FUNCTIONAL PROGRAMMING APPLICATIONS

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT AT CANTERBURY

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

By

Mark Longley

August 1991

Abstract

Functional programming languages have distinct advantages over imperative languages. These include ease of reasoning, formally or informally, about programs and the concise and elegant expression of complex algorithms.

We use several large programming tasks to investigate various aspects of the production of a complete system in a functional language. These include the overall development of an implementation, the implementation of the core algorithms and the implementation of the external interface. We do not attempt verifications of the implementations but instead adopt a specification style that aids informal reasoning about the corresponding implementation.

We employ the language Miranda[1] as our example functional programming language.

[1] Miranda is a Trademark of Research Software Ltd.

Acknowledgements

I would like to thank the following people for their willingness to discuss both their work and mine; Rafael Lins, Steve Hill, Gareth Howells, Sean Supeville, John Cupitt, Robert Duncan, Richard Jones and Sean Leviseur. Particular thanks are due to Paul Gardiner and J. R. Abrial for their help in my understanding of the 'B' system and its logic. Simon Thompson provided many useful insights into the applications of functional programming languages in the implementations of logics and other areas. David Turner provided much advice on the use of Miranda[2].

Most thanks are, of course, due to Allan Grimley; without his constant encouragement and advice this work would not exist in the present form.

This research was funded by BP EMRA 'Functional Programming In the Large'.

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

Dedication

Mum and Dad.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Imperative programming languages can be characterized by the fact that they have an implicit state that is modified by the statements of the language. Programs consist of sequences of statements that control these modifications to the state. Declarative programming languages dispense with this implicit state and programs are presented as sets of equations. This not only makes reasoning, both formal and informal, easier but also allows the concise and elegant expression of complex algorithms. Functional programming languages are based on function definitions and their applications. This differs from the logic programming languages which are based on relations.

We shall not discuss in detail the differences between functional programming languages and other language classes. We shall simply describe those properties of functional languages which are of interest. For a discussion of functional programming

languages and their advantages see Hughes [Hug89] and Hudak [Hud89].

Programs in a functional programming language consist of a sequence of definitions binding names to values. In general these values will be functions from some domain of argument values to some range of result values. We do not 'run a program' as we would with an imperative language but instead evaluate some expression defined in terms of these functions.

Perhaps the most important property of functional programming languages is that they are *referentially transparent*[1]. This means that the value of an expression is determined only by the values of its sub-expressions and its evaluation can only affect those expressions of which it is a sub-expression. This differs from imperative languages, where expressions may reference the state which may be altered in an arbitrary fashion by assignment statements. This property greatly simplifies reasoning about the values of expressions and implies that the proofs of verification theorems will also be greatly simplified. Such verifications will, however, not be easy and a great deal of detailed work would still need to be done. Even so, this property also means that informal reasoning about a program during its development is greatly simplified, thus reducing errors and oversights. Informal verification of functional programs is also easier, and far more plausible arguments can be presented than for imperative languages.

[1]It can be argued that to be truly referentially transparent a functional programming language must have a non-strict semantics.

We employ the language Miranda[2] as our example functional programming language. Miranda has a number of advantages as an example language:

- It has a small elegant syntax which allows the clear expression of function definitions and their applications.
- It is a polymorphic strongly typed language which allows the secure definition of generic functions.
- It is a higher order language which allows the expression of commonly used control structures and data structure manipulations as higher order functions.
- It has both algebraic and abstract types which allow the creation of new data structures and secure types.
- It provides pattern matching for case analysis.
- It has a non-strict semantics which allows the manipulation of potentially infinite objects, allowing elegant solutions to many problems.
- It provides for separate compilation of modules which allows easy system development.

We shall not provide a detailed description of the language Miranda; for a description of Miranda one can refer to [Tur81, Tur85, Tur79, Hil89].

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

In this thesis we take three large programming tasks and use them to investigate various issues that arise when implementing a complete system in a functional programming language.

Experience has shown that a great many algorithms can be expressed as functional programs in such a way as to produce concise programs which can be easily understood [Fle90, dV88]. We examine the economy and elegance of the expression of the functional solutions to these three tasks and compare the advantages obtained for the different sections of the resulting programs. The process by which the systems were developed is also noted and various conclusions about the advantages, and disadvantages, of the functional programming approach are discussed.

We shall not, in general, concern ourselves with the run-time efficiency of our implementations. This is partly because increasingly efficient implementations of functional languages were being developed while this work was underway. More importantly the efficiency of the Miranda implementation, while not up to the state-of-the-art, was itself being improved. Thus it was not possible to obtain consistent performance figures from Miranda and even if they had been available they would not have properly represented the best results that could be obtained from a functional language.

We will not attempt to provide formal verifications of the correctness of our implementations. This is because we would require extensive machine support for the

verification of our programs and this is not presently available.

This work is intended as an investigation of some of the practical details of implementing large systems in a functional language. Unlike some previous work [Jon85, Jon86b, WF89, Bir80, Bir84b] which examined the development of a functional programming style distinct from imperative solutions we instead look at more general issues. We do not attempt to make our entire implementations exemplars of functional programming style but instead use them to support investigation of particular aspects of the overall implementation task. Thus certain aspects of some of our implementations could be described as skeuomorphic in that we simply required a working implementation to support that part that was our main concern. For similar reasons we were sometimes forced to adopt inelegant implementations to cope with bugs and inefficiencies in the pre-release versions of Miranda on which some of the implementations were developed. Larger systems have been implemented in functional programming languages, such as the lazy ML compiler [AJ89]. The utility of these implementations depends on the recent increasingly efficient implementations of functional languages. The description of the lazy ML compiler is mostly concerned with the G machine and its optimizations but some remarks about the implementation process are made. Two of the issues raised, source code related type error messages and run-time tracing, are addressed by two of the applications we investigate.

Our investigation leads us to the following conclusions:

- While quite large components of a system can benefit greatly from the expressive power of functional languages there are some components that do not. In particular, those parts of the system which involve multi-way case analysis, like lexical analysers, or have have an essentially imperative nature, like an interactive user interface, are only slightly clearer than in an imperative language.

- The strong type system and complex evaluation path can make development of an implementation very difficult.

  The first means that as an implementation is being developed the types of the data structures may alter and this can result in large sections of the program becoming badly typed. This is a particular problem if extensive pattern matching over algebraic types is used and the types of the constructors are altered, usually by the addition of extra arguments. Careful use of the abstract type mechanism can help with this problem but one loses the clear expression of the structures being manipulated.

  The second means that when evaluation fails to terminate, terminates with an error or returns an unexpected result it can be difficult to track down the source of the problem. Careful modular development and use of higher order functions can help with this problem. However, as it usually results from some basic error in the implementation strategy the inability to track intermediate values means that the cause of the error cannot be localized.

Chapter 2

Specification and Verification

While we will not be attempting formal verifications of our implementations we will still require some sort of specification of what we are trying to implement. In the following we shall first discuss the problems associated with the specification and verification of functional programs and then the informal approach we shall adopt.

2.1 Specification

Specification languages such as Z [Spi85] VDM [Jon86a] or Clear [BG80] provide a formal language for the specification of programs. They also provide a formal framework for reasoning about these specifications. The initial specification will, in general, be non-algorithmic to avoid implementation bias. This means that the initial specifications do not provide a method for producing the required result. Thus the

initial specification would have to be transformed through a number of stages towards a description of a method which could be translated into Miranda [Mor88, BD77, Bir84a, BH87]. These transformations are often complex and difficult to justify and only apparent if we know the algorithm we wish to employ. Recent work has provided the possibility of calculating the next stage from the current one [Bir89, MG90] but the process is still complex. The advantages of the transformational approach are that each step can be made small enough that its proof is reasonably simple.

As an example, we consider a specification in the style of Z [Spi85] of the unification of two terms. We do not claim that this is a totally correct Z specification or even that it is expressed in the best possible way. We merely intend to show how far the initial specification can be from the final Miranda implementation.

The unification algorithm is of central importance to the typechecker of §3.7 and is described in more detail in §3.4.

We are given two sets, *Op* of operators and *Var* of variables. We need know nothing more about these sets. We can then form expressions using the functions *VE* and *OE* as follows:

$$\begin{aligned} Expr \;\hat{=}\; & VE \;<<\; Var \;>>| \\ & OE \;<<\; Op \;\times\; seq[Expr] \;>> \end{aligned}$$

Thus an expression can be either a variable or a pairing of an operator with a sequence of sub-expressions.

Now we define a variable substitution: this is a mapping from variables to expressions. This mapping won't define a substitution for all variables and will only allow a single substitution for a variable. We also do not allow a substitution of a variable for itself. The following definition captures the required properties:

$$Vsubst \triangleq \{vs : Var \pfun Expr \mid vs \cap VE = \{\}\}$$

Thus *Vsubst* is the set of all partial functions from variables to expressions that do not take any variable to the expression consisting solely of that variable. Given such a substitution on variables we can easily extend it to a substitution on expressions. This is done by applying the variable substitution to the variables in the expression. We therefore define the following function:

$$\begin{array}{|l} e : Vsubst \to (Expr \to Expr) \\ \hline \forall vs : Vsubst \bullet \\ (e\ vs) \circ VE = VE \oplus vs \land \\ \forall op : Op; s : seq[Expr] \bullet e\ vs\ (OE\ (op, s)) = OE\ (op, (e\ vs) \circ s) \end{array}$$

Notice that e is a total function that, when applied to an element of *Vsubst*, gives us a total function from expressions to expressions. If the expression is simply a variable we apply the variable substitution if it is in its domain otherwise we leave the variable alone. For the more complex case we basically apply our expression substitution to the sub-expressions and then rebuild the new expression with the same operator. The above specifies exactly those total functions from expressions to expressions having these properties. To unify two expressions we must find a variable substitution which

makes the two expressions equal. That is, applying the same substitution to both expressions produces the same expression. Not all pairs of expressions will be unifiable in this fashion. We also require the most general unifier of the two expressions. That is, for any other unifying substitution the resulting expression must be an instance of that obtained from the selected unifier. We first define the set of all expression substitutions:

$$\begin{array}{|l} Esubst : \mathbb{P}(Expr \rightarrow Expr) \\ \hline Esubst = ran\ e \end{array}$$

We can now define the unification function:

$$\begin{array}{|l} Unify : (Expr \times Expr) \nrightarrow Esubst \\ \hline \forall exp1, exp2 : Expr, es : Esubst \bullet \\ Unify(exp1, exp2) = es \Leftrightarrow \\ es\ exp1 = es\ exp2 \wedge \\ \forall es' : Esubst \bullet es'\ exp1 = es'\ exp2 \Rightarrow \\ \exists es'' : Esubst \bullet es''\ (es\ exp1) = es'\ exp1 \end{array}$$

Now while the above specification does capture what we require of a unification function it gives us no clue as to the form of the required algorithm. The implementation of the unification algorithm in Miranda is a recursive function over expressions employing recursive substitution and occur check functions. It is not clear what intermediate stages could be used when refining the specification above towards the Miranda implementation. Without such a transformation sequence the specification does little to increase the credibility of the Miranda implementation.

Even so we will still require specifications for our programs. Without a specification we cannot really be sure what we are trying to implement. We choose to use a mathematical presentation of an algorithm as our initial specification. The mathematical nature of the specification then allows us to be exact when discussing the desired implementation. The algorithmic nature of the specification also allows an easy, and credible, translation to the Miranda implementation.

2.2 Verification

Having a specification of a problem and a proposed implementation we might then wish to formally verify the implementation. We examine a small Miranda program implementing a fairly straightforward graph algorithm. The purpose is to show that even when we allow ourselves to assume a great many non-trivial results about Miranda lists[1] and to overlook the fact that they are not the same as sets we still get an unreasonably complex verification. This particular Miranda implementation has no great significance and is simply used to highlight the problems that arise when attempting a verification.

The problem is that of finding the maximal strong components in a directed graph. This description of the maximal strong components algorithm and the structure of the verification proof are taken from [Dij76, pages 192–200]. A *directed graph* is an

[1] We assume all the theorems needed to justify the otherwise unexplained steps in the derivations!

ordered pair $< V, E >$ where V is a finite set and $E \subseteq (V \times V) - I_V$, where I_V is the identity relation on V. We shall call the elements of V the *vertices* of the graph. We shall write $v_1 \mapsto v_2$ if $< v_1, v_2 > \in E$ and call $v_1 \mapsto v_2$ an *edge* from v_1 to v_2. We shall say that $v_1 \mapsto v_2$ *involves* both v_1 and v_2. A *path* is a sequence of these edges $e_1, \ldots, e_k$ such that for adjacent edges e_i and e_{i+1}:

$$e_i = v_i \mapsto v_j \wedge e_{i+1} = v_j \mapsto v_k$$

for some vertex v_j. If we have $e_1 = v_1 \mapsto v_2$ and $e_k = v_{k-1} \mapsto v_k$ then $e_1, \ldots, e_k$ is called a path *between* v_1 and v_k. We consider there to be an empty path between each vertex in the graph and itself. A *strong component* is a set of vertices such that there is a path between any pair of vertices in the set. A *maximal strong component* is a strong component to which no more vertices may be added while maintaining this property. Thus a vertex with no edges to or from it may form a singleton maximal strong component.

We have a number of results from [Dij76] that enable us to describe clearly the algorithm used to find the maximal strong components of a graph. These results will also be used in the verification proof which will be described later. These results allow us to decide when vertices may be in the same strong component and also when they cannot.

Theorem 1 *Cyclically connected vertices may belong to the same strong component.*

From the definition of a maximal strong component this gives us:

Corollary 1 *Cyclically connected vertices must belong to the same maximal strong component.*

We say that there is an edge from strong component A to strong component B if there is an edge from a vertex of A to a vertex of B. We can then define a path between strong components in a similar fashion as for vertices. We then get the following results:

Theorem 2 *Vertices of cyclically connected strong components may belong to the same strong component.*

Corollary 2 *Vertices of cyclically connected strong components must belong to the same maximal strong component.*

The above results also provide the following:

Corollary 3 *A non-empty graph must have at least one maximal strong component.*

This means that we can at least be sure that there is a maximal strong component to be found. The following theorem allows us to decide when we have discovered a maximal strong component and allows us to sub-divide the problem of finding further maximal strong components:

Theorem 3 *If we can partition the vertices of the graph into two sets A and B such that there exist no edges from A to B then:*

1. *The set of maximal strong components of the graph is independent of the presence or absence of edges from B to A.*

2. *No strong component can contain vertices from both A and B.*

Thus, as soon as we find a strong component with no outgoing edges we can use Theorem 3 to conclude that it must be a maximal strong component. We can then use the following result to find further maximal strong components:

Theorem 4 *A strong component all of whose outgoing edges are into maximal strong components is itself a maximal strong component.*

To find the maximal strong components of the graph we must start by building up strong components. Whenever we find cyclically connected strong components we can coalesce them to produce a larger strong component by Theorem 2. When we find a strong component with no outgoing edges we have found our first maximal strong component by Theorem 3. When we find further strong components with outgoing edges only to already discovered maximal strong components then we have found further maximal strong components by Theorem 4.

The obvious way to build up the strong components is by considering the edges of the graph in turn. Our goal is to obtain a partition of the vertices into the maximal strong components of the graph. If we can define a partition of the vertices relative to the edges so far considered such that it forms the maximal strong components of

the graph defined by those edges then by monotonically increasing the set of edges considered we will attain this goal. We therefore partition the edges into two sets:

- Let E_1 be the set of edges considered so far. Initially we will have $E_1 = \emptyset$ and finally we should have $E_1 = E$.

- Let E_2 be the set of edges yet to be considered. Initially we will have $E_2 = E$ and finally we should have $E_2 = \emptyset$.

Having partitioned the edges in this fashion we can partition the vertices similarly:

- Let V_1 be the set of vertices not involved in any of the edges of E_1. We will initially have $V_1 = V$ and finally we should have $V_1 = \emptyset$[2]. These vertices will form singleton maximal strong components relative to the set of edges E_1.

- Let V_3 be the set of vertices belonging to the maximal strong components of the graph found so far. These will also be maximal strong components relative to the set of edges E_1.

- Let V_2 be the set of vertices belonging to strong components of the graph not yet known to be maximal strong components of the graph. These strong components will, however, be maximal strong components relative to the set of edges E_1.

[2]We allow a vertex known to have no outgoing or incoming edges to be moved from V_1 as we can regard all the edges involving it to already be elements of E_1.

In order to employ Theorem 2 to coalesce strong components we need only look at edges that may produce cycles among these strong components. We therefore need only consider edges from vertices in V_2. In order to make this choice more orderly we impose strict limitations on the paths between the strong components formed by the vertices of V_2. We require that these strong components form a chain with exactly one path from the start to the end of the chain. We can then restrict our attention to edges from the last strong component in the chain. We have three possible cases:

- If we find an edge to a vertex in V_3 we can ignore it by Theorem 4.
- If we find an edge to a vertex in V_2 then this indicates a cycle of strong components and we can coalesce the strong components of the chain that form this cycle. Due to the strict condition we have imposed on the strong components this cycle must comprise some tail section of the chain of strong components.
- If we find an edge to a vertex in V_1 then we can add this vertex as a singleton strong component at the end of the chain.

When we find that there are no more edges from the end strong component of the chain this means we have found a maximal strong component of the graph and we can move the strong component to V_3 by Theorem 3 or Theorem 4.

This description of the desired algorithm will serve as our design specification. The Miranda implementation that resulted from this specification is shown in Figure 1.

```
msc vertices edges
      = msc' [] [] vertices
        where
        msc' arg1 [] []
                  = arg1
        msc' arg1 [] (v:arg3)
                  = msc' arg1 [([v],ef v)] arg3
        msc' arg1 ((vl,[]):arg2) arg3
                  = msc' (vl:arg1) arg2 arg3
        msc' arg1 ((vl,(ev1,ev2):el):arg2) arg3
                  = msc' arg1 (([ev2],ef ev2):(vl,el):arg2)
                              (arg3 -- [ev2])
                        , member arg3 ev2
                  = msc' arg1 ((vl,el):arg2) arg3
                        , member_of_member arg1 ev2
                  = msc' arg1 (coalesce ev2 ((vl,el):arg2) ([],[]))
                              arg3
                        , otherwise
        ef = associated_value [] edges

coalesce v ((vl,el):arg2) (vl',el')
      = (vl++vl',el++el'):arg2
            , member vl v
      = coalesce v arg2 (vl++vl',el++el')
            , otherwise
coalesce v [] (vl',el')
      = [(vl',el')]
```

Figure 1: Maximal Strong Components

The function `msc` takes two arguments, the first of which is simply a list of the vertices of the graph. The second argument is an association list which gives the list of edges from each each vertex of the graph. The subsidiary function `ef` is defined by the application of a standard association list lookup function to this second argument. Thus the the function `ef` returns a list of all the edges from a vertex. The function `msc` is defined in terms of the subsidiary function `msc'` which is a straightforward recursive function defined by cases. We shall refer to the formal arguments of `msc'` as *Arg1*, *Arg2* and *Arg3* and the arguments to the recursive calls of `msc'` as *Arg1'*, *Arg2'* and *Arg3'*. We first describe how these arguments relate to the sets of edges and vertices employed in the specification and introduce some useful abbreviations. These abbreviations will be used in the description of the verification proof.

- The third argument, *Arg3*, is simply a list of the vertices not involved in any of the edges so far considered. Thus *Arg3* corresponds directly to the set of vertices V_1. We thus introduce the following schematic Miranda definition which will be used in the proof:

  `v1 = Arg3`

  All the edges from vertices in this list have yet to be considered.

- The first argument, *Arg1*, is a list of the maximal strong components of the graph discovered so far, each of which is represented by a list of its vertices. We

introduce the following schematic Miranda definition:

```
v3 = [x|y<-Arg1;x<-y]
```

All the edges from vertices in this list of lists have been considered.

- The second argument, *Arg2*, is more complex but is essentially a list of the strong components so far built that are not known to be maximal strong components of the graph. Each of these strong components has paired with it a list of all the edges from vertices in that strong component that have yet to be considered. We introduce the following schematic Miranda definition:

```
v2 = [x|(y,z)<-Arg2;x<-y]
```

The list of edges so far considered is implicit in these three arguments. It consists of all the edges from vertices in the first and second arguments less those edges present in the second argument. We will use the following schematic Miranda definition in the proof:

```
e1 = [e|x<-v3;e<-ef x] ++
     [e|x<-v2;e<-ef x] --
     [e|(y,z)<-Arg2;e<-z]
```

These arguments also define a partition of the vertices of the graph which we will show represents the maximal strong components relative to the set of edges so far considered. We introduce the following schematic Miranda definition of this partition:

```
p = Arg1 ++
    [x|(x,y)<-Arg2] ++
    [[x]|x<-Arg3]
```

The verification proof is an inductive argument over the cases of the function `msc'`. The invariant to be established is that the current partition of the vertices, represented by `p`, defines the maximal strong components relative to the set edges so far considered, represented by `e1`, at all applications of `msc'`. To aid in the proof of this invariant the following properties will also be shown to be invariant:

1. There are no edges in `e1` involving vertices from `v1`.

2. There are no edges in `e1` from vertices in `v3` to vertices in `v2`.

3. The edges associated with each strong component in *Arg2* are from vertices in that strong component.

4. There is a single path along the chain of strong components in *Arg2*.

5. *Arg1* contains only maximal strong components of the graph.

2.2.1 Base Case

The base case is the initial application of the function `msc'`:

```
msc' [] [] vertices
```

If we assume that we have available a range of theorems concerning lists and list comprehension then we can prove the following equalities concerning the arguments:

```
v1 = vertices
v2 = [x|(y,z)<-[];x<-y]
   = []
v3 = [x|y<-[];x<-y]
   = []
e1 = [e|x<-[];e<-ef x] ++
     [e|x<-[];e<-ef x] --
     [e|(y,z)<-[];e<-z]
   = []
p = [] ++
    [x|(x,y)<-[] ++
    [[x]|x<-vertices]
  = [[x]|x<-vertices]
```

Thus we have considered no edges and as we would expect all the vertices produce singleton maximal strong components. The invariant properties 1-5 are trivially true as we have considered no edges and we have an empty chain of strong components.

