tab1. It is guaranteed that both tables have the same structure (tested by comppatt), and so the only difference is when a table is of the form (Eqn ks), the ks is tab2 is a sublist of that in tab1 (though not necessarily a proper sublist). These differences are found by findsub, converts the nested contexts into tree-shaped values, and groups them by equation number.

```
>   getexc :: typedicts->tables->tables->[[trees]]
>   getexc types tab1 tab2 =
>     groupr [(es,cvtree types vs) | (es,vs) <- findsub [] tab1 tab2; vs ~= []]
```

cvtree converts a value represented by a history of contexts (as generated by findsub) into a more normal tree-structured value. It is assumed that any contexts passed in are non-empty. The context consists of a list of lists of constructors -- each constructor in each list is applied to the correct arity number of arguments from the next list. The job (performed by cv1) is to find the tree belonging to the first constructor of the first list, and then to repeat recursively, deleting the constructed elements as it goes. The eventual result will be a list of trees (one element for each constructor in the first list) and a list of empty lists (the remaining elements).

```
>   cvtree :: typedicts->[[ids]]->trees
>   cvtree types l =
>     Tree "@" trees
>     where
>       (trees,reml) = cv1 (#(hd l)) l
>       cv1 0 l = ([],l)
>       cv1 k ((x:rxs):rys) =
>         (Tree x v:y,z)
>         where
>           (v,w) = cv1 (arity types x) rys
>           (y,z) = cv1 (k-1) (rxs:w)
```

groupr groups the list of (equations,restriction) pairs to form a list (per equation) of lists (per restriction) of expressions (restrictions). It does this by moving through the list of pairs, and for each one placing the restriction on the front of a list of restrictions for each of the equations for which the restriction applies.

```
>   groupr :: [([num],*)]->[[*]]
>   groupr [] = []
>   groupr ((es,r):rers) =
>     gr1 (groupr rers) es
>     where
>       gr1 sr [] = sr
>       gr1 sr (eq:reqs) =
>         gr1 (place sr eq) reqs
>       place l k
>       = take k l ++ [r:(l!k)] ++ drop (k+1) l, k < #l
>       = l ++ (rep (k-#l) []) ++ [[r]],       otherwise
```

findsub recursively finds elements of tab1 which have been eliminated from tab2. As it finds them, it records which equations are affected, and the

```
value concerned. This is deduced by looking at the list of coverages formed
by each recursive step of the algorithm. Since the tables are guaranteed
structurally identical, c1 = c2 in the recursive case, and the two are
interchangeable.

>   findsub :: [[ids]]->tables->tables->[([num],[[ids]])]
>   findsub h (Eqns ks1) (Eqns ks2)
>   = [],                     ks1 = ks2
>   = [(ks1--ks2,reverse h)], otherwise
>   findsub h (Table vs1) (Table vs2) =
>     concat [findsub (c1:h) v1 v2 | ((c1,v1),(c2,v2)) <- zip2 vs1 vs2]

finddiff examines a pattern together with an ‘‘exception value’’ to see how
they compare: it is looking for instances where there is a variable in the
pattern which matches part of the exception value: the restriction is that this
variable is not permitted to match this value.

>   finddiff :: typedicts->trees->trees->[trees]
>   finddiff types (Tree pc pas) (Tree cc cas)
>   = [Tree "~match" [Tree pc pas, Tree cc cas]],            ~ iscons types pc
>   = concat [finddiff types pt ct | (pt,ct) <- zip2 pas cas], pc = cc
>   = error "can't happen in finddiff",                       otherwise

mkrestr forms the conjunction of disjunctions required by genrestr.

>   mkrestr :: [[exprs]]->exprs
>   mkrestr res =
>     mkconj res
>     where
>       mkconj [] = Tree "True" []
>       mkconj l = foldl1 conj (map mkdisj l)
>       mkdisj [] = Tree "True" []
>       mkdisj l = foldl1 disj l
>       conj e1 e2 = Tree "&" [e1,e2]
>       disj e1 e2 = Tree "|" [e1,e2]
```

Appendix E

Interpreter for G-Code

This Appendix gives source code in the C programming language for a small interpreter (under 400 lines, including comments) for the G Code presented in Chapter 3. This interpreter has been successfully compiled on a Sun 3/50 workstation, and on an IBM PC compatible, and has executed a number of programs, including the test program given in Appendix B.

```
/*****************************************************************************
A FUNCTIONAL LANGUAGE INTERPRETER.  This interpreter loads and runs programs
written in G-Machine Assembly Language. It is intended to demonstrate the
'spineless tagless g-machine' approach to functional language compilation.