2.2.2 Induction Step

The induction step consists of assuming that the invariant conditions are true for the formal arguments in the definition of `msc'` and then proving that they still hold for the arguments to the recursive calls. The first clause of the definition provides the result of the function and is dealt with later. We must consider the second, third and fourth clauses of the definition.

- The second clause of the function defines how we start building a strong component if there are none in the chain already. We simply pick an arbitrary vertex, in this case the head of the list, and make it a singleton strong component.

```
msc' arg1 [] (v:arg3)
      = msc' arg1 [([v],ef v)] arg3
```

This new strong component has a list of all the edges from it associated with it by the function `ef`. This function returns a list of all the edges from a vertex, it is defined by a standard function over association lists. We can prove the following equalities concerning the arguments:

```
v1 = v:arg3
v2 = []
v3 = [x|y<-arg1;x<-y]
e1 = [e|x<-v3;e<-ef x] ++
     [e|x<-[];e<-ef x] --
     [e|(y,z)<-[];e<-z]
   = [e|x<-v3;e<-ef x]
p = arg1 ++
    [x|(x,y)<-[]] ++
    [[x]|x<-v:arg3]
  = arg1 ++
    [[v]] ++
    [[x]|x<-arg3]
```

If we name the corresponding values for the recursive call of `msc'` as `v1'`, `v2'` etc. then we can prove the following equalities concerning these arguments:

```
v1' = arg3
v2' = [x|(y,z)<-[([v],ef v)];x<-y]
    = [v]
v3' = [x|y<-arg1;x<-y]
    = v3
e1' = [e|x<-v3;e<-ef x] ++
      [e|x<-[v];e<-ef x] --
      [e|(y,z)<-[([v],ef v)];e<-z]
    = [e|x<-v3;e<-ef x]
p' = arg1 ++
     [x|(x,y)<-[([v],ef v)]] ++
     [[x]|x<-arg3]
   = arg1 ++
     [v] ++
     [[x]|x<-arg3]
```

So we have not considered any more edges and as we would expect the partition of the vertices remains unchanged. Each of the invariant properties 1-5 is again trivially true.

- The third clause of the function defines what we must do when we have considered all the edges from the vertices of the strong component at the end of the chain in *Arg2*.

```
msc' arg1 ((v1,[]):arg2) arg3
      = msc' (v1:arg1) arg2 arg3
```

Having considered all the edges from the vertices of the strong component `v1`, as shown by the fact that the associated list of edges is empty, we can conclude it is a maximal strong component. This follows from the invariant properties

1-5 as we know that there can be no edges from `v1` into `v1` or back to any other strong component in `v2`. Thus it either has no outgoing edges in which case we can use Theorem 3, or all its outgoing edges are into `v3`, in which case we can use Theorem 4. As above we can prove that we have not added to list of edges considered or changed the partition of the vertices, so `p'` must still represent the maximal strong components relative to `e1'`. All the invariant conditions 1-5 can again be shown to be true.

- The third clause of the function defines what happens when we consider a new edge.

```
msc' arg1 ((v1,(ev1,ev2):el):arg2) arg3
```

This clause has three guarded right hand side expressions, selected by which of the three sets of vertices contains the vertex `ev2`. We can prove the following equalities concerning the arguments:

```
v1 = arg3
v2 = [x|(y,z)<-(v1,(ev1,ev2):el):arg2;x<-y]
   = v1 ++ [x|(y,z)<-arg2;x<-y]
v3 = [x|y<-arg1;x<-y]
e1 = [e|x<-v3;e<-ef x] ++
     [e|x<-v2;e<-ef x] --
     [e|(y,z)<-(v1,(ev1,ev2):el):arg2;e<-z]
   = [e|x<-v3;e<-ef x] ++
     [e|x<-v2;e<-ef x] --
     ([(ev1,ev2)] ++ el ++ [e|(y,z)<-arg2;e<-z])
```

```
p = arg1 ++
    [x|(x,y)<-(vl,(ev1,ev2):el):arg2] ++
    [[x]|x<-arg3]
  = arg1 ++
    vl ++ [x|(x,y)<-arg2] ++
    [[x]|x<-arg3]
```

We shall now deal with each of the three right hand sides in turn.

- The first case defines what we do if the new edge connects to a vertex not involved in any of the edges so far considered.

```
= msc' arg1 (([ev2],ef ev2):(vl,el):arg2)
            (arg3 -- [ev2])
      , member arg3 ev2
```

The guard will only be true if `ev2` is in the set `v1`. In this case we add `ev2` as a singleton strong component at the end of the chain and remove it from the third argument. We can prove the following inequalities concerning the arguments to the recursive cal of `msc'`:

```
v1' = arg3 -- [ev2]
    = v1 -- [ev2]
v2' = [x|(y,x)<-([ev2],ef ev2):(vl,el):arg2;x<-y]
    = [ev2] ++ vl ++ [x|(y,x)<-arg2;x<-y]
    = [ev2] ++ v2
v3' = [x|y<-arg1;x<-y]
    = v3
e1' = [e|x<-v3;e<-ef x] ++
      [e|x<-[ev2] ++ v2;e<-ef x] --
      [e|(y,z)<-([ev2],ef ev2):(vl,el):arg2;e<-z]
    = [e|x<-v3;e<-ef x] ++
```

```
      [e|x<-[ev2];e<-ef x] ++
      [e|x<-v2;e<-ef x] --
      (ef ev2 ++ el ++ [e|(y,z)<-arg2;e<-z]
    = [e|x<-v3;e<-ef x] ++
      [e|x<-v2;e<-ef x] --
      (el ++ [e|(y,z)<-arg2;e<-z])
p' = arg1 ++
     [x|(x,y)<-([ev2],ef ev2):(vl,el):arg2] ++
     [[x]|x<-arg3 -- [ev2]]
   = arg1 ++
     [ev2] ++ vl ++ [x|(x,y)<-arg2] ++
     [[x]|x<-arg3] -- [ev2]
   = arg1 ++
     vl ++ [x|(x,y)<-arg2] ++
     [[x]|x<-arg3]
```

From these we can see that we have the following relationship between the sets of considered edges and the partitions of the vertices:

```
e1' = e1 ++ [(ev1,ev2)]
p' = p
```

So we have considered one more edge but have not changed the partition. From the invariant conditions we know that there are no edges from `v1'` in `e1` and hence none in `e1'`. We can therefore appeal to Theorem 3 to conclude that the maximal strong components of the graph relative to `e1'` are independent of the newly considered edge. We can also prove that the invariant conditions still hold.

- The second case defines what we do if the new edge connects to one of the already found maximal strong components of the graph.

```
= msc' arg1 ((v1,e1):arg2) arg3
      , member_of_member arg1 ev2
```

The guard will only be true if ev2 is in the set v3. In this case we can ignore the new edge in view of Theorem 4. We can prove the following equalities concerning the recursive call of msc':

```
v1' = arg3
    = v1
v2' = [x|(x,y)<-(v1,e1):arg2]
    = v2
v3' = [x|y<-arg1;x<-y]
    = v3
e1' = [e|x<-v3;e<-ef x] ++
      [e|x<-v2;e<-ef x] --
      [e|(y,z)<-(v1,e1):arg2;e<-z]
    = [e|x<-v3;e<-ef x] ++
      [e|x<-v2;e<-ef x] --
      (e1 ++ [e|(y,z)<-arg2;e<-z]
p' = arg1 ++
     [x|(x,y)<-(v1,e1):arg2] ++
     [[x]|x<-arg3]= =arg1 ++
     v1 ++ [x|(x,y)<-arg2] ++
     [[x]|x<-arg3]
```

From these we can again see that we have the following relationship between the sets of considered edges and the partitions of the vertices:

```
e1' = e1 ++ [(ev1,ev2)]
p' = p
```

So we have considered one more edge but have not changed the partition. As the considered edge is into v3' and from the invariant conditions 1-5

we know there are no edges in `e1` from `v3` and hence none in `e1'` from `v3'`, we can appeal to Theorem 4 to show that the partition is independent of this edge. We can also prove that the invariant conditions 1-5 still hold.

- The third case defines what we do if the new edge creates a cycle of strong components.

```
= msc' arg1 (coalesce ev2 ((v1,e1):arg2) ([],[]))
            arg3
      , otherwise
```

We will only reach this case if `ev2` is in the set `v2`. The new edge is therefore from the end strong component in the chain to some strong component before it in the chain. As the invariant conditions 1-5 tell us there is a path along the chain, we must have cyclically connected strong components at the end of the chain, and by Theorem 2 we can coalesce them. At this point the proof becomes far more complex due to the presence of the `coalesce` function which performs this action. We must prove that this function does not alter the set `v2` and that the new strong component produced is produced from a valid appeal to Theorem 2. We must also show that `p'` is still a partition of the vertices and that `e1'` is increased by only the one new edge. It is easy to prove the invariant conditions if one has proved the above.

2.2.3 Result Case

The first clause of the function defines the result returned by the function. This result should be the maximal strong components of the graph.

```
msc' arg1 [] []
      = arg1
```

We can prove the following equalities concerning the arguments:

```
v1 = []
v2 = []
v3 = [x|(y,z)<-arg1;x<-y]
e1 = [e|x<-v3;e<-ef x] ++
     [e|x<-[];e<-ef x] --
     [e|(y,z)<-[];e<-z]
   = [e|x<-v3;e<-ef x]
p = arg1 ++
    [x|(x,y)<-[]] ++
    [[x]|x<-[]]
  = arg1
```

By our inductive hypothesis `p` is a partition of the vertices into the maximal strong components of the graph relative to the edges considered. As `arg1` is therefore a partition of the vertices of the graph, `e1` must contain all the edges in the graph and thus `arg1` contains the maximal strong components of the graph. We are therefore returning the desired result.

2.2.4 Conclusion

There are a number of points to be made about this example. Firstly, even though it is a small program of only twenty four lines and we allowed ourselves to assume some non-trivial theorems, we still could not produce a complete description of the proof at the level of abstraction chosen. Secondly, the structure of the proof in the example is in fact a long way from being a formal proof. We chose to ignore the fact that lists with the same elements in different orders are not equal. We also avoided entirely any discussion of the termination of the function, which in this case does not appear to be an easy thing to prove.

It is therefore clear that without some automated theorem proving support it is not realistic to expect to be able to verify small functional programs, let alone the large systems we shall implement. My personal experience of using the LCF [GMW79] system is that proving results about even quite small functions is not easy. This was partly due to some bugs in the implementation I was using but mostly due to the large number of cases that had to be proved. I found I relied almost entirely on the powerful rewriting facilities of LCF. This meant that when they failed to rewrite things as I hoped I was left with little idea of how to proceed. It is to be hoped that theorem provers tailored to the requirements of functional programming languages will provide more powerful tools supporting reasoning at a higher level.

2.3 Informal Approach

Our approach is to require the existence of a mathematical presentation of the specification of the algorithm to be implemented. We shall then attempt to write our functional programs in such a fashion as to make them amenable to the form of informal reasoning we were forced to resort to in the proof in §2.2.2.

We choose the implementation of the constructive reals [BB85] as the example to demonstrate this approach. As the mathematical specification of the constructive reals already exists we can avoid any criticisms that we have tailored it to make the transformation to Miranda easy. This example also allows us to demonstrate the power of the algebraic and abstract types and the use of potentially infinite structures. In this implementation we concentrate on fidelity to the specification rather than the efficiency of the implementation.

In constructive mathematics, existence proofs require the presentation of a method for producing the required value. These proofs can thus serve as the basis of an implementation as we can use Miranda to implement the constructions in the proofs. The real numbers are implemented as an abstract type with the signature containing various functions over real numbers. The underlying implementation type is a simple algebraic type which would not, by itself, provide the type security required. The implementation of constructive reals is based on a type secure implementation of arbitrary precision rationals which in turn is based on a type secure implementation

of arbitrary size integers. The abstract type implementation of integers utilises the fact that Miranda will perform integer arithmetic for arbitrarily large integers. The underlying implementation type is the Miranda numeric type. The implementation equations simply ensure that only integer numeric values may be used to create values of the abstract integer type and otherwise apply the appropriate Miranda numeric operators. The abstract type implementation of rationals represents them as pairs of integers. The implementation equations simply ensure that common factors are removed when rationals are created. To aid the clarity of exposition we shall not provide definitions of all the subsidiary functions employed but simply describe their function.

We take our definitions and theorems from the start of chapter two of [BB85]. We start by defining real numbers in terms of the rationals Z [BB85, page 18]:

Definition 1 *A sequence* (x_n) *of rational numbers is* regular *if*

$$|x_m - x_n| \leq m^{-1} + n^{-1} \quad (m, n \in Z)$$

A real number *is a regular sequence of rational numbers. The set of real numbers is denoted by* $\mathbb{R}$.

This definition is encoded in Miranda by both the definition of the following algebraic implementation type[3] for the abstract type and the conditions imposed by the

[3]In practise we add an extra constructor `RQ q` for representing rationals directly and add clauses to all the implementation equations to cope with this form of real number efficiently.

functions provided in the signature for creating real numbers.

```
r == r'
r' ::= R [q]
```

The type q is the abstract type which is a Miranda implementation of arbitrary precision rationals. This abstract type is itself defined in terms of an abstract type z of arbitrary sized integers. Before we can define any implementation equations we must define some of the subsidiary functions they require [BB85, page 19]:

Definition 2 *The rational number x_n is called the n^{th}* rational approximation *to the real number $x \equiv (x_n)$.*

Corresponding to this definition we have the following Miranda subsidiary function:

```
nthapprox ::  q -> [q] -> q
```

This function simply takes the n^{th} element of the list of rationals representing the real number. The following definition will be required later [BB85, page 19]:

Definition 3 *We associate with each real number $x \equiv (x_n)$ an integer K, such that:*

$$|x_n| < K_x \quad (n \in \mathrm{Z})$$

This is done by letting K_x be the least integer which is greater than $|x_1| + 2$. We call K_x the canonical bound *for x.*

This in turn gives rise to the following subsidiary function definition which uses the function `qs` to return the sum of two rationals and functions `qent` and `qabs` for extracting the integer part and absolute value of a rational and `q_3` which is the rational representation of the integer three:

```
bound (R (q1:ql)) = qent (qs (qabs q1) q_3)
```

The following definition [BB85, page 19] gives the sum, product and maximum of two real numbers and also the negation of a real number. The real number corresponding to a rational is defined to be the regular sequence of rationals obtained by repeating that rational.

Definition 4 *Let $x \equiv (x_n)$ and $y \equiv (y_n)$ be real numbers with respective canonical bounds K_x and K_y. Write*

$$k \equiv max\{K_x, K_y\}$$

Let α be any rational number. We define

1. $x + y \equiv (x_{2n} + y_{2n})_{n=1}^{\infty}$
2. $xy \equiv (x_{2kn} y_{2kn})_{n=1}^{\infty}$
3. $max\{x, y\} \equiv (max\{x_n, y_n\})_{n=1}^{\infty}$
4. $-x \equiv (-x_n)_{n=1}^{\infty}$

5. $\alpha^* \equiv (\alpha, \alpha, \alpha, \ldots)$

It can be shown [BB85, page 20] that these do indeed define real numbers. These definitions employ the notation x_{kn} to refer to the sequence of kn^{th} rational approximations to x for $n = 1, \ldots, \infty$ and constant k. We therefore have the following subsidiary function which simply extracts this sequence of rational approximations:

```
offset ::  q -> [q] -> [q]
```

This filters the sequence of rationals representing a real number with a state representing the index in the sequence and a predicate which selects every k^{th} element.

We can now describe the implementations corresponding to some of these definitions. The implementation of the real number product employs the the higher order function `map2` for mapping a binary function over two lists. It also uses the functions `qp` and `maxq` which calculate the product and maximum of two rational numbers.

```
rp (R ql1) (R ql2)
      = R (map2 qp (offset i ql1) (offset i ql2)
        where
        k1 = bound (R ql1)
        k2 = bound (R ql2)
        k = maxq k1 k2
        i = qp q_2 k
```

This function can clearly be seen to correspond to definition 2 above. The sum, maximum and negation can be implemented by similarly simple functions. This final function can be used to create a real number from a rational number:

```
mqr q = R ql
        where
        ql = q :  ql
```

This function satisfies the condition that the sequence of rationals representing a real number should be regular.

We now define two inequality relations on real numbers [BB85, page 21]:

Definition 5 *A real number* $x \equiv (x_n)$ *is* positive, *or* $x \in \mathbb{R}^+$, *if*

$$x_n > n^{-1}$$

for some n *in* $\mathbb{Z}^+$. *A real number is* nonnegative, *or* $x \in \mathbb{R}^{0+}$, *if*

$$x_n \geq -n^{-1} \quad \left(n \in \mathbb{Z}^+\right)$$

The first part of the definition gives rise to the following function: This uses the functions `qgr`, which compares two rational numbers, and `qi` which calculates the inverse of a rational number.

```
rgr0 (R ql)
      = rggr0' q_1 ql
        where
        rgr0' i (q:ql)
                  = True, qgr q (qi i)
                  = rgr0' (qs i q_1) ql , otherwise
```

This function searches down the real number representation looking to satisfy the condition. This function will only ever return `True` as an answer and otherwise will fail to terminate.

The final function we implement is the real inverse. This function is interesting as we use the witness of the proof of the existence of an inverse to construct the inverse [BB85, pages 24–25]:

Proposition 1 *Let x be a nonzero real number, so that $|x| \in \mathbb{R}^+$. There exists a positive integer N with $|x_m| \geq N^{-1}$ for $m \geq N$. Define*

$$y_n \equiv (x_{N^3})^{-1} \quad (n < N) \quad \text{and} \quad y_n \equiv (x_{nN^2})^{-1} \quad (n \geq N)$$

Then

$$x^{-1} \equiv (y_n)_{n=1}^{\infty}$$

is a real number which is positive if x is positive, and negative if x is negative; also $xx^{-1} = 1^*$.

We shall merely give those portions of the proof that are used as the basis of our implementation. Given that we can find the number N then we get the following implementation of real number inverses:

```
ri (R ql)
      = R (find_yql q_1)
        where
        n = findN q_1 ql
        n2 = qp n n
        yn_ls_N = qi (nthapprox (qp n2 n) ql)
        find_yql i
                  = yql_gen q_1 (offset n2 ql)
                        , qeq in n
                  = yn_ls_N : find_yql (qs q_1 i)
```

```
                , otherwise
yql_gen i (q:ql)
          = map qi (q:ql)
                , qeq i n
          = yql_gen (qs q_1 i) ql
                , otherwise
```

We use the function `findN` to calculate the number N, this is done in the sub-definition of `n`. Given the value of N we can then calculate the value of x_{N^3}, this is done in the sub-definition of `yn_ls_N`. This value is then used by the function `find_yql` to construct the initial segment of the sequence of rationals representing the inverse, that is for $n < N$. The function `yql_gen` is then used to throw away the first N elements of the argument rational sequence and to construct the tail of the inverse's rational sequence from the remainder.

We now discuss the construction of the number N. Our assumption gives us that $|x| \in \mathbb{R}^+$ so we employ the following lemma and its proof [BB85, page 21]:

Lemma 1 *A real number $x \equiv (x_n)$ is positive if and only if there exists a positive integer N such that*

$$x_m \geq N^{-1} \quad (m \geq N)$$

By the assumption and Definition 5 we have $|x_i| > i^{-1}$ for some $i \in \mathbb{Z}^+$. We choose $N \in \mathbb{Z}^+$ with $2N^{-1} \leq |x_i| - i^{-1}$. This leads us to define the following functions:

```
findN n (qn:ql)
      = findN (qs q_1 n) ql
```

```
            , qle qn_abs in
      = findN' (qs q_1 n) qn_in ql
            , otherwise
        where
        qn_abs = qabs qn
        in = qi n
        qn_in = qd qn_abs in

findN' n' qn_in (qn':ql)
      = n'  , qge qn_in (qs in' in')
      = findN' (qs q_1 n') qn_in ql
            , otherwise
        where
        in' = qi n'
```

Thus the first function, `findN`, calculates the i and $|x_i| - i^{-1}$ and the second function, `findN'`, calculates N.

The implementation of the constructive reals described in this section shows the style of mathematical specification we shall adopt. The credibility of the implementation will depend crucially on the clarity of expression of mathematical concepts in terms of constructions in the programming language. This example also provides a good example of the separation of concerns that is provided by the abstract type mechanism of Miranda, in that we can consider the correctness of the implementation of the rationals separately.

In the remainder of this thesis we will precede all implementations by a specification in this style.

Chapter 3

Type Checking "Miranda"[1]

While the type system of "Miranda" provides security combined with power it can sometimes cause problems. The polymorphic nature of the type system means that the type deduced for a variable is constrained by all the occurrences of that variable. If a definition fails to type check, or is assigned an unexpected type, it can be extremely difficult to locate the source of the problem. Our first example implementation is an attempt to tackle this problem.

We implement a system that not only type checks a "Miranda" script but also annotates all the expressions, and sub-expressions, in the script with their deduced types. It then allows the user to interactively examine these deduced types. It is

[1]Miranda is a Trademark of Research Software Ltd.
In this chapter we shall use a language resembling Miranda as our example language, and will refer to it as "Miranda". However, this chapter should not be taken as defining in any way the language Miranda, for such a definition one should see the documents from Research Software Ltd.

hoped that examination of the types of sub-expressions in a badly typed script will aid in the location of the source of type errors. We also intend that the system serve a tutorial function in explaining the type checking of Miranda scripts through examination of well typed scripts.

We present a formal account of the details of the type discipline and type checking of "Miranda". If we were simply interested in presenting the typing rules of Miranda we would first describe a translation from the full "Miranda" syntax to an extended λ-calculus [Jon87] and then describe its typing as an inference system [Dam85]. We, however, require that the presentation serve as an aid to the user of the system in understanding the type checking process and locating type errors. As the interactive examination of the deduced types in a script must be performed over the syntactic structures understood by the user the type checking must be described in terms of these same syntactic structures. Because the type checking is sequential and the particular sequence in which occurrences of a variable affect its deduced type can be very important we also choose to make this sequencing explicit. We therefor adopt the same style employed by Milner, extended to cope with the richer syntax of "Miranda". The major extension is that needed to cope with mutually recursive groups of definitions. We choose not to employ a higher order function to describe the type checking of sequences of sub-expressions in order that the presentation remain useful to the novice "Miranda" user. The higher order function we would have to use

otherwise would be something like:

```
type_seq = foldl type_item (id_subst,[])

type_item item (subst,item_list)
= (subst'',item' :  map (instantiate subst') item_list)
  where
  (subst',item') = type_check item
  subst'' = compose subst' subst
```

This would also not correctly mimic the behaviour of Miranda in that the type error messages produced would not be the same as those produced by Miranda[2].

We first describe the type discipline which "Miranda" enforces and explain algebraic and abstract type definitions. We then present an abstract syntax for "Miranda" which will be the basis for the later definitions. Given this abstract syntax we define the concept of well typing in terms of type conditions on these prefixed-sub-expressions. We then present the well typing algorithm $\mathcal{M}$. We finally describe some details of the implementation of the system.

3.1 Type Discipline

"Miranda" is a higher order polymorphic language which is strongly typed. "Miranda" has a compile time type checker that can deduce the types with no need for user provided type descriptions.

[2]It was also necessary, due to inefficiencies in the Miranda implementation, to determine when one could avoid applying the resulting substitutions.

"Miranda's" type discipline enforces certain conditions on a script. Many of these are fairly straightforward, such as requiring the arguments of numeric operators such as addition, + to be of numerical type. The interesting conditions are those that allow and constrain the polymorphism in a script.

The idea of definitions with polymorphic type is central to "Miranda's" type system. It seems natural that operators like = be able to be applied to arguments of various types, and this consideration follows through to definitions of other operators[3].

For example we certainly expect to be able to write both the following list expressions:

```
1:[2,3,4]
'A':['B','C','D']
```

Here : (infix CONS) can be seen to be taking two different types:

```
num -> [num] -> [num]
char -> [char] -> [char]
```

However, we do not want to able to write things like:

```
1:['A']
```

What we need is for : to have a type that allows it to take the two first types but disallows the third expression. This is the polymorphic type;

[3]The polymorphism of = is *ad hoc* rather than *parametric* but the point holds.

```
* -> [*] -> [*]
```

where * is a "Miranda" type variable. This is a polymorphic type because it contains type variables. The first two types are instances of this polymorphic type. In the first * becomes **num** and in the second it becomes **char**. The details of "Miranda's" type discipline concern the creation and particular application of such polymorphic types.

As can be seen from the above, ':' can take any instances of its polymorphic type within the script. "Miranda's" top level definitions of functions and patterns behave in a similar way. Thus if we have a definition of a list reversing function:

```
reverse (x:l) = reverse l ++ [x]
reverse [] = []
```

then **reverse** has the polymorphic type `[*] -> [*]` and can take any instances of this type at its occurrences in the script. Thus in the expressions:

```
reverse [1,2,3]
reverse ['A','B','C']
```

the function **reverse** takes the following types:

```
[num] -> [num]
[char] -> [char]
```

Notice that the first use of the **reverse** function is not taken as meaning that reverse must be of type:

```
[num] -> [num]
```

Thus we have the following:

Condition 1 *Variables declared by top level definitions can take any instance of their polymorphic type, as deduced from their definition, at their occurrences in the rest of the script.*

The ability to have mutually recursive definitions adds a complication to this general case. This is because the mutual recursion may result in the types of the variables being defined being mutually dependent, for example:

```
f x y z = (1,y,z) :  g x y z
g x y z = (x,'a',z) :  f x y z
```

Here the type of the result of each function depends of the type of the result of the other function. If the types of two, or more, definitions are mutually dependent then they must be type checked together. The types for the variables being defined being deduced from their occurrences in all the definitions. There is no simple way to determine when two definitions are type dependent so we simply assume they are if they are mutually recursive. The following condition must be imposed:

Condition 2 *All those variables defined in a group of mutually recursive definitions must take the same type at all their occurrences within the group.*

This approach, which assumes we know the calling graph of the script, differs from that proposed by Mycroft [Myc84]. Thus the functions **f** and **g** will both have the type:

```
num -> char -> * -> [(num,char,*)]
```

Variables defined in a mutually recursive group of definitions can still take instances of their types, as deduced from their definitions, at their occurrences outside the group.

The user may provide type descriptions for top level definitions but these are unnecessary as the types can be deduced[4]. If present, the declared type must be an instance of the deduced type, and will serve only to restrict the types of those variables being defined. For instance if we had the following:

```
reverse2 ::  [bool] -> [bool]

reverse2 (x:l) = reverse2 l ++ [x]
reverse2 [] = []
```

Then **reverse2** would only be allowed to appear at type:

```
[bool] -> [bool]
```

in the rest of the script. This gives us the following condition:

Condition 3 *The user type description for a definition must be an instance of the type deduced from the definition.*

[4]In fact certain recursive definitions, where the variable being defined occurs at multiple types in the definition, will only type check if a type description is provided.

There are also restrictions on the types taken by those variables being declared on the left and right hand sides of a definition, and the types of the formal parameters in a function definition.

The motivation for these restrictions can be seen if we consider the following illegal function definition (remember that all the elements of a list must be of the same type).

```
list_pair a = ([a,1],[a,'A'])
```

Such a definition will be excluded by the following condition:

Condition 4 *All variables appearing in the formal parameter patterns of a definition must take the same type at all their occurrences on the right hand side of the definition.*

The consequence of this is that when an argument is of polymorphic type its occurrences in the right-hand-side of the definition are not as "fully" polymorphic as one might expect. Consider the following definition:

```
pair_list_fun r a b = (r a,r b)
```

One might expect,

```
pair_list_fun reverse
```

to be a function which reverses two lists of arbitrary type but this is not the case. This is because the formal parameter `r` must take the same type at both its occurrences on the right hand side and thus the other formal parameters `a` and `b` must be of the same type. Thus `pair_list_fun` has the type:

```
(* -> **) -> * -> * -> (**,**)
```

So the application of `pair_list_fun` to `reverse` has type:

```
[*] -> [*] -> ([*],[*])
```

The function `pair_reverse` as defined by:

```
pair_reverse a b = (reverse a,reverse b)
```

would have the more general type:

```
[*] -> [**] -> ([*],[**])
```

This is because `reverse`, being defined at the top level, can take different instances of its defined type at its two occurrences on the right hand side of this definition.

It is not always convenient to define functions at the top level, and we would like to be able to achieve this level of polymorphism with local definitions. For example, we would like the following function to reverse two lists of arbitrary type:

```
pair_rev a b = (rev a,rev b)
               where
               rev (x:l) = rev l ++ [x]
               rev [] = []
```

This can clearly be achieved if we type check the locally defined reverse function, `rev`, and then allow it to take any instance of its type, as deduced from its definition, at its occurrences in the right hand side expression. This is, however, not the correct condition to impose. Consider the following function definition:

```
tester a = (test 1,test 'A')
           where
           test x = a <= x
```

The formal parameter **a** can clearly be of any type, and thus we would deduce the type of the local function **test** to be:

```
* -> bool
```

If we then let it take arbitrary instances of this type in the right hand side expression it occurs at these two types:

```
num -> bool
char -> bool
```

But this analysis cannot be correct as it implies that we able to perform both these inequalities:

```
a <= 1
a <= 'A'
```

and this cannot be the case. The essential point to notice is that in the first case the the type of **rev** can be deduced without reference to the types of the parameters of the **pair_rev** function, whereas in the second case the type of **test** depends on the type of the parameter **a** of the **tester** function. These considerations lead to the following condition on local definitions:

Condition 5 *Variables declared by local definitions can, at their occurrences in the right hand side expressions, take any instance of their polymorphic type, as deduced from their definition, as long as this instance does not involve type variables associated with the formal parameters of any enclosing definition.*

The other variable-declaring expressions in a "Miranda" script are the generators in a ZF expression. We have a similar condition here to that for the variables in formal parameter patterns:

Condition 6 *All the variables declared by generators in ZF expressions must take the same type at all their occurrences in the ZF expression.*

"Miranda" has three kinds of user defined types; synonym types, algebraic types and abstract types. Type synonyms can be regarded simply as transparent rewrite rules and are of little interest here.

An algebraic type definition introduces a new type and certain constructors associated with it. The constructors associated with an algebraic type can take arbitrary instances of their defined types at their occurrences in the script. An example is this useful type:

```
value * ::= Value * | None
```

where we have the two constructors of the following types:

```
Value ::  * -> value *
None ::  value *
```

Algebraic type definitions may also have associated laws[5]. The only condition imposed here is that the left and right hand sides of the law be the same type as the associated algebraic type.

Abstract type definitions are more interesting. Consider the following definition.

```
abstype stack *
with  empty :  stack *
      pop :  stack * -> (*,stack *)
      push :  * -> stack * -> stack *

stack * == [*]

empty = []
pop l = (hd l,tl l)
push x l = x:l
```

Here we have an abstract type definition of a type which we hope captures something about stacks.

We have introduced a new type `stack *` which has the three associated functions `empty`, `pop` and `push` with the types shown in the signature. These functions can take arbitrary instances, at their occurrences in the script, of their types as defined in the signature.

We then have the definition giving the implementation type that will correspond to `stack *`.

This is followed by the implementation equations for the three associated functions.

[5]This feature has been removed in version 2.0 of Miranda.

The type discipline we describe here is more general than that described by Turner [Tur85] and implemented in version 0.378 of Miranda[6].

The implementation equations are defined on the implementation type. Thus the implementation equations above give rise to the following types:

```
empty ::  [*]
pop ::  [*] -> (*,[*])
push ::  * -> [*] -> [*]
```

In the rest of the script we want to be able to write both the following, so that we can have stacks of various types:

```
push 1 empty
push 'A' empty
```

Thus we have the following:

Condition 7 *Variables defined in an abstract type definition can take arbitrary instances of their polymorphic type, as given in the abstract type definition, at their occurrences in the rest of the script.*

The following is an illegal expression that we do not wish to allow:

```
push 1 empty = [1]
```

[6]Versions 0.978 and later of Miranda now implement a type discipline corresponding to that described here

We must have that the types `stack *` and `[*]` are not equal and will not unify. This gives the following condition:

Condition 8 *The correspondence between the abstract type and the implementation type is hidden in the rest of the script.*

In the implementation equations we obviously want to know about the correspondence between the abstract type and its implementation type and would like to be able to use functions defined on either interchangeably. We therefore have the following condition:

Condition 9 *The correspondence between the abstract type and its implementation type is available throughout the implementation equations*[7].

In version 0.378 of Miranda this is not the case: the type correspondence is only known for those variables defined in the signature of the abstract type definition[8].

Finally we must check that the types given in the abstract type definition are instances of those deduced by the typing of the implementation equations.

Condition 10 *The types given in the signature of an abstract type definition must be instances of the type deduced from the implementation equations.*

[7]It is important to note that this does *not* include any subsidiary definitions used by the implementation equations.

[8]In Miranda version 1.998 there are still restrictions on the implementation type synonyms and the implementation equations.

3.2 Abstract Syntax

The *abstract syntax* given in Table 1 includes all the major features of "Miranda"; it excludes some uninteresting cases which are merely syntactic sugar. Each of the productions captures some essential feature of "Miranda".

In this abstract syntax we only represent the most complex forms that a "Miranda" script can take. We do this as we believe that the details of the prefixed-sub-expressions etc. in the simpler cases would only complicate the presentation without adding anything, and that these cases can easily be derived from those given.

This presentation of an abstract syntax for "Miranda" retains certain features that make the definition of the well typing algorithm more complex than it might otherwise be. This is because in this form it is easier to derive a type checking function from the well typing algorithm.

In the following we shall refer to *expressions*; these will be all those forms given by any of the productions in Table 1. We shall also refer to *simple expressions*: these will be all those forms given by the productions for e given in Table 1. We will also refer to *patterns*: these are restricted forms of expressions containing only lists, tuples and constructor application.

We now explain the various productions of the abstract syntax given in Table 1.

i) A "Miranda" script, regarded as a set of declarations which is partitioned into mutually recursive groups. Each declaration in a "Miranda" script may depend

$script$	$\Rightarrow$	$md_1 \cdots md_n$	i)
md	$\Rightarrow$	$d_1\ \tau_1 \cdots d_n\ \tau_n$	ii)
d	$\Rightarrow$	$c_1 \cdots c_n$	iii)
	$\mid$	$pat = rhs$	iv)
	$\mid$	$pat => rhs$	v)
c	$\Rightarrow$	$g\ pat_1 \cdots pat_n = rhs$	vi)
rhs	$\Rightarrow$	$rhsl$ <u>where</u> md	vii)
$rhsl$	$\Rightarrow$	$ge_1 \cdots ge_n$	viii)
ge	$\Rightarrow$	e, grd	ix)
e	$\Rightarrow$	$x\ op\ y$	x)
	$\mid$	$mop\ x$	xi)
	$\mid$	$x\ x_1 \cdots x_n$	xii)
	$\mid$	$(x_1, \ldots, x_n)$	xiii)
	$\mid$	$[x_1, \ldots, x_n]$	xiv)
	$\mid$	$[x, y\ ..\ z]$	xv)
	$\mid$	$[e \mid q_1; \cdots; q_n]$	xvi)
	$\mid$	var	xvii)
	$\mid$	str	xviii)
	$\mid$	chr	xix)
	$\mid$	num	xx)
q	$\Rightarrow$	e	xxi)
	$\mid$	$pat \leftarrow x$	xxii)
	$\mid$	$pat \leftarrow x, y$	xxiii)

Table 1: "Miranda" Abstract Syntax

on various other declarations in the script in such a way as to produce a group of mutually recursive declarations. These mutually recursive groups are topologically sorted by their calling graph dependencies, each mutually recursive group possibly being dependent on the previous groups but not being dependent on the later groups after the sorting.

These declarations include the implementing definitions for abstract type definitions and the laws for algebraic type definitions. The laws do not declare any variables, except locally, so they will all appear at the end of the list of declarations due to the topological sorting.

The effect of the type information from the abstract and algebraic type definitions is not present in this abstract syntax and will be explained later.

ii) A group of one or more mutually recursive declarations. The τ_i are the user provided types for these declarations. In general these need not be present and "Miranda" will deduce the types for the user. However, as the presence of user assigned types is the more complex case in terms of type checking, we only present that case. Note that when declarations appear as subsidiary definitions the user cannot provide type definitions and there will then be no τ_i.

Laws get the type from their associated algebraic type declaration. The implementation equations for an abstract type get their type from the signature of the abstract type declaration.

iii) A function definition can be defined by multiple clauses using pattern matching on the arguments: there may be only one clause.

iv) A pattern definition can have only one clause.

v) A law defines a rewrite rule for an algebraic type.

vi) One clause of a multiple clause function definition of the function g.

vii) The right hand side of a definition may have subsidiary definitions introduced by a where clause. All the definitions in a where clause are regarded as being mutually recursive regardless of whether they actually are or not. Notice that the subsidiary definitions in *md* will not have associated user defined types.

viii) The right hand side expression may in general be a sequence of guarded simple expressions. In the concrete syntax each of these would be preceded by an = symbol.

ix) A guarded expression is a simple expression followed by a guard. A guard is also a simple expression.

x) We have a variety of infix operators for defining simple expressions. The x and y are simple expressions.

xi) There are also some monadic operators. The x is a simple expression.

xii) Function, or constructor, application is indicated by juxtaposition. The x and the x_i are simple expressions.

xiii) A tuple of n elements. The x_i are simple expressions.

xiv) A list of n elements. The x_i are simple expressions.

xv) A dot-dot expression: it has some simpler forms which we can ignore. The x, y and z are simple expressions.

xvi) A ZF-list: the q_i are qualifiers which govern the values taken by the simple expression e. The generators in the q_i provide an extended environment for e while the filters restrict the values e can take.

xvii) Variables.

xviii) Strings.

xix) Characters.

xx) Numbers.

xxi) A qualifier may be a boolean simple expression which acts as a filter on the values produced by generators to its left.

xxii) The simple expression x should be a list from which the pattern will take successive values.

xxiii) In this form of the generator the pattern first takes the value of the simple expression x and then the successive values of the simple expression y which may be defined in terms of the previous value of the pattern.

The syntax shown does not explicitly cover all forms of a "Miranda" script. However all the missing forms are simplifications of those shown, for instance right hand sides without subsidiary definitions.

3.3 Variable Bindings and Scopes

We shall need the concept of prefixed-sub-expressions when defining what it means for a script to be well typed. The prefixed-sub-expressions can be regarded as capturing the binding and scopes of variables in a "Miranda" script.

We need to keep track of the variables currently bound and in scope for any part of a "Miranda" script. A *variable environment* P is a sequence of *variable binding components* of the following forms:

let v

mutual v	**pattern-formal** v	**law-formal** v
function-formal v	**generator-formal** v	**fix** v
lambda v	**pattern** v	**law** v
where-mutual v	**where** v	**generator** v

The major distinction is between the first form and all the others, as will be explained later. The variety of the latter forms is just to keep track of the different binding cases, even though they will all have the same effect in the well typing algorithm.

The *binding occurrence* of a variable v is its right-most occurrence in the sequence of components. Thus there will be at most one binding occurrence for a variable in a variable environment.

We shall use the symbol $\circ$ to indicate the *extension* of a variable environment by a new binding component, as in $P \circ \textbf{let}\ v$.

We now define a *prefixed-expression* to be a pairing of a variable environment P with an expression e where all the free variables of e are bound in P. We shall write this pairing as $P \mid e$.

All such prefixed-expressions have *prefixed-sub-expressions*. These are usually just pairings of the variable environment P with the sub-expressions of the expression e. For those expressions that declare variables, however, we shall have to extend the variable environment for some of the sub-expressions.

3.4 Types

We now give the syntax for types that we shall be using and relate this to the concrete syntax of "Miranda". We then define various terms which will be used later.

The following is the syntax of *types*.

- We have two *basic types*: num and char for numbers and characters.

  (The type bool is defined as an algebraic type in the standard library)

- We have *type variables*. We shall take $\alpha, \beta \ldots$ as ranging over type variables. These correspond to *, ** etc. in a "Miranda" script.

- We have *function-types*. If ρ and σ are types then $\rho \rightarrow \sigma$ is a type.

- We have *tuple-types*. If $\rho_1, \ldots, \rho_n$ are types then $(\rho_1, \ldots, \rho_n)$ is a type.

- We have *list-types*. If ρ is a type then $[\rho]$ is a type.

- We have *user defined types*. If $\rho_1, \ldots, \rho_n$ are types and t is a *type-name* of arity n then $t\ \rho_1 \cdots \rho_n$ is a type.

  Type-names are defined by algebraic or abstract type declarations, type synonym declarations are not regarded as declaring type names.

  We shall take $\rho, \sigma \ldots$ as ranging over types.

We now introduce various terms which will be used later.

A *type variable substitution* is a map from type variables to types, which we can represent as

$$[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

for types $\tau_1, \ldots, \tau_n$ and distinct type variables $\alpha_1, \ldots, \alpha_n$; it maps α_i to τ_i for $i = 1, \ldots, n$ and leaves all other type variables unchanged. We can extend a type variable

substitution in a natural way to types by applying it to the type variables occurring in the type.

A type substitution $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ *involves* a type variable β if $\beta = \alpha_i$ for some i or if β occurs in some τ_i. We shall use I to represent the empty substitution.

We shall say that a type ρ is an *instance* of a type σ if there is a type substitution S such that $\rho = S\sigma$, we shall write this as $\rho \prec \sigma$.

From Robinson[Rob65] we have the following result about the existence of a *unification* algorithm which can be applied to types.

> There is an algorithm $\mathcal{U}$, taking a pair of types and either failing, or yielding a type substitution, such that for any pair of types σ and τ:
>
> (A) If $\mathcal{U}(\sigma, \tau)$ succeeds yielding U, then U unifies σ and τ, that is $U\sigma = U\tau$.
>
> (B) If R unifies σ and τ, then $\mathcal{U}(\sigma, \tau)$ succeeds yielding a U such that for some substitution S, $R = SU$.
>
> Moreover, U involves only type variables in σ and τ.
>
> Thus $\mathcal{U}$ finds the most general type which is an instance of both σ and τ. We shall call this, following Hindley [Hin69], the *highest common instance* of σ and τ.

As an example of the use of unification of types we have the following:

$$\mathcal{U}(\beta_4 \to [\beta_4] \to [\beta_4], \beta_2 \to \beta_3 \to \beta_5) = [\beta_4/\beta_2, [\beta_4]/\beta_3, [\beta_4]/\beta_5]$$

We also will need to solve the simpler problem of finding a substitution S such that $\rho = S\sigma$ for types ρ and σ with $\rho \prec \sigma$. We shall call this algorithm $\mathcal{I}$ and we will have $\mathcal{I}(\rho, \sigma) = S$.

We shall use the notation $\overline{x}$ to indicate that the object x is *typed*. This will mean different things for different objects, but its meaning will be obvious from the context in which it is used.

For an expression this will mean that we have associated a type with the expression and also associated types with all its sub-expressions.

A typing of a prefixed-expression $P \mid e$ is an assignment of a type to each component of P and all sub-expressions of e; we shall indicate this by $\overline{P} \mid \overline{e}$. $\overline{P}$ can now be regarded as a *type environment* for $\overline{e}$.