It is written in C, but uses simple integer techniques only, to avoid type
problems and to allow it to be converted easily into other languages.
   Everything is held in a single integer array 'mem'.  The integers are used
both as raw data and as 'pointers', i.e. indexes into the array itself.  The
code is loaded into the beginning of the array, using a symbol table built at
the end of the array.  During execution, the heap works upwards from the code,
and the stack works downwards from the symbol table.
*****************************************************************************/

#include <stdio.h>
int mem[10000];
int startmem=0, code, heap, heaplim, stacklim, stack, syms, endmem=10000;
int pc, node, tag, acc;

enum opcodes {
   FUNCTION, LABEL, CONTINUE, CASE, DATA, HEAP, STACK, ARGS,
   PUSH, POP, PUT, GET, SAVENODE, SETNODE, NEW, FILL,
   LOAD, STORE, ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPARE,
   READ, WRITE, JUMP, ENTER, RETURN, SWITCH, TRY, UNTRY, STOP
};

char *opnames[] = {
   "FUNCTION","LABEL","CONTINUE","CASE","data","heap","stack","args",
   "push","pop","put","get","savenode","setnode","new","fill",
   "load","store","add","subtract","multiply","divide","compare",
   "read","write","jump","enter","return","switch","try","untry","stop"
};

main(argc,argv)
char **argv;
{
   FILE *program;
```

```
   if (argc != 2) { printf("Wrong no. of args\n"); exit(1); }
   program = fopen(argv[1],"r");
   if (program == NULL) { printf("Can't open program file\n"); exit(1); }
   load(program);
   run();
}

/*****************************************************************************
THE LOADER.  The 'load' procedure first initialises the symbol table by
entering the operation symbols, then it loads in a program file.  All symbols
in the program are assumed to be distinct.  The value of an operation symbol is
its opcode, of an entry point is its address in the code, and of an integer
constant is the corresponding integer.  Other symbols are assumed to be local
variables whose values are left undefined.
   Each operation is on one line, and has one argument.  Multiple arguments are
an abbreviation for repeating the operation on each argument in turn.  Blank
lines, and everything on a line after the '|' symbol, are comments.  The loader
imposes arbitrary restrictions on the sizes of various things.
*****************************************************************************/

load(program)
FILE *program;
{
   char name[50], line[80], s[6][20];
   int op, sym, i, n;

   syms = endmem;
   for (op=FUNCTION; op<=STOP; op++) {
      sym = lookup(opnames[op]);
      mem[sym] = op;
   }

   code = startmem;
   while (fgets(line,80,program) != NULL) {
      n = sscanf(line,"%s%s%s%s%s%s",s[0],s[1],s[2],s[3],s[4],s[5]);
      for (i=0; i<n; i++) if (strcmp(s[i],"|") == 0) n = i;
      if (n==0) continue;
      op = mem[lookup(s[0])];
      for(i=1; i<n; i++) {
         sym = lookup(s[i]);
         mem[code++] = op;
         mem[code++] = sym;
         if (op <= CASE) mem[sym] = code;
         if ((s[i][0] >= '0')&&(s[i][0] <= '9')) sscanf(s[i],"%d",&mem[sym]);
      }
   }
}

/*****************************************************************************
THE SYMBOL TABLE.  The 'lookup' procedure looks a name up in the symbol table,
and returns the corresponding entry, creating a new entry if necessary.  Each
entry in the table consists of a word holding the value, followed by the number
of characters in the name, followed by the name itself, one character per word.
*****************************************************************************/

lookup(name)
char *name;
{
   int sym, i, len;

   len = strlen(name);
   for (sym = syms; sym < endmem; sym = sym + 2 + mem[sym+1]) {
```

```
      if (mem[sym+1] != len) continue;
      for (i=0; i<len; i++) if (name[i] != mem[sym+2+i]) break;
      if (i==len) return sym;
   }
   syms = syms - len - 2;
   mem[syms+1] = len;
   for (i=0; i<len; i++) mem[syms+2+i] = name[i];
   return syms;
}

/*****************************************************************************
EXECUTION.  The 'run' procedure executes the code.  Each op has one argument
which is a pointer to the value word of a symbol table entry. This allows us to
have only one addressing mode.