For a binding occurrence **let** $v : \sigma$ in a type environment $\overline{P}$ the type variables occurring in σ are called *generic type variables* if they only occur in enclosing binding components of the **let** form. Those type variables in σ that occur in enclosing non-**let** components are *non-generic type variables* as are all the type variables occurring in the types in any non-**let** component. As we shall see later, these generic type variables capture the polymorphism in a "Miranda" script as only they may be substituted for.

Given a type environment $\overline{P}$, a *generic instance* of a type σ is an instance of σ where all the non-generic type variables are left unchanged. We shall write $\rho \prec\!\!\prec \sigma$ to indicate that ρ is a generic instance of σ.

We shall call $\overline{P} \mid \overline{e}$ *standard* if and only if for every typed prefixed-sub-expression $\overline{P'} \mid \overline{e'}$, with the natural induced typing, for any binding component **let** $x : \sigma$ in $\overline{P'}$ the generic type variables in σ occur nowhere else in $\overline{P'} \mid \overline{e'}$. The typed prefixed-sub-expressions of $\overline{P} \mid \overline{e}$ are simply the sub-expressions of $\overline{e}$ paired with a suitably extended variable environment. The types assigned to the additional variables in this environment are induced by the types given to these variables in $\overline{e}$.

3.5 Well Typing

We now define what we mean when we say a script has a well typing. This is done in terms of certain type conditions on the prefixed-sub-expressions.

A typed prefixed-expression, $\overline{P} \mid \overline{e}$, is *well-typed* if:

A) It is standard.

B) For every bound occurrence $x : \sigma$ in $\overline{e}$, the binding occurrence of x in $\overline{P}$ is either **let** $x : \rho$, where $\sigma \prec\!\!\prec \rho$, or some other component like **lambda** $x : \rho$ where $\sigma = \rho$.

C) Certain conditions on the typing of the prefixed-sub-expressions under the induced typing are met by all the sub-expressions of e.

The full set of these conditions for all the productions is given in [Lon87b]. We now present a selection of the interesting cases. We will use parentheses $(\cdots)$ to make clear which parts of an expression the types apply to. The context will allow us to distinguish this use of (and) from their use in tuples.

i) $\overline{P} \mid \overline{md_1} \cdots \overline{md_n}$

No conditions.

ii) $\overline{P} \mid \overline{d_1} : \rho_1 \ \tau_1 \cdots \overline{d_n} : \rho_n \ \tau_n$

We require that the user given type definitions τ_i be the same as the types ρ_i, $\rho_i = \tau_i$ for $i = 1, \ldots, n$.

iii) $\overline{P} \mid \overline{c_1} : \sigma_1 \cdots \overline{c_n} : \sigma_n$

We require that all the clauses of a function definition have the same type, $\sigma_i = \sigma_1$ for $i = 2, \cdots, n$.

vi) $\overline{P} \mid (\overline{g} : \sigma_1 \ \overline{pat_1} : \rho_1 \cdots \overline{pat_n} : \rho_n = \overline{e} : \sigma_2) : \tau$

We first require that the type assigned to the function being declared, g, be the same as that assigned to the entire expression, $\tau = \sigma_1$. We also require that the types assigned to the formal parameter patterns pat_i and the right hand side expression e correspond to the type of g, $\sigma_1 = \rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \sigma_2$.

vii) $\overline{P} \mid (\overline{rhsl} : \sigma \text{ where } \overline{md}) : \rho$

The type of a right hand side with subsidiary definitions should be the type of the expression list, $\sigma = \rho$.

viii) $\overline{P} \mid (\overline{ge_1} : \sigma_1 \cdots \overline{ge_n} : \sigma_n) : \rho$

We require that all the guarded expressions have the same type, $\sigma_i = \sigma_1$ for $i = 2, \ldots, n$, and that the type of the list of guarded expressions also be this same type, $\rho = \sigma_1$.

ix) $\overline{P} \mid (\overline{e} : \sigma, \overline{grd} : \rho) : \tau$

The guard must be of boolean type, $\rho =$ bool, and the type of a guarded expression should be the type of the expression, $\tau = \sigma$.

xii) $\overline{P} \mid (\overline{x} : \sigma\ \overline{x_1} : \rho_1 \cdots \overline{x_n} : \rho_n) : \tau$

We require that the types of the arguments and of the entire expression correspond to the type of the function being applied, $\sigma = \rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \tau$.

xiii) $\overline{P} \mid (\overline{x_1} : \sigma_1, \ldots, \overline{x_n} : \sigma_n) : \rho$

The type of the tuple should be the tuple-type of the types of its elements, $\rho = (\sigma_1, \ldots, \sigma_n)$.

xvii) $\overline{P} \mid \overline{var} : \sigma$

If *var* is bound in $\overline{P}$ by **let** *var* : ρ then we require that σ be a generic instance of ρ, $\sigma \prec\!\!\prec \rho$. If *var* is bound by any other form of component, such as **lambda** *var* :

ρ, we then require that its type be that of its binding occurrence in $\overline{P}$, $\rho = \sigma$.

3.6 Well Typing Algorithm $\mathcal{M}$

We present a recursive algorithm $\mathcal{M}$ which finds a well typing for a "Miranda" script if it has one. If the script does not have a well typing, one of the attempted unifications will fail.

3.6.1 The Basic Algorithm

At all times we have a type environment $\overline{P}$. Given this type environment and an expression f, we return the typed expression $\overline{f}$ and also the type variable substitution produced by the well typing of f. The initial type environment includes all the "Miranda" operators, all those variables declared by the standard library and also all the constructors declared by algebraic type definitions in the standard library and the script.

The final type deduced for an expression is affected by the entire script. The algorithm is, however, sequential and the preceding and succeeding portions of of the script affect the deduced type in different ways, see Figure 2. The type checking of an expression produces two things: the first is a modified type environment and the second is a substitution. The type environment is modified by the constraints on the types of the variables in it imposed by their occurrences in the expression. The

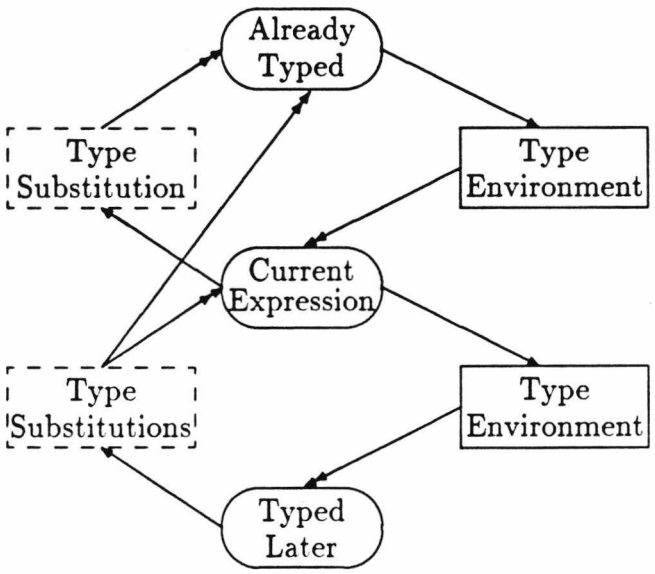

Figure 2: Sequential Nature of $\mathcal{M}$

substitution is produced by the unifications of the types of the occurrences of variables in the expression with their previous occurrences. The modified type environment affects the type checking of the succeeding portion of the script while the substitution is applied to the preceding, already type checked, portion of the script, thus changing its deduced types.

We proceed through the script in a sequential fashion; at each declaration of a variable we add it to the type environment with the most general possible type. Then as we come across occurrences of the variable in the script these will impose restrictions on the type of the variable.

We make sure we get the most general type by unifying the already deduced type for the variable, to be found in the type environment, with that of its current

occurrence in the script.

In the following we shall refer to *modified type environments*, which are produced by applying type substitutions to a type environment, and to *extended type environments*, which are produced by adding new binding components to a type environment and possibly applying type substitutions.

The full description of the well typing algorithm is given in [Lon87b]; here we shall only present a selection of the more interesting cases.

The algorithm is defined recursively by cases on the expression f. That is,

$$\mathcal{M}(\overline{P}, f) = (T, \overline{f})$$

where T and $\overline{f}$ are computed as follows:

i) $f \equiv md_1 \cdots md_n$

This set of groups of mutually recursive definitions is topologically sorted so for $i = 1, \ldots, n$ no variables declared in md_i depend on any variables declared in md_j, $j > i$. Also, as all the variables declared by md_i take generic instances of their types when occurring in md_j, the type substitutions produced by the typing of md_j cannot affect the typing of md_i. We type each of the md_i in sequence and build up the type environment as we go. We first define $\overline{P_0} = \overline{P}$, then for $i = 1, \ldots, n$:

Let $v_1^i, \ldots, v_{k_i}^i$ be the variables declared by md_i. We then have the

extended type environment $\overline{P_i'}$,

$$\overline{P_i'} = \overline{P_{i-1}} \circ \textbf{mutual } v_1^i : \beta_1^i \circ \cdots \circ \textbf{mutual } v_{k_i}^i : \beta_{k_i}^i$$

where $\beta_1^i, \ldots, \beta_{k_i}^i$ are new type variables. We can now type the group of mutually recursive definitions md_i.

$$(R_i, \overline{md_i}) = \mathcal{M}(\overline{P_i'}, md_i)$$

This set of definitions then provides the extended type environment $\overline{P_i}$ for the rest of the script. That is, for md_j, $j > i$:

$$\overline{P_i} = \overline{P_{i-1}} \circ \textbf{let } v_1^i : R_i\beta_1^i \circ \cdots \circ \textbf{let } v_{k_i}^i : R_i\beta_{k_i}^i$$

We finally get:

$$T \equiv I$$

$$\overline{f} \equiv \overline{md_1} \cdots \overline{md_n}$$

ii) $f \equiv d_1\ \tau_1 \cdots d_n\ \tau_n$

We type each of the d_i in sequence, applying the type variable substitutions produced for each d_i to the type environment $\overline{P}$. All the variables declared by the d_i will be bound in $\overline{P}$ by **mutual** components. We first define $\overline{P_0} = \overline{P}$, then for $i = 1, \ldots, n$:

We type the definition d_i:

$$(R_i, \overline{d_i} : \rho_i) = \mathcal{M}(\overline{P_{i-1}}, d_i)$$

The user defined type τ_i should be an instance of the deduced type ρ_i:

$$S_i = \mathcal{I}(\rho_i, \tau_i)$$

We then apply the type substitutions produced to get the modified type environment:

$$\overline{P_i} = S_i R_i \overline{P_{i-1}}$$

We finally get:

$$T \equiv S_n R_n \cdots S_1 R_1$$

$$\overline{f} \equiv (S_n R_n \cdots S_2 R_2 S_1 \overline{d_1}) \cdots (S_n \overline{d_n})$$

iii) $f \equiv c_1 \cdots c_n$

We type each of the c_i in sequence applying the type variable substitutions produced for each c_i to the type environment $\overline{P}$. We also check that the types of all the clauses unify. We first type the clause c_1:

$$(R_1, \overline{c_1} : \rho_1) = \mathcal{M}(\overline{P}, c_1)$$

We define U_1:

$$U_1 = I$$

and we apply the type substitution produced to get the modified type environment $\overline{P_1}$:

$$\overline{P_1} = R_1\overline{P}$$

Then for $i = 2, \ldots, n$:

We type the clause c_i.

$$(R_i, \overline{c_i} : \rho_i) = \mathcal{M}(\overline{P_{i-1}}, c_i)$$

Then we check that the type of this clause unifies with that of the previous clause:

$$U_i = \mathcal{U}(R_i U_{i-1}\rho_{i-1}, \rho_i)$$

We then apply the type substitutions produced to get the modified type environment:

$$\overline{P_i} = U_i R_i \overline{P_{i-1}}$$

We also check that the type assigned to the function g, defined by the c_i, is consistent with any conditions imposed on the type of g by its prior occurrence in the set of mutually recursive definitions containing this function definition.

Let **mutual** $g : \sigma$ or **where-mutual** $g : \sigma$ be the binding occurrence of g in $\overline{P}$. Then the type we have deduced for g should unify with this type σ:

$$U = \mathcal{U}(\sigma, U_n\rho_n)$$

We finally get:

$$T \equiv UU_n R_n \cdots R_1$$

$$\overline{f} \equiv (UU_n R_n \cdots R_2 \overline{c_1}) \cdots (UU_n \overline{c_n}) : U\sigma$$

vi) $f \equiv g\ pat_1 \cdots pat_n = rhs$

Let $v_1, \ldots, v_k$ be the free variables in the formal parameter patterns pat_i. We then have the extended type environment $\overline{P_0}$:

$$\overline{P_0} = \overline{P} \circ \textbf{function-formal}\ v_1 : \beta_1 \circ \cdots \circ \textbf{function-formal}\ v_k : \beta_k$$

where $\beta_1, \ldots, \beta_k$ are new type variables. We first type the pat_i in sequence in the type environment $\overline{P_0}$, applying the type variable substitutions produced by the typing of each pat_i to $\overline{P_0}$. That is, for $i = 1, \ldots, n$:

We type the pattern pat_i.

$$(R_i, \overline{pat_i} : \rho_i) = \mathcal{M}(\overline{P_{i-1}}, pat_i)$$

and then apply the type variable substitution produced to get the modified type environment:

$$\overline{P_i} = R_i \overline{P_{i-1}}$$

Having typed the formal parameter patterns, and thus having deduced types for the variables declared by them, we now have the extended type environment

$\overline{P'}$. This has two possible forms, depending on whether g is free in *rhs*. We first have a new type variable β. Then, if g appears free in *rhs*:

$$\begin{aligned}\overline{P'} \;=\; & \overline{P} \circ \textbf{fix}\ g : \beta \circ \\ & \textbf{lambda}\ v_1 : R_n \cdots R_1 \beta_1 \circ \cdots \circ \textbf{lambda}\ v_k : R_n \cdots R_1 \beta_k\end{aligned}$$

Otherwise, we have:

$$\begin{aligned}\overline{P'} \;=\; & \overline{P} \circ \\ & \textbf{lambda}\ v_1 : R_n \cdots R_1 \beta_1 \circ \cdots \circ \textbf{lambda}\ v_k : R_n \cdots R_1 \beta_k\end{aligned}$$

We now type the right hand side expression in this extended type environment:

$$(S, \overline{rhs} : \sigma) = \mathcal{M}(\overline{P'}, rhs)$$

We now check that the type deduced for the occurrences, if any, of g in *rhs* unifies with the type given to g by this definition. Note that if g is not free in *rhs* then the type substitution S will not involve β.

$$U = \mathcal{U}(SR_n \cdots R_2 \rho_1 \rightarrow \cdots \rightarrow S\rho_n \rightarrow \sigma, S\beta)$$

We finally get:

$$T \equiv US$$

$$\overline{f} \equiv (US\overline{g})\ (USR_n \cdots R_2 \overline{pat_1}) \cdots (US\overline{pat_n}) = (U\overline{rhs}) : US\beta$$

vii) $f \equiv rhsl$ where md

Let $v_1, \ldots, v_k$ be the variables declared by md. We then have the extended type environment $\overline{P'}$:

$$\overline{P'} = \overline{P} \circ \textbf{where-mutual } v_1 : \beta_1 \circ \cdots \circ \textbf{where-mutual } v_k : \beta_k$$

where $\beta_1, \ldots, \beta_k$ are new type variables. We now type the subsidiary definitions in this extended type environment:

$$(R, \overline{md}) = \mathcal{M}(\overline{P'}, md)$$

Having typed the subsidiary definitions, and thus having deduced types for the variables declared by them, we now have the extended type environment $\overline{P''}$:

$$\overline{P''} = R\overline{P} \circ \textbf{where } v_1 : R\beta_1 \circ \cdots \circ \textbf{where } v_k : R\beta_k$$

We now type the right hand side list of expressions in this extended type environment:

$$(S, \overline{rhsl} : \rho) = \mathcal{M}(\overline{P''}, rhsl)$$

We finally get:

$$T \equiv SR$$

$$\overline{f} \equiv \overline{rhsl} \text{ where } S\overline{md} : \rho$$

viii) $f \equiv ge_1 \cdots ge_n$

We type each of the guarded expressions ge_i in sequence applying the type variable substitutions produced for each ge_i to the type environment $\overline{P}$. We also check that the types of all the guarded expressions unify.

We first type the guarded expression ge_1:

$$(R_1, \overline{ge_1} : \rho_1) = \mathcal{M}(\overline{P}, ge_1)$$

We define U_1:

$$U_1 = I$$

and we apply the type substitution produced to get the modified type environment $\overline{P_1}$:

$$\overline{P_1} = R_1\overline{P}$$

Then for $i = 2, \ldots, n$:

We type the guarded expression ge_i:

$$(R_i, \overline{ge_i} : \rho_i) = \mathcal{M}(\overline{P_{i-1}}, ge_i)$$

We then check that the type of this guarded expression unifies with that of the previous guarded expression:

$$U_i = \mathcal{U}(R_i U_{i-1} \rho_{i-1}, \rho_i)$$

We then apply the type substitutions produced to get the modified type environment:

$$\overline{P_i} = U_i R_i \overline{P_{i-1}}$$

We finally get:

$$T \equiv U_n R_n \cdots R_1$$

$$\overline{f} \equiv (U_n R_n \cdots U_2 R_2 \overline{ge_1}) \cdots (U_n \overline{ge_n}) : U_n \rho_n$$

ix) $f \equiv e, grd$

We first type the guard:

$$(R, \overline{grd} : \rho) = \mathcal{M}(\overline{P}, grd)$$

We then check that its type unifies with the boolean type:

$$U = \mathcal{U}(\rho, \text{bool})$$

We then type the expression.

$$(S, \overline{e} : \sigma) = \mathcal{M}(UR\overline{P}, e)$$

We finally get:

$$T \equiv SUR$$

$$\overline{f} \equiv \overline{e}, (SU\overline{grd}) : \sigma$$

xii) $f \equiv x\ x_1 \cdots x_n$

We type the expression x and the expressions x_i in sequence applying the type substitutions produced to the type environment $\overline{P}$:

$$(R, \overline{x} : \rho) = \mathcal{M}(\overline{P}, x)$$

We define the modified type environment $\overline{P_0}$:

$$\overline{P_0} = R\overline{P}$$

Then for $i = 1, \ldots, n$:

We type the expression x_i:

$$(R_i, \overline{x_i} : \rho_i) = \mathcal{M}(\overline{P_{i-1}}, x_i)$$

We apply the type substitution produced to get the modified type environment:

$$\overline{P_i} = R_i\overline{P_{i-1}}$$

The type of the expression x should correspond to the types of the expressions it is being applied to:

$$U = \mathcal{U}(R_n \cdots R_1\rho, R_n \cdots R_2\rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \beta)$$

where β is a new type variable. We finally get:

$$T \equiv UR_n \cdots R_1 R$$

$$\overline{f} \equiv (UR_n \cdots R_1 \overline{x})\ (UR_n \cdots R_2 \overline{x_1}) \cdots (U\overline{x_n}) : U\beta$$

xiii) $f \equiv (x_1, \ldots, x_n)$

We type each of the expressions x_i in sequence applying the type substitutions produced to the type environment $\overline{P}$:

We define:

$$\overline{P_0} = \overline{P}$$

Then for $i = 1, \ldots, n$:

We type the expression x_i:

$$(R_i, \overline{x_i} : \rho_i) = \mathcal{M}(\overline{P_{i-1}}, x_i)$$

We then apply the type substitution produced to get the modified type environment:

$$\overline{P_i} = R_i \overline{P_{i-1}}$$

The type of the tuple is then the tuple-type of the types of its components. We finally get:

$$T \equiv R_n \cdots R_1$$

$$\overline{f} \equiv (R_n \cdots R_2 \overline{x_1}, \ldots, \overline{x_n}) : (R_n \cdots R_2 \rho_1, \ldots, \rho_n)$$

xvii) $f \equiv var$

There are two separate cases: If *var* is bound by **let** $var : \sigma$ in $\overline{P}$, then:

Let τ be a generic instance of σ, that is $\tau \prec\!\!\prec \sigma$. We get:

$$T \equiv I$$

$$\overline{f} \equiv \overline{var} : \tau$$

If *var* is bound by something else, like **lambda** $var : \sigma$,

We get:

$$T \equiv I$$

$$\overline{f} \equiv \overline{var} : \sigma$$

3.6.2 Type Synonyms and Abstract Types

The algorithm $\mathcal{M}$ does not mention the type information presented in the script by the abstract and type synonym definitions.

As noted previously we can simply regard the type synonym definitions as rewrite rules for the types. We add an initial step to the unification algorithm to do this rewriting. Suppose we have the following two type synonym definitions:

```
list1 * == [*]
list2 * == [*]
```

Then if we are required to unify these two types:

```
list1 num
list2 num
```

we first apply the type synonyms as rewrite rules to get the two types:

```
[num]
[num]
```

which we then attempt to unify.

Abstract type definitions do not initially appear to fit easily into the framework provided by algorithm $\mathcal{M}$. However, when we add the above idea for dealing with type synonyms, we can achieve an elegant solution.

We can associate with each implementation equation the type definitions and the correspondence between the abstract type and its implementation type. We now simply extend algorithm $\mathcal{M}$ to take an extra argument, a list of type rewrite rules. Initially this will consist only of the type synonym definitions, but when we are typing an implementation equation we add the type correspondence for the abstract type to the list of type rewrite rules.

When typing the script, variables defined in an abstract type definition are treated in the same manner as top level definitions. That is they take arbitrary instances of their defined polymorphic types. The type correspondence between the abstract type and its implementation type is hidden in the rest of the script as it is not present in the list of type rewrite rules. When typing the implementation equations the correspondence is in the list of rewrite rules so it is available as we require.

3.6.3 Type Definitions

The well typing algorithm as described simply takes the user-provided type definitions as extra type constraints on the top-level definitions. We can, however, use the type definitions to produce better type error messages and also to increase the efficiency of the well typing algorithm.