*****************************************************************************/

run()
{
   int op, arg, val, i, steps;

   pc = mem[lookup("result")];
   stack = syms;   stacklim = stack - 500;
   heap = code;    heaplim = stacklim - 200;
   steps = 0;
   while (1) {
      steps++;
      op = mem[pc++];  arg = mem[pc++];  val = mem[arg];
      switch (op) {
case HEAP:    if (heap + val > heaplim) checkheap(val); break;
case STACK:   if (stack - val < stacklim) checkstack(val); break;
case ARGS:    for (i=0;i<val;i++) if (mem[stack+i]<code) checkargs(val); break;
case PUSH:    mem[--stack] = val; break;
case POP:     mem[arg] = mem[stack++]; break;
case PUT:     mem[node++] = val; break;
case GET:     mem[arg] = mem[node++]; break;
case SAVENODE:mem[arg] = node; break;
case SETNODE: node = val; break;
case NEW:     mem[arg] = heap; break;
case FILL:    mem[heap++] = val; break;
case LOAD:    acc = val; break;
case STORE:   mem[arg] = acc; break;
case ADD:     acc = acc + val; break;
case SUBTRACT:acc = acc - val; break;
case MULTIPLY:acc = acc * val; break;
case DIVIDE:  acc = acc / val; break;
case COMPARE: tag = (acc < val) ? 0 : (acc == val) ? 1 : 2; break;
case READ:    mem[arg] = getchar(); tag = (mem[arg] == EOF) ? 0 : 1; break;
case WRITE:   printf("%c",val); break;
case JUMP:    pc = val; break;
case ENTER:   node = val; pc = mem[node]; break;
case RETURN:  tag = val; pc = mem[stack]; break;
case SWITCH:  if (tag==0) pc=val; else tag--; break;
case TRY:     tag-=val; break;
case UNTRY:   tag+=val+1; break;
case STOP:    exit(val);
default:      printf("Error : %d steps -> ",steps); printop(pc-2); exit(1);
      }
   }
}

/*****************************************************************************
CHECKING. These procedures check for sufficient heap space, stack space, or
```

```
function arguments. In the first two cases the collector is called, and in the
third, a partial application node is created.
   Heap checks can happen at FUNCTION, LABEL, and RETURN entry points, stack
checks at FUNCTIONs or LABELs, and argument checks at FUNCTIONs only.
*****************************************************************************/

checkheap(n)
{
   int entry;

   if (heap + n <= heaplim) return;
   entry = pc;
   while (mem[entry] > CASE) entry = entry - 2;
   if (mem[entry] != FUNCTION) mem[--stack] = node;
   collect();
   if (mem[entry] != FUNCTION) node = mem[stack++];
   if (heap + n > heaplim) { printf("No heap\n"); exit(1); }
}

checkstack(n)
{
   int entry;

   if (stack - n >= stacklim) return;
   entry = pc;
   while (mem[entry] > CASE) entry = entry - 2;
   if (mem[entry] != FUNCTION) mem[--stack] = node;
   collect();
   if (mem[entry] != FUNCTION) node = mem[stack++];
   if (stack - n < stacklim) { printf("No stack\n"); exit(1); }
}

char *applylabels[] = {"Apply0", "Apply1", "Apply2"};

checkargs(n)
{
   int i, fun;

   for (i=0; i<n; i++) if (mem[stack+i]<code) break;
   if (i == n) return;
   if (i > 2) {printf("Long apply\n"); exit(1);}
   fun = pc;
   while (mem[fun-2] != FUNCTION) fun = fun - 2;
   node = heap;
   mem[heap++] = mem[lookup(applylabels[i])];
   mem[heap++] = fun;
   while (i--) mem[heap++] = mem[stack++];
   tag = 0;  pc = mem[stack];
}

/*****************************************************************************
THE GARBAGE COLLECTOR.  This removes dead nodes from the heap, compacting the
remaining live ones. On entry to the garbage collector, the following
conditions must hold.

1) The heap must consist entirely of valid nodes, so that the collector can
   scan the heap correctly.  Thus if a node is overwritten with a shorter one,
   the left-over words must be filled with one-word nodes (e.g. Nil).

2) All current pointers to heap nodes must be on the stack.  The collector uses
   these pointers to find the live nodes, and then updates them when the nodes
   are moved.  This means the node register must be pushed on the stack if it
```

```
   is valid (LABEL & RETURN), and the stack frame must be intact (RETURN).

The first pass of the collector finds all the live nodes by following pointers.
At the same time, all the references found to a live node are threaded onto a
chain stored in place of the node's label.  The chain is terminated by the
original label, which can be distinguished because it is a pointer into the
code rather than the heap.  During the first pass, nodes which have been
visited but whose references have not been scanned are kept on the stack (using
the space reserved between the heap and stack limits).
   During the second pass, the heap is scanned, and all the references to a
live node are made to point to its new location -- where it will be after
compaction. The node itself is not moved (since the references in it may need
updating later in the pass).  A live node can be detected during the scan
because it has a reference chain in place of a label. Its new location can be
calculated from the space taken up by live nodes found before it.  Dead nodes
are marked with negative numbers in place of labels.