If we assume that the user-provided type definition represents the type the user expects a top-level definition to take then we can use this information to simplify the well typing task. Recall that we must apply the well typing algorithm to each mutually recursive group of definitions unifying all occurrences of the variables defined by all the definitions. This is because we take mutually recursive definitions to be mutually type dependent. If, however, a definition has a user-provided type definition then no other definition need be dependent on its well typing having been found as it can use the user-provided type. In this manner we can reduce the size of the groups of mutually type dependent definitions and hence the complexities of type unifications involved.

We obtain better type error messages as each occurrence of a variable with a user-provided type definition need merely be an instance of this type and hence error messages related to the user-provided type are given. Without this addition to the algorithm the type deduced for a variable from its occurrences before its definition can be very different from that intended, giving rise to very obscure type errors.

3.6.4 Example

We now present an example of the application of this algorithm to a simple definition.

```
reverse (x:l) = reverse l ++ [x]
reverse [] = []
```

We shall assume that the initial type environment $\overline{P_0}$ is the minimal one required to type this function definition.

$$\overline{P_0} \equiv \textbf{let} : \underline{\alpha_1 \to [\alpha_1] \to [\alpha_1]} \circ \textbf{let} \; \texttt{++} \; \underline{[\alpha_2] \to [\alpha_2] \to [\alpha_2]} \circ \textbf{let} \; [] \; \underline{[\alpha_3]}$$

Notice that we shall use underlining to distinguish types from the surrounding "Miranda" text; we shall also add parentheses as required to avoid confusion as to which part of a "Miranda" expression a type refers to. We will also leave out the : preceding types as this would be confused with the : "Miranda" operator. We shall also leave out the overlining of the typed expressions to keep the presentation simple. We shall refer to the current arguments of the recursive call of $\mathcal{M}$ we are dealing with as $\overline{P}$ and f.

We initially have $\overline{P} \equiv \overline{P_0}$, while f is the definition of `reverse` above. We proceed through the script, typing the mutually recursive groups of definitions. In this case we only have one such group containing a single definition. Thus we extend the type environment $\overline{P}$ by adding a binding component for `reverse` which is the only variable declared by this mutually recursive group. Thus, as $\overline{P} \equiv \overline{P_0}$ we have the following

extended type environment.

$\overline{P_0}$ ∘ **mutual reverse** $\underline{\beta_1}$

Then we proceed to type the definition in this extended type environment. This is a multiple clause definition so we will have to type both clauses and then check that their types unify.

i) For the first clause of the definition we then have:

$\overline{P} \equiv \overline{P_0}$ ∘ **mutual reverse** $\underline{\beta_1}$

$f \equiv$ `reverse (x:l) = reverse l ++ [x]`

Now we must first type the formal parameter pattern and then the right hand side expression.

(a) For the formal parameter pattern we extend $\overline{P}$ by the variables in the formal parameter pattern. So we have:

$\overline{P} \equiv \overline{P_0}$ ∘ **mutual reverse** $\underline{\beta_1}$ ∘ **function-formal x** $\underline{\beta_2}$ ∘ **function-formal l** $\underline{\beta_3}$

$f \equiv$ `(x:l)`

This is an application of : to the arguments `x` and `l`. We must first type these two arguments.

i. For the first argument we have:

$$\overline{P} \equiv \overline{P_0} \circ \textbf{let reverse } \underline{\beta_1} \quad \circ \quad \textbf{function-formal x } \underline{\beta_2} \circ \textbf{function-formal 1 } \underline{\beta_3}$$

$$f \equiv \mathbf{x}$$

As **x** is bound by a **function-formal** component in the type environment we take its type from the type environment, that is $\underline{\beta_2}$. There are no type substitutions produced and we have:

$$\overline{f} \equiv \mathbf{x}\ \underline{\beta_2}$$

ii. For the second argument we have:

$$\overline{P} \equiv \overline{P_0} \circ \textbf{let reverse } \underline{\beta_1} \quad \circ \quad \textbf{function-formal x } \underline{\beta_2} \circ \textbf{function-formal 1 } \underline{\beta_3}$$

$$f \equiv \mathtt{1}$$

As 1 is bound by a **function-formal** component in the type environment we take its type from the type environment, that is $\underline{\beta_3}$. There are no type substitutions produced and we have:

$$\overline{f} \equiv \mathtt{1}\ \underline{\beta_3}$$

Now we need the type of the : operator. As it is bound by a **let** component in the type environment we must take a generic instance of its type in the type environment.

$$\underline{\beta_4 \rightarrow [\beta_4] \rightarrow [\beta_4]}$$

We now require that the type of : unify with the types of its arguments x and 1. We have the following application of the unification algorithm.

$$\mathcal{U}(\underline{\beta_4 \rightarrow [\beta_4] \rightarrow [\beta_4]}, \underline{\beta_2 \rightarrow \beta_3 \rightarrow \beta_5}) = [\underline{\beta_4}/\underline{\beta_2}, \underline{[\beta_4]}/\underline{\beta_3}, \underline{[\beta_4]}/\underline{\beta_5}]$$

Notice that we shall choose our unifiers carefully to make the presentation as clear as possible.

Thus we have:

$$T \equiv [\underline{\beta_4}/\underline{\beta_2}, \underline{[\beta_4]}/\underline{\beta_3}, \underline{[\beta_4]}/\underline{\beta_5}]$$

Now applying this substitution to the types we have already deduced for the sub-expressions of `(x:1)` we get:

$$\overline{f} \equiv (\mathtt{x}\ \underline{\beta_4}\ :\ \underline{\beta_4 \rightarrow [\beta_4] \rightarrow [\beta_4]}\ \mathtt{1}\ \underline{[\beta_4]})\ \underline{[\beta_4]}$$

(b) For the right hand side expression of the first clause we have the following extended type environment. This is obtained by applying the type substitution produced by the typing of the formal parameter pattern to the types initially assigned to the variables appearing in the formal parameter pattern, x and 1.

As the function **reverse** being defined occurs in the right hand side expression it must also be added to the extended type environment.

$$\overline{P} \equiv \overline{P_0}\ \circ\ \textbf{mutual reverse}\ \underline{\beta_1} \circ \textbf{fix reverse}\ \underline{\beta_6} \circ \textbf{lambda x}\ \underline{\beta_4} \circ \textbf{lambda 1}\ \underline{[\beta_4]}$$

$$f \equiv \mathtt{reverse\ 1\ ++\ [x]}$$

This is an application of the ++ operator to the two arguments `reverse l` and `[x]`. So we must first type these two argument expressions.

i. For the first argument we have:

$$\overline{P} \equiv \overline{P_0} \circ \textbf{mutual reverse } \underline{\beta_1} \circ \textbf{fix reverse } \underline{\beta_6} \circ \textbf{lambda x } \underline{\beta_4} \circ \textbf{lambda l } \underline{[\beta_4]}$$

$f \equiv$ `reverse l`

This is an application of `reverse` to `l`. We must first type these two expressions.

A. For the first argument we have:

$$\overline{P} \equiv \overline{P_0} \circ \textbf{mutual reverse } \underline{\beta_1} \circ \textbf{fix reverse } \underline{\beta_6} \circ \textbf{lambda x } \underline{\beta_4} \circ \textbf{lambda l } \underline{[\beta_4]}$$

$f \equiv$ `reverse`

As `reverse` is bound by a **fix** component in the type environment we take its type from the type environment, that is $\underline{\beta_6}$. There are no type substitutions produced and we have:

$$\overline{f} \equiv \textbf{reverse } \underline{\beta_6}$$

B. For the second argument we have:

$$\overline{P} \equiv \overline{P_0} \circ \textbf{mutual reverse } \underline{\beta_1} \circ \textbf{fix reverse } \underline{\beta_6} \circ \textbf{lambda x } \underline{\beta_4} \circ \textbf{lambda l } \underline{[\beta_4]}$$

$f \equiv$ `l`

As `l` is bound by a **lambda** component in the type environment we take its type from the type environment, that is $\underline{[\beta_4]}$. There are no type substitutions produced and we have:

$$\overline{f} \equiv \texttt{l}\ \underline{[\beta_4]}$$

We must now unify the type deduced for **reverse** with that of its argument `l`. We have the following unification:

$$\mathcal{U}(\underline{\beta_6}, \underline{[\beta_4] \rightarrow \beta_7}) = [\underline{[\beta_4] \rightarrow \beta_7}/\underline{\beta_6}]$$

Thus we have:

$$T \equiv [\underline{[\beta_4] \rightarrow \beta_7}/\underline{\beta_6}]$$

Now applying this substitution to the types we have already deduced for the sub-expressions of **reverse** `l` we get:

$$\overline{f} \equiv (\textbf{reverse}\ \underline{[\beta_4] \rightarrow \beta_7}\ \texttt{l}\ \underline{[\beta_4]})\ \underline{\beta_7}$$

ii. For the second argument we have:

$$\begin{aligned}\overline{P} \equiv \overline{P_0}\ \circ\ & \textbf{mutual reverse}\ \underline{\beta_1} \circ \\ & \textbf{fix reverse}\ \underline{[\beta_4] \rightarrow \beta_7} \circ \\ & \textbf{lambda x}\ \underline{\beta_4} \circ \\ & \textbf{lambda l}\ \underline{[\beta_4]}\end{aligned}$$

$$f \equiv [\mathbf{x}]$$

Notice that we have imposed restrictions on the type of **reverse** in the type environment from its occurrence in the first argument.

As **x** is bound by a **lambda** component in the type environment we take its type from the type environment, that is $\underline{\beta_4}$. There are no type

substitutions produced and we have:

$$\overline{f} \equiv [\mathtt{x}\ \underline{\beta_4}]\ \underline{[\beta_4]}$$

Now we need the type of the **++** operator. As it is bound by a **let** component in the type environment we must take a generic instance of its type in the type environment.

$$\underline{[\beta_8] \rightarrow [\beta_8] \rightarrow [\beta_8]}$$

We now must unify the type of the operator **++** with the types deduced for its two operands **reverse l** and **[]**. We have the following application of the unification algorithm.

$$\begin{aligned}&\mathcal{U}(\underline{[\beta_8] \rightarrow [\beta_8] \rightarrow [\beta_8]}, \underline{\beta_7 \rightarrow [\beta_4] \rightarrow \beta_9})\\ &\quad = [\underline{\beta_4}/\underline{\beta_8}, \underline{[\beta_4]}/\underline{\beta_7}, \underline{[\beta_4]}/\underline{\beta_9}]\end{aligned}$$

Thus we have:

$$T \equiv [\underline{[\beta_4] \rightarrow [\beta_4]}/\underline{\beta_6}, \underline{\beta_4}/\underline{\beta_8}, \underline{[\beta_4]}/\underline{\beta_7}, \underline{[\beta_4]}/\underline{\beta_9}]$$

Now applying this substitution to the types we have already deduced for the sub-expressions of **reverse l ++ [x]** we get:

$$\begin{aligned}\overline{f} \equiv\ &(\mathtt{(reverse}\ \underline{[\beta_4] \rightarrow [\beta_4]}\ \mathtt{l}\ \underline{[\beta_4]})\ \underline{[\beta_4]}\\ &\mathtt{++}\ \underline{[\beta_4] \rightarrow [\beta_4] \rightarrow [\beta_4]}\\ &[\mathtt{x}\ \underline{\beta_4}]\ \underline{[\beta_4]})\ \underline{[\beta_4]}\end{aligned}$$

We must now check that the type deduced for **reverse** from the deduced types of its formal parameter pattern and its right hand side expression corresponds

with the type deduced for its use on the right hand side.

By applying the substitution T produced by the typing of the right hand side expression to the type initially assigned to **reverse** in its **fix** component in the type environment we get:

$$\underline{[\beta_4] \rightarrow [\beta_4]}$$

Applying the substitution T to the already typed formal parameter pattern has no effect, so it has type:

$$\underline{[\beta_4]}$$

As does the right hand side expression. Thus we have the following unification:

$$\mathcal{U}(\underline{[\beta_4] \rightarrow [\beta_4]}, \underline{[\beta_4] \rightarrow [\beta_4]})$$

Which succeeds with the empty substitution.

Thus we have:

$$T \equiv [\underline{[\beta_4] \rightarrow [\beta_4]}/\underline{\beta_6}, \underline{\beta_4}/\underline{\beta_8}, \underline{[\beta_4]}/\underline{\beta_7}, \underline{[\beta_4]}/\underline{\beta_9}]$$

And we also have:

$$\begin{aligned}
\overline{f} \equiv\ & \mathbf{reverse}\ \underline{[\beta_4] \rightarrow [\beta_4]}\ (\mathtt{x}\ \underline{\beta_4} : \underline{\beta_4 \rightarrow [\beta_4] \rightarrow [\beta_4]}\ \mathtt{1}\ \underline{[\beta_4]})\ \underline{[\beta_4]} \\
& = \\
& ((\mathbf{reverse}\ \underline{[\beta_4] \rightarrow [\beta_4]}\ \mathtt{1}\ \underline{[\beta_4]})\ \underline{[\beta_4]} \\
& \mathbf{++}\ \underline{[\beta_4] \rightarrow [\beta_4] \rightarrow [\beta_4]} \\
& [\mathtt{x}\ \underline{\beta_4}]\ \underline{[\beta_4]})\ \underline{[\beta_4]}
\end{aligned}$$

ii) For the second clause of the definition we then have:

$$\overline{P} \equiv \overline{P_0} \circ \textbf{mutual reverse } \underline{\beta_1}$$

$$f \equiv \texttt{reverse [] = []}$$

Now we must first type the formal parameter pattern and then the right hand side expression.

(a) For the formal parameter pattern we extend $\overline{P}$ by the variables in the formal parameter pattern. In this case there are none so we have:

$$\overline{P} \equiv \overline{P_0} \circ \textbf{mutual reverse } \underline{\beta_1}$$

$$f \equiv []$$

We need the type of []. As it is bound by a **let** component in the type environment we must take a generic instance of its type in the type environment:

$$\underline{[\beta_{10}]}$$

There are no type substitutions produced and we have:

$$\overline{f} \equiv [] \ \underline{[\beta_{10}]}$$

(b) For the right hand side expression of the second clause we have the following extended type environment, in this case it is simple as there are no variables in the formal parameter pattern.

As the function **reverse** being defined does not occur in the right hand side expression it needn't be added to the extended type environment.

$$\overline{P} \equiv \overline{P_0} \circ \textbf{mutual reverse } \underline{\beta_1}$$

We also have:

$$f \equiv []$$

We need the type of []. As it is bound by a **let** component in the type environment we must take a generic instance of its type in the type environment:

$$\underline{[\beta_{11}]}$$

There are no type substitutions produced and we have:

$$\overline{f} \equiv [] \ \underline{[\beta_{11}]}$$

Now we have the following unification to determine the type to be deduced for this clause.

$$\mathcal{U}(\underline{[\beta_{10}] \rightarrow [\beta_{11}]}, \underline{\beta_{12}}) = [\underline{[\beta_{10}] \rightarrow [\beta_{11}]}/\underline{\beta_{12}}]$$

Thus we have:

$$T \equiv [\underline{[\beta_{10}] \rightarrow [\beta_{11}]}/\underline{\beta_{12}}]$$

And we also have:

$$\overline{f} \equiv \texttt{reverse}\ \underline{[\beta_{10}] \rightarrow [\beta_{11}]}\ []\ \underline{[\beta_{10}]} = []\ \underline{[\beta_{11}]}$$

We must now check that the types deduced for the two clauses unify. We have the following unification.

$$\mathcal{U}(\underline{[\beta_4] \rightarrow [\beta_4]}, \underline{[\beta_{10}] \rightarrow [\beta_{11}]}) = [\underline{\beta_4}/\underline{\beta_{10}}, \underline{\beta_4}/\underline{\beta_{11}}]$$

Applying this substitution to the first clause has no affect but for the second clause we now have:

$$\overline{f} \equiv \texttt{reverse}\ \underline{[\beta_4] \rightarrow [\beta_4]}\ []\ \underline{[\beta_4]} = []\ \underline{[\beta_4]}$$

We have thus deduced the type of the definition of the **reverse** function as being:

$$\underline{[\beta_4] \rightarrow [\beta_4]}$$

We must now check that this corresponds with the type deduced for **reverse** by its occurrence elsewhere in this group of mutually recursive definitions. To do this we must be able to unify this deduced type with that associated with **reverse** by its **mutual** component in $\overline{P}$. We have the following successful unification.

$$\mathcal{U}(\underline{\beta_1}, \underline{[\beta_4] \rightarrow [\beta_4]}) = [\underline{[\beta_4] \rightarrow [\beta_4]}/\underline{\beta_1}]$$

We have now finished typing the definition of **reverse**. We now have the following extended type environment for typing the rest of the script.

$$\overline{P_0} \circ \textbf{let}\ \texttt{reverse}\ \underline{[\beta_4] \rightarrow [\beta_4]}$$

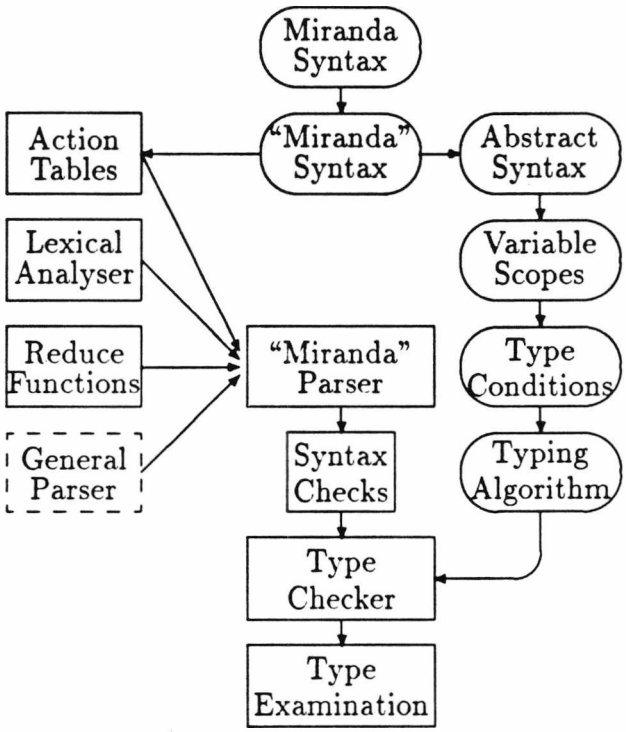

Figure 3: The Typechecker

All occurrences of **reverse** in the rest of the script will take generic instances of this type.

3.7 Implementation

The well typing algorithm $\mathcal{M}$ was used as the basis for the implementation of a type checker and interactive type examiner for "Miranda" written in Miranda. The overall structure of the work leading to this implementation and its dependencies is shown in Figure 3. The boxed entries in the figure represent differing sections of the actual

implementation in Miranda while the other entries represent the theoretical work on which the implementation is based. The details of the use of the system are presented in Appendix B.

As we wished to produce a complete system which would demonstrate the implementation of the type checking algorithm we needed a "Miranda" parser and an interactive user interface. Neither of these components was our main concern and so, beyond their existence, we did not devote a great deal of attention to them.

3.7.1 Parser

It can be seen that the first piece of work on which everything else depended was the derivation of a syntax for "Miranda". This was used as the basis of the abstract syntax and hence the derivation of the well typing algorithm $\mathcal{M}$. The "Miranda" syntax was also used, as described in [Lon87a], to produce a parser for "Miranda". We make no claims for this method of generating parsers compared to others, eg. Fairbairn's [Fai87]. We simply use it as it produces reasonably efficient parsers for a wide range of grammars. The syntax of "Miranda" that was used is shown in Appendix A.

This parser has four major components which we discuss in turn:

i) The process by which the action tables are derived from the grammar is described fully in [Lon87a] and will not be discussed here. We merely note that

this method of writing parsers allows easy construction of parsers for languages having complex syntaxes. It is worth noting that the hundreds of functions that define the action tables are mutually recursive and hence would have to all be type checked together if we did not provide type definitions for them. Adding type definitions produces a significant increase in speed of compilation.

ii) The lexical analyser is a relatively easy function to write in a straightforward manner. As in most programs of this sort, the lexical analyser is the real bottleneck in the parser, and unfortunately, when trying to parse "Miranda" (which has fifty lexical tokens) the lexical analyser written in Miranda is bound to be rather slow. This is because it must proceed by pattern matching on the head of the input when scanning for tokens. It is not so slow, however, as to be prohibitive and fairly large source scripts can still be parsed in a reasonable time.

iii) The reduce functions are the tree building functions associated with each production in the grammar. The sophisticated type system of Miranda makes writing functions for building parse trees a great deal easier than it is in other languages. The algebraic type mechanism allows one to define constructors corresponding to all the differing forms of parse tree that occur and to keep them distinct. It also proved to be very useful to hide the algebraic type implementing the parse trees inside an abstract type. This is because, while we do

lose straightforward pattern matching[9], during the development of large programs the definitions of algebraic types will usually be modified and this would otherwise involve rewriting all the code that uses the old definitions of the constructors. If, however, we use an abstract type, all the existing code can be used unmodified. In particular, one can add an extra field to each of the existing constructors without having to change anything but the algebraic type definition and the functions defined in the abstract type. This problem is solved in a more general way by Wadler's views [Wad85].

iv) The general grammar independent parser uses the action tables, the lexical analyser and the reduce functions to build parse trees by parsing the input. The details of its operation are given in [Lon87a].

Because the syntax is, in fact,context sensitive, it was necessary to implement a fairly complex set of syntax checks to avoid passing rubbish on to the type checker. These syntax checks could have been handled in a sophisticated fashion [Joh87] but we simply used another pass over the parse tree.