   During the third pass, the nodes are moved to their new locations.
******************************************************************************/

thread(ref)                /* Thread a reference to a node onto its chain.  */
{                          /* If the node hasn't been visited before,       */
   int node;               /* put it on the stack for later scanning.       */

   node = mem[ref];
   if (mem[node] < code) {
      if (stack <= heaplim) { printf("No GC stack\n"); exit(1); }
      mem[--stack] = node;
   }
   mem[ref] = mem[node];
   mem[node] = ref;
}

scan(node)                 /* Scan a node (or stack frame) to deal with its */
{                          /* references. Find its label, and from that its */
   int lab, size, layout;  /* size and layout (position of references).     */

   lab = mem[node];
   while (lab >= code) lab = mem[lab];
   size = mem[mem[lab-5]]; layout = mem[mem[lab-3]];
   while (layout != 0) {
      if (layout & 1) thread(node);
      node = node + 1;  layout = layout >> 1;
   }
   return size;
}

collect()                  /* Perform garbage collection and compaction.    */
{
   int oldstack, p, ref, lab, size, new;

printf("GC1"); fflush(stdout);
   oldstack = stack;                       /* Visit all live nodes, thread */
   for (p = oldstack; p < syms; ) {        /* all references onto chains.  */
      if (mem[p] < code) p = p + scan(p);
      else thread(p++);
   }
   while (stack < oldstack) scan(mem[stack++]);

printf("\b2"); fflush(stdout);
   new = code;
   for (p = code; p < heap; p = p + size) {
      lab = mem[p];
```

```
      if (lab < code) {                    /* If node is dead, mark a gap. */
         size = mem[mem[lab-5]];           /* This could be improved by    */
         mem[p] = -size;                   /* marking groups of dead nodes.*/
      } else {
         while(lab >= code) {              /* Find new locations of live   */
            ref = lab;                     /* nodes & update references.   */
            lab = mem[ref];
            mem[ref] = new;
         }
         mem[p] = lab;                     /* Restore original label.      */
         size = mem[mem[lab-5]];
         new = new + size;
      }
   }

printf("\b3"); fflush(stdout);
   new = code;                             /* Move all live nodes to their */
   for (p = code; p < heap; ) {            /* new locations.               */
      lab = mem[p];
      if (lab < 0) {
         p = p + (-lab);
      } else {
         size = mem[mem[lab-5]];
         while (size--) mem[new++] = mem[p++];
      }
   }
   heap = new;
printf("\b\b\b"); fflush(stdout);
}

/*****************************************************************************
DEBUGGING PROCEDURES.  Procedures for printing the symbol table, the
loaded code, a single operation, a single symbol or a heap node.
*****************************************************************************/

printtable()
{
   int i,sym;
   for (sym = syms; sym < endmem; sym += 2+mem[sym+1]) {
      for (i=0; i<mem[sym+1]; i++) fprintf(stderr,"%c",mem[sym+2+i]);
      fprintf(stderr," %d\n",mem[sym]);
   }
}

printcode()
{
   int i;
   for (i=0; i<code; i++) {
      if (i%10 == 0) fprintf(stderr,"\n%d: ",i);
      fprintf(stderr,"%d ",mem[i]);
   }
   fprintf(stderr,"finished\n\n");
}

printop(pc)
{
   int i,arg,opcode;

   fprintf(stderr,"%d: ",pc);
   opcode = mem[pc];
   if ((opcode >= 0)&&(opcode <= STOP)) fprintf(stderr,"%s ",opnames[opcode]);
   else fprintf(stderr,"%d ",opcode);
```

```
   arg = mem[pc+1];
   if ((arg >= syms)&&(arg <= endmem)) printsym(arg);
   else fprintf(stderr,"%d",arg);
   fprintf(stderr," %d",mem[arg]);
   fprintf(stderr,"\n");
}

printsym(sym)
{
   int i;

   for(i=0;i<mem[sym+1];i++)
      fprintf(stderr,"%c",mem[sym+2+i]);
}

printnode(node)
{
   int lab, size, layout, i;

   fprintf(stderr,"p%d: ",node);
   lab = mem[node];
   while (lab >= code) lab = mem[lab];
   printsym(mem[lab-1]);
   size = mem[mem[lab-5]]; layout = mem[mem[lab-3]];
   for (i=1; i<size; i++) {
      if (((layout>>i)&1)) fprintf(stderr," p"); else fprintf(stderr," ");
      fprintf(stderr,"%d",mem[node+i]);
   }
   fprintf(stderr,"\n");
}
```