[9]By building a tag field, defined by another algebraic type, into the parse trees we can recover pattern matching.

```
typecheck' (ST_identifier name)
           variable_env constructor_env context type_var rewrites s
           [] [] []
      = (s',identity_substitution,type_var1)
        where
        (variable_type,type_var1)
                  = find_variable_binding variable_env type_var name
        s' = set_expression_type variable_type s
```

Figure 4: Type Check Identifier

3.7.2 Type Checker

The implementation of the type checking function is derived directly from the well typing algorithm given in [Lon87b]. Thus the typechecking function is called with three arguments: a type environment, a list of synonym rewrite rules and an expression. It returns the type checked expression and the substitution produced while type checking the expression. There are, in fact, a few more arguments to the function, but these are present in order that a great deal of useful information may be provided for each occurrence of a type error. This information includes not only a detailed description of the type error found but also a description of its location in the source script. Two cases of the type checking function are shown in Figure 4 and Figure 5. In the actual Miranda source there is almost one line of comment for each line of code but these have been omitted here.

When a type error is discovered the information is displayed immediately but it is also stored as an annotation on the parse tree so that the interactive type examination

```
typecheck' (ST_monadic op)
           variable_env constructor_env context type_var rewrites s
           [] [] []
      = (s',subst,type_var3)
        where
        (op_type,type_var1) = find_prefix_binding type_var op
        [s1] = substructures_of s
        (s1',s1_subst,type_var2)
                  = typecheck variable_env constructor_env
                              context type_var1 rewrites s1
                              [] [] []
        s1_type = expression_type_of s1'
        type_var3 = new_type_variable type_var2
        result = unify_types rewrites op_type etype
        etype = make_Type_function [s1_type,type_var3]
        op_unify_OK = unify_succeeds result
        unify_subst
                  = unify_subst'
                        , op_unify_OK
                  = identity_substitution
                        , otherwise
        Unifies unify_subst' = result
        s1'' = instantiate_structure unify_subst s1'
        s'_type
                  = instantiate unify_subst type_var3
                        , op_unify_OK
                  = Type_Error
                              (("Type Error in prefix expression, " :
                                pp_lines line line0 ++
                                ".\n" :
                                message_list ++
                                "\n" :
                                pp_context context,Type_Empty) :
                               failure_message_list
                              )
                              , otherwise
                    where
                    (message_list,line,line0) = pp_structure_lines s
           s' = set_expression_type s'_type
                                    (set_substructures [s1''] s)
           subst = compose_substitutions_list [unify_subst,s1_subst]
```


Figure 5: Type Check Unary Operator Expression

program can provide the same information. As many type errors as possible are found by simply regarding an expression that failed to type check as if its type were fully polymorphic. Its type will therefore unify with any other type and this prevents a cascade of type errors. The unification function used by the type checker is defined in a script parametrized by the type of the expressions being unified. This means it can be reused very easily in other programs by simply giving it different parameters. This approach also means one can consider the correctness of the implementation of the unification function independent of any details of the structure of types as used by the type checker.

3.7.3 Interactive User Interface

A type error occurs when the preceding occurrences of some variable have led us to deduce that its type is some instance of a type ρ and the current occurrence leads us to deduced that it is some instance of a type σ where these two types cannot be unified. Now it is often the case that the occurrence which causes this failure may not correspond to the real error in the script. It is for this reason that it is not enough merely to report such errors: we must also be able to examine the type successfully deduced for each of the sub-expressions of the script. It is not possible to produce a listing of a script annotated with the types deduced for each sub-expression, as can be seen by looking at the example in §3.6.4. In this simple example we use type

annotations on the sub-expressions, and it can be seen that even here they become somewhat confusing. The use of an interactive program to navigate around the type checked script, allowing investigation of the well typed sub-expressions as well as those that caused type errors, provides a solution to the problem. I also believe that examination of well typed scripts may well lead one to a better understanding of the type discipline of the language and hence speed the location of the true causes of type errors.

The user interface was simply required to demonstrate the the use of the implementation of the type checking algorithm. We therefore made no attempt to specify it [Chi85, Ale88] or to produce anything more sophisticated than a command driven interface. It was implemented as a number of tail recursive functions which maintain a stack of the previous positions as one navigates around the script.

The use of the interactive type examination program is best explained by the tutorial in Appendix B.8.

3.8 Conclusion

This example implementation highlighted a number of issues in the development of a complete system in a functional language. The first of these was that while pattern matching over algebraic types is very powerful during development it can be a hindrance. The second was that certain parts of the implementation, while not

our primary concern in this implementation, appear to benefit far less than others from the expressive power of a functional language. The two components in this implementation where this was particular apparent were the lexical analyser and the user interface. The lexical analyser is simply a 50 case definition using pattern matching to separate the cases. This is no more elegant than an imperative case statement and far less efficient[10]. The user interface is essentially an imperative, state based function. It consumes some input, modifies the state, generates some output and then starts again. A technique for writing user interfaces that greatly simplifies the task was developed after the implementation of this system [Lon89].

[10]Assuming sequential pattern matching.

Chapter 4

Implementing Logics

In this chapter we examine the use of Miranda as an implementation language for logics. It is interesting to contrast Miranda with existing languages already used for this purpose such as LISP [BM75] and ML [GMW79].

The major advantage offered by Miranda over LISP is its sophisticated type system and in particular the abstract type mechanism. In LISP one must convince oneself that the entire implementation of the system is correct whereas with Miranda, and ML, one can restrict ones attention to a relatively small section of the implementation. This can be done if we implement the structures we intend to correspond to terms, formulae and theorems by an abstract type. The type security will then *guarantee* that the only way to construct an expression of type *theorem* is by the application of the functions defined in the abstract type that return objects of this type. Thus

we can ignore the rest of the implementation and need only convince ourselves of the correctness of the implementation of the abstract type.

The advantages of Miranda over a language like ML are less striking but, I believe, still important. The first is the general point that using functional languages is better than using imperative languages, and ML retains certain imperative features. The fact that any attempts at verification of implementations will be easier, if not easy, in a functional language becomes even more important when we are implementing logics. There seems little point in using a theorem prover to prove required results when the verification of its implementation is harder than it need be. The second is that lazy evaluation allows one to express various search functions that return lists of alternatives in a very concise manner, while not having to be concerned about the inefficiency of calculating all such alternatives when only one is needed.

It must be said that the proponents of ML would claim that assignment and exception handling are needed if one is to implement usable systems. The first is useful in that one can work at the ML top-level and use assignment to record partial results. It is, however, not difficult to implement a user interface that provides all the facilities required for handling and accessing partial results. The exception handling is used in order to allow flexible input and also in the implementation of complex tactics. I believe that an approach like that described in [Lon89] allows one to both write flexible input functions and also to write complex tactics using exceptions to invoke

alternatives on failure. Other approaches to writing user interfaces in functional languages might also be used [O'D85, Dwe89, FL89, Koo87].

The remainder of this chapter is taken up with two different examples of implementing logic in Miranda which are contrasted with the approach used in the Edinburgh LF [HHP87]. These can also be seen as covering a range of abstraction in the specification of the logic. In the first implementation we take a standard presentation of the logic and in the second the implementation already existed and the specification we use was developed afterwards. The definition of a logic in LF can be seen as using a more abstract and more powerful specification language than the standard presentation. These examples demonstrate that a functional implementation can be easily related to the specification of the logic. The second example demonstrates the power of even simple rewriting rules in an otherwise somewhat limited logic.

4.1 Intuitionistic Logic

We describe an implementation of the *first order intuitionistic logic* as described by Prawitz [Pra73]. The logic is presented as a natural deduction system in the style due to Gentzen. In this approach we consider the deductive inferences involving a logical constant as giving us the meaning of that constant. We further reduce all these inferences to atomic inference steps each of which involves only one logical constant. There are two kinds of atomic inference step; those that allow the introduction of

a logical constant and those that eliminate a logical constant. In the first case the outermost symbol of the conclusion will be the constant introduced while in the second the outermost symbol of one of the premisses will be the constant eliminated.

We regard the object of interest as being the proof rather than simply the resulting theorem. A proof will start from certain assumptions and produce some conclusion, with the possibility that some of the inference steps in the proof may discharge some of these assumptions. We can represent proofs as *derivations* presented in the form of trees. The leaves of the tree are the assumptions, discharged and undischarged, and the root of the tree is the conclusion of the proof. The internal nodes of the tree represent the individual inference steps which derive sub-conclusions from the conclusions of the sub-trees above. We say that the conclusion of a proof *depends* on those assumptions of the proof that have not been discharged.

We now describe the atomic inference rules in a schematic form, using a notation essentially the same as that used by Prawitz. The formulae standing above the line represent the conclusions of derivations and the formula below the line the conclusion of the derivation built with the inference rule. We let the symbols A, B and C range over formulae, the symbols x and y over variables and the symbol t over terms. We use the notation $A[t/x]$ to indicate the formula obtained by substituting the term t for the free occurrences of the variable x in the formula A. Multiple occurrences of the same formula, A say, are to be understood as referring to alpha-equivalent

formulae. We further use the notation $\{A\}$ standing above a premiss to indicate that assumptions of this form are discharged from the proof of this premiss by the inference rule. We also assume that the language contains a constant Λ for falsehood. We label the inference rules by the logical constant they involve followed by I for introduction rules and by E for elimination rules.

$\wedge I)$ $\quad \dfrac{A \quad B}{A \wedge B}$

$\wedge E)$ $\quad \dfrac{A \wedge B}{A} \quad \dfrac{A \wedge B}{B}$

$\vee I)$ $\quad \dfrac{A}{A \vee B} \quad \dfrac{B}{A \vee B}$

$\vee E)$ $\quad \dfrac{\begin{array}{ccc} & \{A\} & \{B\} \\ A \vee B & C & C \end{array}}{C}$

$\Rightarrow I)$ $\quad \dfrac{\begin{array}{c} \{A\} \\ B \end{array}}{A \Rightarrow B}$

$\Rightarrow E)$ $\quad \dfrac{A \Rightarrow B \quad A}{B}$

$\forall I)$ $\dfrac{A}{\forall x.A[x/y]}$

Provided *either* x is not free in $A[x/y]$ *or* y is not free in the assumptions that the conclusion A depends on and either x is not free in A or x is not free in these assumptions.

$\forall E)$ $\dfrac{\forall x.A}{A[t/x]}$

$\exists I)$ $\dfrac{A[t/x]}{\exists x.A}$

$\exists E)$ $\dfrac{\exists y.A \qquad \overset{\{A[x/y]\}}{B}}{B}$

Provided x is not free in B and is also not free in the assumptions that the conclusion B depends on (except those of the form $A[x/y]$ discharged by this inference rule).

$\Lambda)$ $\dfrac{\Lambda}{A}$

We require that the formula A be an atomic formula different from Λ. If we allowed A to be a compound formula, or Λ, it would invalidate the principle that the meaning of the logical constants is given by the inference rules that introduce and eliminate them.

If we assume that $\sim A$ is taken as shorthand for $A \Rightarrow \Lambda$ then we have the following two special cases of $\Rightarrow I$ and $\Rightarrow E$:

$$\sim I) \qquad \frac{\begin{array}{c}\{A\}\\ \Lambda\end{array}}{\sim A}$$

$$\sim E) \qquad \frac{A \quad \sim A}{\Lambda}$$

We now describes various aspects of the implementation of this logic.

4.1.1 Parser

As already mentioned we have implemented the types of terms and formulas by an abstract type. This abstract type defines the type of the predicates that take terms and produce formulae as well as the types of terms and formulae. Each logical constant has three functions associated with it. For example for conjunction, $\wedge$, we have:

```
is_and ::  formula -> bool
mk_and ::  formula -> formula -> formula
brk_and ::  formula -> (formula,formula)
```

These functions allow us to check the form of formulae, build formulae from other formulae and take formulae apart. We have a similar set of functions for the atomic formulae produced by the application of a predicate to some terms:

```
is_relation ::  formula -> bool
mk_relation ::  predicate -> [term] -> formula
brk_relation ::  formula -> (predicate,[term])
```

We finally have functions for distinguishing the various forms of term: constants, numbers, variables and applications.

The parser, again written using the technique described in [Lon87a], uses these functions to build expressions of type `term` or `formula`.

We also use a matching function derived during the work on type checking to provide tests for alpha equivalence of formulae and the derivation of substitution instances of formulae.

4.1.2 Derivations

We now describe the abstract type implementation of derivations and the inference rules that build them. As already stated we are interested in proofs, not simply theorems, and thus we at all times manipulate complete derivations and not merely the theorems they prove. The abstract type definition is given in Figure 6 and it can be seen that there are only sixteen functions that return objects of type `derivation` and hence only sixteen functions that need be examined in order to ensure the correctness of any system built using these functions. We also have functions for extracting the conclusion of a derivation and the assumptions that this conclusion depends on. The implementation type for derivations is of course an algebraic type:

```
abstype derivation
with
      assume ::  formula -> derivation
      for_all_I, for_all_E,
      there_exists_I, there_exists_E,
      implies_I, implies_E,
      or_lI, or_rI, or_E,
      and_I, and_lE, and_rE,
      not_I, not_E,
      falsity
            ::  [derivation] -> formula -> derivation
      conclusion ::  derivation -> formula
      assumptions ::  derivation -> [formula]
      pp_tree ::  derivation -> [sys_message]
```

Figure 6: Derivations

```
derivation == derivation'
derivation' ::= Assume formula bool |
                Infer inference [derivation'] formula
```

where `inference` is an enumerated algebraic type with one constructor for each of the fifteen inference rules. Thus a derivation is either an assumption or a valid application of one of the fifteen inference rules to some list of derivations resulting in the given conclusion. We now describe the implementations of some of these inference rules. Each inference rule is given a list of derivations and the proposed conclusion and if the conclusion can validly be deduced from the derivations by that inference rule then the appropriate derivation is built.

- The `assume` function takes an arbitrary formula and assumes it:

```
assume f = Assume f False
```

- The and_I function takes two arguments: a list of two derivations and a formula which should be the conjunction of the the conclusions of these derivations.

```
and_I [dl,dr] f
      = error "and_I: Not conjunction\n"
            , ~ is_and f
      = error "and_I: Not alpha-equivalent\n"
            , ~ alpha_equivalent l cl \/
              ~ alpha_equivalent r cr
      = Infer And_I [dl,dr] f
            , otherwise
        where
        cl = conclusion dl
        cr = conclusion dr
        (l,r) = brk_and f
```

Notice that we use the functions for testing and breaking apart a conjunction as well as the function that tests for alpha equivalence.

- There are two forms of the and_E inference rule, and_lE and and_rE. The first extracts the left conjunct of the conclusion of the argument derivation and the second the right conjunct. We present the implementation of the first of these:

```
and_lE [d] f
      = error "and_lE: Not conjunction\n"
            , ~ is_and c
      = error "and_lE: Not alpha-equivalent\n"
            , ~ alpha_equivalent l f
      = Infer And_lE [d] f
```

```
        , otherwise
    where
    c = conclusion d
    (l,r) = brk_and c
```

We again use the functions for testing and breaking apart formulae and for checking alpha equivalence.

- The `implies_I` inference rule discharges an assumption from the resulting derivation. The function that is used to do this is defined as follows:

```
discharge formula_list (Infer inference derivation_list formula)
      = Infer inference
              (map (discharge formula_list) derivation_list)
              formula
discharge formula_list (Assume formula False)
      = Assume formula
               (or (map (alpha_equivalent formula) formula_list))
discharge formula_list derivation
      = derivation
```

This function takes a list of formulae and a derivation and discharges from the derivation any assumption that is alpha-equivalent to any of the formulae in the list. Using this function it is then easy to implement the `implies_I` rule:

```
implies_I [d] f
      = error "implies_I: Not implication\n"
            , ~ is_implies f
      = error "implies_I: Not alpha-equivalent\n"
            , ~ alpha_equivalent r c
      = Infer Implies_I [d'] f
```

```
    , otherwise
where
c = conclusion d
(l,r) = brk_implies f
d' = discharge [l] d
```

- The `for_all_E` inference rule must check that the proposed conclusion is in fact an instance, of a particular form, of the conclusion of the argument derivation.

  In order to do this it uses a function called `single_substitution` which is defined by use of the formulae matching function already mentioned. This function takes two formulae and checks that the first is obtained from the second by a singleton substitution $[t/x]$. If the two formula are alpha-equivalent then the variable argument, y say, is used to form the identity substitution $[y/y]$. If the formulae do not match or are matched by a non-singleton substitution this function will return `None`.

  This inference rule can validly be applied if the proposed conclusion is a singleton-substitution instance of the body of the universal quantification which is the conclusion of the argument derivation. We also require that the variable being substituted for be the variable bound by the universal quantifier.

```
for_all_E [d] f
      = error "for_all_E: Not universal\n"
            , ~ is_forall c
      = error "for_all_E: No match\n"
            , ~ match_OK
```

```
    = Infer For_all_E [d] f
        , otherwise
      where
      c = conclusion d
      (var,c') = brk_forall c
      result = single_substitution var f c'
      match_OK = result ~= None & var' = var
      Value (exp,var') = result
```

- The `for_all_I` inference rule uses matching and also checks the conditions imposed on the free variables of the assumptions of the argument derivation. We again use the `single_substitution` function and now also use a function that returns a list of all the free variables in a formula.

```
for_all_I [d] f
      = error "for_all_I: Not universal quantification\n"
          , ~ is_forall f
      = error "for_all_I: No match\n"
          , ~ match_OK
      = error "for_all_I: Free variables problem\n"
          , ~ free_variables_OK
      = Infer For_all_I [d] f
          , otherwise
        where
        c = conclusion d
        (var,c') = brk_forall f
        result = single_substitution var c' c
        match_OK = result ~= None & exp = var
        Value (exp,var') = result
        free_variables_OK
            = ~ member (free_variables c') var \/
              (~ member assumptions_free_variables var' &
               (~ member (free_variables c) var \/
                ~ member assumptions_free_variables var))
        assumptions_free_variables
            = concat (map free_variables (assumptions d))
```

While the free variable condition does look a little complex it is really quite straightforward.

- The **there_exists_E** inference rule takes a list of two derivations and a formula. It uses a locally defined function **dischargeable** to find those assumptions of the second derivation that may be discharged. This function again use the function for alpha equivalence and the function that returns the free variables of a formula and also uses the **discharge** function. Notice that this function also uses the higher order functions **map** and **filter**.

```
there_exists_E [dmaj,dmin] f
      = Inference_Error ["there_exists_E: Not existential\n"]
          , ~ is_exists cmaj
      = Inference_Error ["there_exists_E: Not alpha-equivalent\n"]
          , ~ alpha_equivalent cmin f
      = Derivation (Infer There_exists_E [dmaj,dmin'] f)
          , otherwise
        where
        cmaj = conclusion dmaj
        cmin = conclusion dmin
        (var,cmaj') = brk_exists cmaj
        dmin_assumptions = assumptions dmin
        dmin' = discharge (filter dischargeable dmin_assumptions)
                          dmin
        dischargeable fm
            = match_OK & free_variables_OK
              where
              result = single_substitution var fm cmaj'
              match_OK = result ~= None & var'' = var &
                         is_variable exp
              Value (exp,var'') = result
              free_variables_OK
                  = ~ member (free_variables cmin) exp &
```

```
        ~ member (concat (map find_free_variables
                                dmin_assumptions)) exp
find_free_variables f'
    = [] , alpha_equivalent f f'
    = free_variables f' , otherwise
```

These examples demonstrate the clear relationship between the specifications of the inference rules of the logic and their corresponding implementations. This is due to the clarity of expression that is obtained by the use of abstract types and higher order functions. This relationship makes it far easier to convince oneself of the correctness of the implementation.

While the implementation described is secure and can be used it has one major flaw in the way it handles errors. Calling the `error` function when a derivation cannot be built does preserve the security of the type of derivations but makes it difficult to build a system using these functions. What we require is some non-error return that indicates that a derivation was not in fact produced. It is possible to elaborate the underlying implementation type to include an error indication and add functions to check whether we have an error and extract its description. A far better solution is to let all the derivation building functions return a value of an algebraic type something like[1]:

```
deriv_result ::= Good derivation | Bad [char]
```

[1]The implementation does in fact use this approach, it is omitted here for ease of presentation.

This ensures that any value of type **derivation** is always a derivation.

The final function defined in the abstract type, **pp_tree**, provides centred tree structured presentations of the derivations. These presentations show not only the inference rules but also the conclusions of all the intermediate derivations. This function is implemented as a particular application of a more general tree pretty printing function. This more general function takes an object of the algebraic type **pretty_printing_tree** and, by folding pretty-printing functions over it, pretty prints it.

```
node_name == [char]
object == [char]

pretty_printing_tree
      ::= Pretty_printing_Node node_name object [pretty_printing_tree] |
          Pretty_printing_Leaf node_name object
```

Thus we need merely turn a **derivation** tree into a **pretty_printing_tree** to obtain a pretty-printer for derivations. This can easily be done by mapping functions to extract the conclusions and inference rule names over the derivation tree. This pretty printer is not a generic or language independent pretty printer as described elsewhere [Jok89, Opp80, Rub83] and assumes we can display arbitrarily wide output.

Given this abstract type we can define various derived inference rules in terms of the basic inference rules. The first set of such functions are simpler versions of the basic inference rules where we need only provide the minimum amount of information

to the function. A good example is the simpler version of the $\Rightarrow I$ rule which only takes the formula being discharged and not the entire resulting conclusion.

```
implies_I' l d
      = implies_I [d] (mk_implies l (conclusion d))
```

Given this derived inference rule we can now define a rule that discharges all the assumptions that the conclusion of a derivation depends on.

```
disch_all d
      = foldl (converse implies_I') d (assumptions d)
```

Th function `converse` simply reverses the order of the arguments supplied to `implies_I'`.

```
converse f a b = f b a
```

This derived rule simply extracts the assumptions on which the conclusion depends and then discharges them in sequence using the `implies_I'` derived rule. Schematically we might represent this derived rule as follows:

$$\frac{\begin{array}{c}\{A_1 \cdots A_n\} \\ B\end{array}}{A_n \Rightarrow \cdots \Rightarrow A_1 \Rightarrow B}$$

where the $A_1, \ldots, A_n$ are the assumptions that the conclusion B depends on.

We can also use the `implies_I'` derived inference rule to write another more complex, recursive, derived inference rule. This rule also uses the derived rules `and_lE'` and `and_rE'` which are related to `and_lE` and `and_rE` as `implies_I'` is related to `implies_I`.

```
implies_I'' f d
      = implies_I' f d , ~ is_and f
      = implies_I' f d2 , otherwise
        where
        (fl,fr) = brk_and f
        d' = assume' f
        d1 = implies_I'' fl (implies_I'' fr d)
        d2 = implies_E' (implies_E' d1 (and_lE' d')) (and_rE' d')
```

Given a formula which is an arbitrary nesting of conjunctions this discharges all the conjuncts in the formula from a derivation, producing a derivation whose conclusion is an implication with the formula as antecedent. This is achieved by recursive calls of the function on the two conjuncts in the formula. In this way we can define increasingly complex derived inference rules without having to check the validity of any of them because at worst they will merely fail to produce a result rather than produce an incorrect result.

At present this system lacks one feature that would be essential if we intended to use it in any serious way: it should include some form of type checking. . The obvious thing to do is to only allow constants that have been introduced by some sort of declaration like that used for Miranda algebraic types, and to then use a form of the Milner type checking algorithm. As this implementation was only intended as a study of the implementation of a standard logic to serve as a contrast to the 'B' logic described in §4.3 we did not take the work further in this direction. A more detailed description of a proposed type checking discipline for proofs implemented in this fashion is presented in Appendix C.

Given this implementation of the secure type of derivations we could implement a number of different theorem provers on top of it. The simplest would be a bottom-up system that only allowed us to apply the inference rules to previously built derivations. A more complex system would require the implementation of a top-down tactics based system in the style of LCF where the inference rules would be applied by the validation functions. As in LCF it would probably be necessary to provide some composite tactics for performing multiple inferences. In particular, tactics that allow composite inferences like rewriting greatly simply the production of proofs.

A small interactive subgoaling function was written which allows one to elaborate a derivation in a top-down fashion. The interactive user-interface to this subgoaling function was written using interactions in the style described in [Lon89]. Starting from a goal one can generate a set of subgoals by invoking a tactic. For this small subgoaling function we only provided tactics corresponding to the basic inference rules. In this way one can build up a partial proof tree where the leaves are unproved goals. One may navigate about this partial proof tree, allowing one to prove these goals in any order. As one completes the proofs of the subgoals of a goal, the associated inference rule is used to produce a derivation with the desired conclusion.

4.2 Defining Logics in LF[2]

The Logical Framework is intended to allow the definition of a wide class of logics. The LF is based on a version of type theory related to the early work of Martin-Löf, that is, a typed lambda calculus with dependent types. The syntax, inference rules and proofs of a logic are all defined in terms of this type theory. The lambda calculus structure of the logic provides for binding operators, variable substitution, variable capture and alpha-equivalence in the syntax of the logic. It also allows for schematic abstraction and instantiation in the inference rules of the logic. The treatment of inference rules and proofs is based on an extended notion of *judgements*. The details of these ideas will be presented as required.

The type theory has three basic classes of entity: objects, types and families of types, and kinds. Objects will have types, types will have kinds and kinds will also have kinds. If we let c range over constants, x and y over variables, M and N over objects and A and B over types then we have the following forms of object.

$$c \mid x \mid \lambda x : A.M \mid M\ N$$

That is we have constants, variables, typed lambda abstractions and applications. We also have the following forms of families of types:

$$c \mid \Pi_{x:A}.B \mid \lambda x : A.B \mid A\ M$$

[2]We will not provide a detailed description of the Edinburgh Logic Framework here as this is provided in a number of Edinburgh LFCS reports [HHP87].

Thus we have constants, dependent types, lambda abstracted types and types obtained by specialising a dependent type with an object. If we now let K range over kinds we have the following forms of kinds:

$$\mathit{Type} \mid \Pi_{x:A}.K$$

The types will all be of kind *Type* while the second form allows functions which return types. Any function definable in the system will have a type as its domain but may have either a type or a kind as its range.

Given these three classes we can now define signatures and contexts. A signature provides the bindings for the constant objects, types and kinds. If we let Σ range over signatures we have the following forms of signature:

$$\langle\rangle \mid \Sigma, c : A \mid \Sigma, c : K$$

A signature can either be empty or can be built by extending a signature with either a constant object or a constant type or kind. Contexts introduce variables in a similar way, though variables only range over objects and therefore only ever have types. If we let Γ range over contexts we have the following forms of context:

$$\langle\rangle \mid \Gamma, x : A$$

The LF has a set of inference rules with which one can deduce if a term is well-typed. A term is well typed, in some signature and context, if it can be shown it has either a type or a kind or that it is a kind.

4.2.1 Intuitionistic Logic

We shall examine how the first order intuitionistic logic implemented in §4.1 is defined in the LF. We start with the syntax of the logic and then follow this with a description of how proofs and inference rules are defined in the LF along with some examples.

Syntax

A logic is defined in the LF by first defining a signature declaring the constants required by the syntax. Starting with the empty signature we extend it with two constants to represent the syntactic categories for terms and formulae. The terms are the objects over which we may quantify and the formulae stand for propositions. These constants, τ and ϕ, are types and therefore have kind *Type*:

$$\langle\rangle,\ \tau : \mathit{Type},\ \phi : \mathit{Type}$$

We can now further extend the signature with constants representing the formula building operations of the syntax. If we use the notation $A \rightarrow B$ to represent the type $\Pi_{x:A}.B$ when x does not occur free in B then the declarations for conjunction, disjunction and implication all take the following form:

$$\wedge : \phi \rightarrow \phi \rightarrow \phi$$

Thus these constants represent functions that take two formulae and return a formula. The negation, $\sim$: $\phi \rightarrow \phi$, and falsity, $\Lambda : \phi$, formulae are declared in a similar fashion.

When it comes the the universal and existential quantifications, $\forall$ and $\exists$, we must model the fact that they are variable binders. This is done by declaring constants representing higher order functions as follows:

$$\forall : (\tau \rightarrow \phi) \rightarrow \phi$$

All of this corresponds closely to the definitions of the abstract types of `term` and `formula` and their associated functions in §4.1.1.

Proofs and Inference Rules

A logical system can be seen as a system for establishing proofs of assertions of certain forms by means of basic or derived rules. It is these assertions of the logic that are modelled by judgements. Judgements are defined as dependent types over the expressions they make assertions about. The basic assertions are represented by basic dependent judgements while the inference rules, derived rules and proofs are represented by higher order judgement types. Elements of a judgement type are used to represent the evidence for that judgement. Thus the basic assertions and inference rules of a logic are obtained by extending the signature with constants of the appropriate judgement types. In order to prove a derived rule one must present an element of the appropriate higher order judgement type. These elements are obtained by combining the constants with lambda abstractions and applications. The lambda abstractions of the LF are used to capture both the schematic nature of inference

rules and the parametrized forms of inference rules. The schematic instantiation of inference rules is captured by lambda application while beta reduction captures all forms of substitution.

In our example logic the only judgement is the assertion that we have proved some formula to be a theorem, this might be written $\vdash \phi$. We must therefore extend the signature with a further constant which will correspond to this judgement:

$$\textbf{Proved} : \phi \rightarrow \mathit{Type}$$

Thus presenting an object of type **Proved** ϕ is the proof of $\vdash \phi$. We can now define the inference rules of the logic as higher order judgements. We have two particular forms of higher order judgement that are used when defining inference rules.

1. The *hypothetical* judgement is represented by a function from proofs of judgements to proofs of judgements, **Proved** $\phi_1 \rightarrow$ **Proved** ϕ_2. This asserts that the judgement **Proved** ϕ_2 is a logical consequence of the judgement **Proved** ϕ_1 according to the rules of the logic.

2. The *schematic* judgement is represented by a function from objects x of some type A to proofs of judgements, $\Pi_{x:A}$.**Proved** ϕ. This asserts that the judgement **Proved** ϕ is evident for any object x of type A.

These two forms of judgement are used when defining the inference rules of a logic to model the discharge of assumptions and the free variable conditions associated with

some inference rules. The latter form is also used to model the schematic nature of inference rules and thus all inference rules will be modelled by schematic judgements. Those inference rule which discharge assumptions will therefore be both schematic and hypothetical.

The following examples demonstrate the use of these two forms of judgement for our example logic. A constant of the types described below is added to the signature and in this way defines the basic inference rules of the logic.

$\wedge I$) This introduction rule is both schematic, in that it applies for any conjunction of formulae, and hypothetical, in that the proof of the conjunction depends on the proofs of the conjuncts.

$$\Pi_{a,b:\phi}.\textbf{Proved}\ a \rightarrow \textbf{Proved}\ b \rightarrow \textbf{Proved}\ a \wedge b$$

The correspondence between this judgement type and the inference rule it represents is clear.

$\wedge E$) These elimination rules are also both hypothetical and schematic, we only show one of the rules.

$$\Pi_{a,b:\phi}.\mathit{Proved}\ a \wedge b \rightarrow \mathit{Proved}\ a$$

$\Rightarrow I$) This inference rule is both hypothetical and schematic but is also higher order. It used a hypothetical judgement as an argument to represent the discharge of

an assumption. This can be read as saying that, given a function from proofs of a to proofs of b we can obtain a proof of $a \Rightarrow b$.

$$\Pi_{a,b:\phi}.(Proved\ a \rightarrow Proved\ b) \rightarrow Proved\ a \Rightarrow b$$

The correspondence between this judgement and the inference rule it represents is still quite clear.

$\Rightarrow E$) The elimination rule for implication is represented by a simple hypothetical and schematic judgement type.

$$\Pi_{a,b:\phi}.Proved\ a \Rightarrow b \rightarrow Proved\ a \rightarrow Proved\ b$$

$\forall E$) The representation of the inference rules for the quantifiers are more complex as we use function types to represent expressions which may have free variables. Thus the LF variable Φ declared by $\Phi : \tau \rightarrow \phi$ represents a formula which may have a free variable. If we then have a LF variable π declared by $\pi : \tau$, that is it is a term, then the expression $\Phi\ \pi$, which has type ϕ, represents the formula obtained by substituting the term π for the free variable in the formula Φ. Recalling the declaration of the constant $\forall$ it is clear that the expression $\forall\ \Phi$ represents the formula obtained by universally quantifying over the free variable of Φ. Given all of this we can now define the constant corresponding to this inference rule.

$$\Pi_{\Phi:\tau\rightarrow\phi}.\Pi_{\pi:\tau}.Proved\ \forall\ \Phi \rightarrow Proved\ \Phi\ \pi$$

Careful consideration of the above will convince one it does represent the inference rule but I believe the Miranda implementation in §4.1 is more easily understood.

$\exists I$) The representation of this introduction rule is similar to that of the $\forall E$) rule and uses the same variable names as that rule.

$$\Pi_{\Phi:\tau\to\phi}.\Pi_{\pi:\tau}.Proved\ \Phi\ \pi \to Proved\ \exists\ \Phi$$

$\forall I$) The representation of this introduction involves an added complexity as we must find a way to represent the condition that the variable to be quantified over not appear in the assumptions. Given $\Phi : \tau \to \phi$ consider the following judgement type:

$$\Pi_{\pi:\tau}.Proved\ \Phi\ \pi$$

This states that the judgement $Proved\ \Phi\ \pi$ is evident for any term π. Thus this judgement cannot in any way be dependent on the free variable of the formula Φ and we can therefore quantify over it:

$$\Pi_{\Phi:\tau\to\phi}.(\Pi_{\pi:\tau}.Proved\ \Phi\ \pi) \to Proved\ \forall\ \Phi$$

The free variable condition is essentially a requirement on the generality of the proof, and this is represented by a function that takes a schematic judgement

as argument. Again, it is possible to convince oneself of this representation but I believe the Miranda implementation in §4.1 is clearer.

$\exists E$) This final inference rule discharges an assumption and also enforces a free variable condition. The discharge of an assumption can again be modelled by having a hypothetical judgement as the argument. The free variables condition is more complex: it requires that free variables in the assumption being discharged not occur in any of the other assumptions or in the conclusion to be inferred. The higher order judgement type required to represent this inference rule is:

$$\Pi_{\Phi:\tau\to\phi}.\Pi_{\Psi:\phi}.\mathit{Proved}\ \exists\ \Phi \to (\Pi_{\pi:\tau}.\mathit{Proved}\ \Phi\ \pi \to \mathit{Proved}\ \Psi) \to \mathit{Proved}\ \Psi$$

The judgement type of the second argument states that for any term π it is evident that the proof of Ψ is a logical consequence of the proof of $\Phi\ \pi$. Thus this proof cannot depend on the free variable of the formula Φ: this captures the first part of the free variables condition. The second part of the condition is derived from the variable scoping rules of LF and the fact that Ψ is defined before π and hence cannot refer to it.

The LF is a very powerful abstraction in that it allows us to specify a large range of logics in the same framework. This can clarify the similarities and differences between these various logics. It also means that an implementation of the LF then gives

you implementations of all these logics. The extremely abstract nature of the specifications does, however, mean that they can be quite hard to relate to the standard presentation of a logic.

4.3 A Miranda Implementation of Paul Gardiner's Logic of 'B'

4.3.1 Introduction

The aim of the 'B' system, written by J.R. Abrial, and the logic presented by Paul Gardiner is to facilitate the use of a natural deduction style in theorem proving. Gentzen's natural deduction analysis of proof theory [Pra73] is described in §4.1. We describe an implementation, in Miranda, of the core of a theorem prover based on this logic.

The essence of this approach is the analysis into atomic inferences, so that it may be claimed [Pra73] that the essential logical content of intuitive logical operations can be understood as being composed of the atomic inferences isolated by Gentzen. It is for this reason that the term *natural deduction* is seen as appropriate. The system can also be seen as natural in the more superficial sense of corresponding well with informal practices.

The 'B' system aims to provide a framework for reasoning in a natural deduction

style where the set of inferences rules is defined by the user. The logic presented by Paul Gardiner provides a formalism in which inference rules can be presented and applied in a secure manner. The essence of the system is thus the presentation, by the user, of inference rules. We must, therefore, consider what forms inference rules can take.

Inference rules are, in general, presented in a schematic form and any particular application of a rule in fact invokes some instance of the schematic form. For instance the standard conjunction-introduction rule is presented as:

$$\frac{A \quad B}{A\&B}$$

Where A and B stand for arbitrary formulae. It means that given proofs of formulae A and B we can derive a proof of the formula $A\&B$. Given a proof of **True** and the following instance of the above rule:

$$\frac{\textbf{True} \quad \textbf{True}}{\textbf{True} \ \& \ \textbf{True}}$$

we can derive a proof of **True** & **True**. Thus we need to be able to present inference rules in a schematic form and then apply particular instances of them.

Certain of the inference rules may discharge some of the assumptions of the sub-proofs and we must be able to accept this form of inference rule. An example is the standard rule for the introduction of an implication:

$$\frac{\begin{array}{c}\{A\}\\ B\end{array}}{A \Rightarrow B}$$

This means that given a proof of a formula B based on some assumption A we can deduce a proof of $A \Rightarrow B$ that does not depend on the assumption A.

As a further complexity we must be able to cope with side conditions. These side conditions are usually stated alongside the inference rule. These conditions must be met by the antecedents, and possibly the consequent, of the proposed application of the inference rule in order for it to be a valid use of the rule. An example is the standard rule for the introduction of a universal quantifier:

$$\frac{A(a)}{\forall x A(x)} \quad \text{Where } a \text{ is not free in the assumptions } A(a) \text{ depends on.}$$

Here, the side condition means that the variable we wish to quantify over must not be free in the assumptions of the proof of the premiss $A(a)$. We must be able to encode these side conditions into our statement of an inference rule, and manipulate them along with the inference rule.

These considerations lead to the adoption of the deductive system and the meta-language described below.

This presentation evolved from Paul Gardiners draft papers from May and August of 1987 entitled "The Logic of 'B'". My approach follows more closely that of the earlier paper, though I modify the logic slightly. The major modification I introduce

is to add antecedents constraining the forms of object language expressions where these are appropriate though I also restate some of the axioms for clarity.

We present a meta-language which is powerful enough to express the side conditions associated with inference rules along with the inference rules themselves. This meta-language has embedded in it an object language in which the inference rules themselves are expressed. We then describe a deductive system for manipulating statements in this meta-language.

4.3.2 Object and Meta Language

We first present the object language in which the inference rules are to be expressed. It is designed to favour inference rules such as those of the predicate calculus. This object language contains function, logical and quantifier symbols; we assume nothing about the members of these symbol classes except that they are distinct. We now describe these symbol classes and give the the particular symbols we will use as representatives of these classes:

Constants The object language constants will be represented by the single constant "0".

Variables The object language variables will be represented by the variables "a" and "b".

Function symbols The function symbols may be of any arity; we shall represent them all by the binary infix function symbol "+".

Predicate symbols The predicate symbols may be of any arity; we shall represent them all by the binary infix predicate symbol "=".

Logical symbols The logic symbols may be of any arity; we shall represent them all by the binary infix logic symbol "&".

Quantifier symbols The quantifier symbols of the object language will be represented by the symbol "$\forall$".

We only need take a representative for each of these symbol classes, as the axioms for any symbol apply to all symbols of that class.

The meta-language resembles the atomic formulae fragment of the predicate calculus. We choose our predicate symbols to facilitate reasoning about an object language intended to express predicate calculus style inference rules and also the side conditions of such inference rules. The meta-language has the following symbol classes:

Constants The object language constants and variables are embedded in the meta-language under their own names. Thus "0", "a" and "b" are all meta-constants.

Variables The meta-variables may denote arbitrary expressions in the object language. We shall use upper case characters to represent meta-variables.

Function symbols The object language function, predicate, logical and quantifier symbols are embedded in the meta-language as meta-function symbols. Thus "+", "=", "&" and "$\forall$" are all meta-function symbols. We also have the meta-function symbol ":=" which denotes *safe substitution.*

Thus $[\phi := \psi]\theta$ denotes the expression obtained by the safe substitution of the term denoted by ψ for the variable denoted by ϕ in the expression denoted by θ. Safe substitution means we don't allow the substitution of a term for a free variable in an expression unless the free variables of the term will also be free variables of the resulting expression. Thus we can only replace free variables with expressions that will then be free sub-expressions in the resulting expression.

Predicate symbols We have six meta-predicates:

- Formula ϕ

  ϕ denotes a formula[3].

- Term ϕ

  ϕ denotes a term.

- Var ϕ

  ϕ denotes a variable.

[3]Gardiner uses the confusing name "pred" for this meta-predicate

- $\phi \setminus \psi$

  ϕ denotes a variable which is not free in the expression denoted by ψ.

- $\phi == \psi$

  ϕ and ψ denote equivalent expressions.

- $\psi_1, \ldots, \psi_n \vdash \phi$

  The formulae denoted by $\psi_1, \ldots, \psi_n$ entail the formula denoted by ϕ.

As seen above we shall use Greek letters $\phi, \psi, \ldots$ to represent arbitrary meta-language expressions, unless otherwise stated.

4.3.3 Sequents

Sequents in the meta-language make statements about meta-language formulae, which can only be formed by the six meta-predicates. We will write sequents as:

$$\frac{\begin{array}{c}\phi_1 \\ \vdots \\ \phi_n\end{array}}{\psi}$$

where $\phi_1, \ldots, \phi_n$ and ψ are meta-language formulae.

The side conditions of an inference rule will be stated in terms of the meta-predicates Formula, Term, Var, $\setminus$ and $==$ and the meta-function-symbol $:=$. The inference rules themselves will be expressed in terms of the meta-predicate $\vdash$.

A sequent is valid if every assignment of object language expressions, embedded as terms in the meta-language, to meta-variables that makes the antecedents true also makes the consequent true.

We can derive new sequents as follows[4].

1. We can always deduce as a consequent any of the antecedents.

$$\frac{\begin{array}{c}\phi_1 \\ \vdots \\ \phi_n\end{array}}{\phi_i} \quad \text{Where } 1 \leq i \leq n.$$

2. We can take a sequent and replace its meta-variables with arbitrary meta-expressions. Thus we can take an instance of the statement of an inference rule as required. Thus given:

$$\frac{\begin{array}{c}\phi_1 \\ \vdots \\ \phi_n\end{array}}{\psi}$$

and given a substitution ρ of meta-language expressions for meta-variables we can derive:

[4]This section of Paul Gardiner's logic is still not completed and this version may not be powerful enough.

$$\frac{\begin{array}{c}\rho(\phi_1)\\ \vdots \\ \rho(\phi_n)\end{array}}{\rho(\psi)}$$

3. We can derive a new valid sequent if we can start with the list of its antecedents and expand this list of meta-formulae until the consequent is obtained. The list of meta-formulae can be extended by adding to it the consequent of any sequent whose antecedents are already present in the list. This method is justified by the use of the cut rule for sequents. This states that given:

$$\frac{\begin{array}{c}\phi_1\\ \vdots \\ \phi_n\end{array}}{\psi} \quad \text{and} \quad \frac{\begin{array}{c}\phi_1\\ \vdots \\ \phi_n\\ \psi\end{array}}{\theta}$$

we can derive:

$$\frac{\begin{array}{c}\phi_1\\ \vdots \\ \phi_n\end{array}}{\theta}$$

4.3.4 Axioms

We now present the axioms of the logic for 'B'. In Gardiner's draft papers there are no antecedents involving the meta-predicates Var, Term and Formula in the axioms

except those concerning the three meta-predicates themselves. This omission makes things very unsafe as we can form things such as $(A\&B) + (C\&D)$, $\forall a.a$ or even $\forall(A + B).100$. I have added the missing antecedents to the following presentation of the axioms.

These axioms define the meanings of the six meta-predicates and the single additional meta-function-symbol. These meta-predicates are chosen in order that the common form of side-condition invoked for an inference rule in the predicate calculus can be written in the meta-language and therefore be built into the inference rule represented by the sequent. Most such side-conditions involve the non-freeness of variables in certain expressions.

Object language syntax classes

1.1) $\dfrac{}{\text{Var } a} \quad \dfrac{}{\text{Var } b}$

1.2) $\dfrac{}{\text{Term } 0}$

1.3) $\dfrac{\text{Var } X}{\text{Term } X}$

1.4) $\dfrac{\begin{array}{c}\text{Term } X \\ \text{Term } Y\end{array}}{\text{Term } X + Y}$

$$1.5)\quad \frac{\text{Term } X \quad \text{Term } Y}{\text{Formula } X = Y}$$

$$1.6)\quad \frac{\text{Formula } X \quad \text{Formula } Y}{\text{Formula } X \& Y}$$

$$1.7)\quad \frac{\text{Var } X \quad \text{Formula } Y}{\text{Formula } \forall X.Y}$$

Equivalence

$$2.1)\quad \frac{}{A == A}$$

$$2.2)\quad \frac{A == B}{B == A}$$

$$2.3)\quad \frac{A == B \quad B == C}{A == C}$$

$$2.4)\quad \frac{\text{Term } A \quad \text{Term } B \quad \text{Term } C \quad \text{Term } D \quad A == B \quad C == D}{A + C == B + D}$$

$$\text{2.5)}\quad \frac{\begin{array}{cc}\text{Term } A & \text{Term } B\\ \text{Term } C & \text{Term } D\\ A == B & C == D\end{array}}{A = C == B = D}$$

$$\text{2.6)}\quad \frac{\begin{array}{cc}\text{Formula } A & \text{Formula } B\\ \text{Formula } C & \text{Formula } D\\ A == B & C == D\end{array}}{A\&C == B\&D}$$

$$\text{2.7)}\quad \frac{\begin{array}{cc}\text{Var } A & \text{Var } B\\ \text{Formula } C & \text{Formula } D\\ A == B & C == D\end{array}}{\forall A.C == \forall B.D}$$

Non-Freeness

$$\text{3.1)}\quad \frac{}{a \setminus b} \qquad \frac{}{b \setminus a}$$

$$\text{3.2)}\quad \frac{\text{Var } X}{X \setminus 0}$$

$$\text{3.3)}\quad \frac{\begin{array}{cc}\text{Var } A & \text{Var } B\\ A == B & C == D\\ \multicolumn{2}{c}{A \setminus C}\end{array}}{B \setminus D}$$

$$\text{3.4)}\quad \frac{\begin{array}{c}\text{Var } X\\ \text{Term } S \quad \text{Term } T\\ X \setminus S \quad X \setminus T\end{array}}{X \setminus S + T}$$

$$\text{3.5)}\quad \frac{\begin{array}{c}\text{Var } X\\ \text{Term } S \quad \text{Term } T\\ X \setminus S \quad X \setminus T\end{array}}{X \setminus S = T}$$

$$\text{3.6)}\quad \frac{\begin{array}{c}\text{Var } X\\ \text{Formula } P \quad \text{Formula } Q\\ X \setminus P \quad X \setminus Q\end{array}}{X \setminus P \& Q}$$

$$\text{3.7)}\quad \frac{\begin{array}{c}\text{Var } X\\ \text{Var } Y\\ \text{Formula } P\\ X \setminus P\end{array}}{X \setminus \forall Y.P} \qquad \frac{\begin{array}{c}\text{Var } X\\ \text{Formula } P\end{array}}{X \setminus \forall X.P}$$

Entailment

$$\text{4.1)}\quad \frac{\begin{array}{ccc}\text{Formula } G_1 & \text{Formula } H_1 & G_1 == H_1\\ \vdots & \vdots & \vdots\\ \text{Formula } G_n & \text{Formula } H_n & G_n == H_n\\ \text{Formula } X & \text{Formula } Y & X == Y\\ \multicolumn{3}{c}{G_1, \ldots, G_n \vdash X}\end{array}}{H_1, \ldots, H_n \vdash Y}$$

$$\frac{\begin{array}{c}\text{Formula } H_1 \\ \vdots \\ \text{Formula } H_n\end{array}}{H_1, \ldots, H_n \vdash H_i} \quad \text{Where } 1 \leq i \leq n. \tag{4.2}$$

$$\frac{\begin{array}{c}\text{Formula } H_1 \cdots \text{Formula } H_n \\ \text{Formula } G_1 \cdots \text{Formula } G_m \\ \text{Formula } X \\ H_1, \ldots, H_n \vdash G_1 \\ \vdots \\ H_1, \ldots, H_n \vdash G_m \\ G_1, \ldots, G_m \vdash X\end{array}}{H_1, \ldots, H_n \vdash X} \tag{4.3}$$

Safe Substitution

$$\frac{\begin{array}{ccc}\text{Var } V_1 & \text{Term } T_1 & V_1 \setminus E \\ \vdots & \vdots & \vdots \\ \text{Var } V_n & \text{Term } T_n & V_n \setminus E\end{array}}{[V_1 := T_1, \ldots, V_n := T_n]E == E} \tag{5.1}$$

$$\frac{\begin{array}{cc}\text{Var } V_1 & \text{Term } T_1 \\ \vdots & \vdots \\ \text{Var } V_n & \text{Term } T_n\end{array}}{[V_1 := T_1, \ldots, V_n := T_n]V_i == T_i} \quad \text{Where } 1 \leq i \leq n. \tag{5.2}$$

$$\frac{\begin{array}{cccccc}\text{Var } V_1 & \text{Var } U_1 & \text{Term } T_1 & \text{Term } S_1 & V_1 == U_1 & T_1 == S_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \text{Var } V_n & \text{Var } U_n & \text{Term } T_n & \text{Term } S_n & V_n == U_n & T_n == S_n \\ & & & E == F & & \end{array}}{[V_1 := T_1, \ldots, V_n := T_n]E == [U_1 := S_1, \ldots, U_n := S_n]F} \tag{5.3}$$

$$\frac{\begin{array}{cc}\text{Var } V_1 & \text{Term } T_1\\ \vdots & \vdots\\ \text{Var } V_n & \text{Term } T_n\\ \text{Term } A & \text{Term } B\end{array}}{\begin{array}{l}[V_1 := T_1, \ldots, V_n := T_n](A + B) ==\\ [V_1 := T_1, \ldots, V_n := T_n]A + [V_1 := T_1, \ldots, V_n := T_n]B\end{array}} \quad 5.4)$$

$$\frac{\begin{array}{cc}\text{Var } V_1 & \text{Term } T_1\\ \vdots & \vdots\\ \text{Var } V_n & \text{Term } T_n\\ \text{Term } A & \text{Term } B\end{array}}{\begin{array}{l}[V_1 := T_1, \ldots, V_n := T_n](A = B) ==\\ [V_1 := T_1, \ldots, V_n := T_n]A = [V_1 := T_1, \ldots, V_n := T_n]B\end{array}} \quad 5.5)$$

$$\frac{\begin{array}{cc}\text{Var } V_1 & \text{Term } T_1\\ \vdots & \vdots\\ \text{Var } V_n & \text{Term } T_n\\ \text{Formula } A & \text{Formula } B\end{array}}{\begin{array}{l}[V_1 := T_1, \ldots, V_n := T_n](A\&B) ==\\ [V_1 := T_1, \ldots, V_n := T_n]A\&[V_1 := T_1, \ldots, V_n := T_n]B\end{array}} \quad 5.6)$$

$$\frac{\begin{array}{cccc}\text{Var } Y & \text{Formula } F & & \\ \text{Var } V_1 & \text{Term } T_1 & Y \setminus V_1 & Y \setminus T_1\\ \vdots & \vdots & \vdots & \vdots\\ \text{Var } V_n & \text{Term } T_n & Y \setminus V_n & Y \setminus T_n\end{array}}{[V_1 := T_1, \ldots, V_n := T_n]\forall Y.F == \forall Y.[V_1 := T_1, \ldots, V_n := T_n]F} \quad 5.7)$$

4.3.5 Additional Rules

The additional rules formalise certain parts of the object language which we assume we will always need. These rules presume we have the predicate symbol "=", the logical operator symbols "&" and "⇒" and the quantifier symbol "∀" and give special meaning to these symbols.

The first four of these rules are built into the 'B' system as the Conjunction, Deduction, Generalisation and Equality rules.

In a natural deduction system the inference rules tend to come in pairs. One rule in a pair will serve to introduce a symbol and the other to eliminate it. In the 'B' system, and in Gardiner's logic, this pattern is not followed, and for some symbols we only have one of the expected pair of inference rules. Rule 6.1) is the &-introduction rule but we don't include the corresponding elimination rule. Rules 6.2) and 6.5) are the ⇒-introduction and ⇒-elimination rules. Rules 6.3) and 6.6) are the ∀-introduction and ∀-elimination rules. Rule 6.7) is important as it is the one used to justify the rewriting of terms by equality rules[5].

Gardiner's draft paper does use the three object language syntax meta-predicates in antecedents, but not in a complete fashion. As with the axioms I present the rules with such antecedents added as necessary to prevent unsound inferences.

[5]This may alter in future versions

$$\frac{\begin{array}{c}\text{Formula } A\\ \text{Formula } B\end{array}}{A, B \vdash A\&B} \qquad 6.1)$$

$$\frac{\begin{array}{c}\text{Formula } H_1\\ \vdots\\ \text{Formula } H_n\\ \text{Formula } P\\ \text{Formula } Q\\ H_1, \ldots, H_n, P \vdash Q\end{array}}{H_1, \ldots, H_n \vdash P \Rightarrow Q} \qquad 6.2)$$

$$\frac{\begin{array}{cc}\text{Var } X & \text{Formula } P\\ \text{Formula } H_1 & X \setminus H_1\\ \vdots & \vdots\\ \text{Formula } H_n & X \setminus H_n\\ H_1, \ldots, H_n \vdash P & \end{array}}{H_1, \ldots, H_n \vdash \forall X.P} \qquad 6.3)$$

$$\frac{\text{Term } T}{\vdash T = T} \qquad 6.4)$$

$$\frac{\begin{array}{c}\text{Formula } P\\ \text{Formula } Q\end{array}}{P, P \Rightarrow Q \vdash Q} \qquad 6.5)$$

$$\frac{\begin{array}{c}\text{Var } X\\ \text{Formula } P\\ \text{Term } T\end{array}}{\forall X.P \vdash [X := T]P} \qquad 6.6)$$

$$\frac{\begin{array}{c}\text{Var } V_1 \quad \text{Term } T_1 \quad \text{Term } S_1 \\ \vdots \qquad \vdots \qquad \vdots \\ \text{Var } V_n \quad \text{Term } T_n \quad \text{Term } S_n \\ \text{Formula } P\end{array}}{\begin{array}{c}T_1 = S_1, \ldots, T_n = S_n, [V_1 := T_1, \ldots, V_n := T_n]P \vdash \\ [V_1 := S_1, \ldots, V_n := S_n]P\end{array}} \quad 6.7)$$

4.3.6 Derived Rules

The 'B' logic, with the additional rules described in §4.3.5, can be used to justify some powerful derived rules. The derived rules are important as they provide powerful methods of proof, but they are not as straightforward as might be hoped. The major problem is that the presence of meta-variables complicates the use of these rules in a non-trivial fashion.

Rewriting

As already mentioned, we can use rule 6.7) to justify term rewriting. In general an equality rewrite rule will be of the form:

$$\frac{\begin{array}{c}\sigma_1 \\ \vdots \\ \sigma_n\end{array}}{\tau_1, \ldots, \tau_m \vdash \phi = \psi}$$

If, under the current hypotheses, we can use this sequent to prove $\vdash \phi = \psi$ then we can use the equality $\phi = \psi$ for rewriting using rule 6.7). In order to prove $\vdash \phi = \psi$ we

must derive proofs of the meta-formula antecedents and also the antecedents of the entailment. The former are dealt with by our rules for deriving new sequents while the latter are dealt with by rules 4.1) to 4.3).

Suppose we have an expression ξ containing a free sub-expression[6] ϕ' which is an instance of ϕ. Then we can rewrite ξ by replacing the sub-expression ϕ' by the corresponding instance of ψ.

Suppose that the matching substitution of ϕ' and ϕ is ρ. Then ϕ' is $\rho(\phi)$, and using our second rule for deriving sequents, page 138, we can derive a new sequent, and hence a proof of $\vdash \rho(\phi) = \rho(\psi)$. Thus we have a proof of $\vdash \phi' = \psi'$ where ψ' is $\rho(\psi)$.

We now construct a new expression from ξ, identical to ξ except that where the sub-expression ϕ' appeared in ξ we now have an object language variable, z say. We choose z such that it occurs nowhere in ξ. We will write this constructed expression as $\xi(z/\phi')$. We then have this instance of rule 6.7):

$$\frac{\begin{array}{c}\text{Var } z \quad \text{Term } \phi' \quad \text{Term } \psi' \\ \text{Formula } \xi(z/\phi')\end{array}}{\phi' = \psi', [z := \phi']\xi(z/\phi') \vdash [z := \psi']\xi(z/\phi')}$$

The meta-level antecedents should, *a priori*, be easy to prove as they only involve simple meta-predicates. The first hypothesis of the entailment is the equality we are

[6] A sub-expression is said to be free in an expression if its free variables, at their occurrences in the sub-expression, are free in the expression. This means that these free variables of the sub-expression are not bound by any enclosing quantification in the expression.

assuming we have proved. The intention is that for the second hypothesis we should have $[z := \phi']\xi(z/\phi') == \xi$. We should also have $[z := \psi']\xi(z/\phi') == \xi'$, where ξ' is some formula which we wish to prove. Thus given a proof of ξ we can use this sequent to prove ξ' which is obtained by rewriting the sub-expression ϕ' to ψ'.

In order to succeed we must be able to prove that these two equivalences hold. We do this by use of the safe substitution axioms 5.1) to 5.7). Various problems can arise in attempting these proofs: we will consider the proof of the first equivalence, as the same problems occur with both.

- If there is a meta-variable X in $\xi(z \setminus \phi')$ then we will, after repeated use of rules 5.4) to 5.7), be left with either rule 5.1) or 5.2) to apply. Rule 5.2) is not appropriate so the only rule we can use to prove the equivalence is 5.1). We will therefore have the following instance of this rule:

$$\frac{\text{Var } z \quad \text{Term } \phi' \quad z \setminus X}{[z := \phi']X == X}$$

  Thus we must assume $z \setminus X$ in order to prove the equivalence. A similar assumption will be generated for each meta-variable occurring in $\xi(z \setminus \phi')$. These assumptions will have to appear in the proof constructed using this rewrite rule and hence in any sequent derived from it. For any use of such a derived sequent we could of course redo the proof of the sequent using a different object variable so there is a sense in which these conditions are unnecessary. However, if we

are to stick to the strict interpretation of the axioms we cannot remove these conditions[7].

- If there is a quantification $\forall X.Y$ in $\xi(z \setminus \phi')$, where z occurs in Y, then we will eventually attempt to use the following instance of axiom 5.7):

$$\frac{\text{Var } X \quad \text{Formula } Y \qquad \text{Var } z \quad \text{Term } \phi' \quad X \setminus z \quad X \setminus \phi'}{[z := \phi']\forall X.Y == \forall X.[z := \phi']Y}$$

  Most of the antecedents of this sequent can be proved easily, but we are left with $X \setminus z$ and $X \setminus \phi'$. The first of these will have to assumed unless X is actually an object variable, in which case we could use axiom 3.1). The second can only be proved if we add assumptions of the form $X \setminus A$ for each meta-variable occurring in ϕ'.

Thus the use of this derived rewriting rule may generate many non-freeness antecedents. The basic problem is that the presence of meta-variables means we cannot actually decide whether or not ϕ', or ψ' at the same position, is a free sub-expressions of ξ. The best we can do is derive sufficient conditions for this to be true. These conditions must then appear as assumptions in the final proof.

The 'B' system itself makes no distinction between object language terms and formulae. Thus it can use its equality rewriting rules to rewrite formulae as well as

[7]Paul Gardiner has recently reformulated the rewriting axioms to remove these conditions.

terms[8]. This cannot be justified by Gardiner's logic for 'B'. If a form of type checking were added to the 'B' system then it would be necessary to add extra rules concerning equivalence in order to get rewriting rules for formulae.

"Jokerisation"

In the 'B' system, all the rules input by the user are processed after parsing to turn free object variables into meta-variables. Thereafter, the entire system only sees meta-variables. The process of going from the user's rule to the meta-variable form involves the addition of certain extra meta-predicate antecedents, which are all non-freeness conditions.

A similar, but more limited, effect can be achieved by application of the axioms and rules 6.3) and 6.6). This form of "jokerisation" is not that implemented by the 'B' system as it is impossible to affect the antecedents of the rule by this method. The solution now chosen by Paul Gardiner is to consider only a limited form of "jokerisation". His solution is to only apply the process to rules containing no meta-variables and with no antecedents, otherwise if the user wants a schematic rule then they should use meta-variables explicitly.

The non-freeness antecedents arise because the only way to replace object variables is to use axioms and rules concerning safe substitution, which involves non-freeness

[8]In the LCF system the rewriting rules for formulae are justified by equivalence between formulae, not by equality.

conditions.

The basic idea is to generalise over all the free object variables using rule 6.3), then specialise them all to meta-variables by use of rule 6.6). As can be seen from rule 6.3), we can only generalise over object variables that are not free in the antecedents of the hypothesis.

Thus, for instance, the meta-formula $\vdash a = b$ is generalised to $\vdash \forall a.\forall b.a = b$. From this, using rule 6.6), we get $[a := A]\forall b.a = b$. This reveals the first restrictions we must impose. In order to push the substitution inside the quantification, we must have $b \setminus a$ and $b \setminus A$. The first of these is trivial, but the second must be included as an antecedent of the rule we are deriving.

Assuming we have this antecedent we can proceed until we get $\forall b.A = b$. We then specialise again to get $[b := B](A = b)$ which gives $[b := B]A = B$. This reveals the second restriction we must impose, that is $b \setminus A$. This must also be present as an antecedent of the rule we are deriving. This restriction is the reason why we choose to ignore rules already containing meta-variables as we would have to generate similar antecedents for them all.

The form of jokerisation provided by the derived rule is much more limited than that employed by the 'B' system, and it would seem to be of little practical use.

4.3.7 Relationship to 'B'

We now examine the relationship between the Gardiner's logic for 'B' and the 'B' system.

Sets of rules

Some of the axioms involve constructs of the form $H_1, \ldots, H_n$ where we mean this to refer to some number n of hypotheses. These axioms are intended to represent the set of all axioms for any n. This occurs most often in rules referring to the antecedents in entailment. There is a problem associated with rules of this kind: how is a user to enter them?

The problem is not insoluble but it leads to a very complex syntax for the case of entailment if a general solution is sought. We can avoid the problem, if we build in all the rules of this form. For the axioms there is no problem as the functions implementing the rules can simply be defined to take lists of arguments. But for user defined rules the problem is serious and this is one reason why rules 6.2) and 6.3) are added as additional rules.

In the 'B' system one cannot refer to the hypotheses of an entailment[9] in user provided rules. Thus the only rules that can do so are the rules built into the system. As we are assuming we have the seven additional rules we can take the same approach

[9]In a more recent version of 'B' there is a somewhat limited ability to refer to the hypotheses in entailment, but it doesn't solve all the problems

and not allow the user access to the hypotheses of an entailment.

There is a further point about those rules defined over all members of a particular class of symbol. In the logic we took representatives of the symbol classes in specifying the axioms. If we have a lexical means of deciding the class of a symbol then we can generate the appropriate axioms for it automatically.

Object language syntax classes

As already noted, the 'B' system does not perform any sort of syntax check on object language expressions. In the Gardiner logic for 'B' some checks are enforced by the use of the meta-predicates Var, Term and Formula in the antecedents of the additional rules. Notably, in rules 6.4) and 6.6) we have the antecedent Term T.

In the 'B' system, meaningless expressions can be input and can then be passed around by the proof system for quite a while before they cause the proof to fail. A notable such case is that involving expressions of the form $a \setminus X$, where there is no axiom that can help prove this if it is produced as a goal. The 'B' system regards all non-freeness meta-formulae concerning meta-variables as true! The ability of the 'B' system to accept such meaningless input as $\forall(A + B).0$ is, in my view, one of the major problems with the system. This lack of concern about the type of object languages expressions causes serious problems when one uses the rewriting rules.

In Gardiner's presentation of the additional rules, and my extension of it, the symbol "=" is an object language predicate. Thus its operands must be object language

terms. Similarly the left operand of the meta-function symbol ":=" is expected to be an object language variable, and its right an object language term. These restrictions are not enforced by the 'B' system. This is because the 'B' system fails to make the important distinction between object language terms and formulae.

User input

In the 'B' system there is no distinction between the sequent's antecedents and the hypotheses of the entailment in the consequent. This causes rules like 6.3), which generate non-freeness conditions on the hypotheses, to generate non-freeness conditions on the antecedents of the sequent. It can therefore generate sub-goals like $x \setminus (a \setminus B)$ which don't mean anything. We will allow the user to enter sequents as a list of meta-formulae for the antecedents and a single formulae for the consequent. These meta-formula can of course be applications of the entailment meta-predicate.

4.3.8 The Implementation

We now describe the Miranda implementation of the core of a theorem prover based on the logic described in §4.3.2 to §4.3.5.

User Input

We again used the method of generating parsers in Miranda described in [Lon87a] to derive the parser for meta-formulae, formulae and terms. The grammar used

allows the parsing of expressions of all such forms. The grammar does not refer to any particular symbols for any of the symbol classes, because the allowed symbols are selected by the user. The grammar mentions ten different classes of symbol: quantifiers, logical infix, logical prefix, predicate infix, predicate prefix, function infix, function prefix, meta-variables, variables and constants. We enforce certain syntactic restrictions on the symbols of these different classes.

We distinguish four types of symbols: quantifiers and infix symbols, prefix symbols and variables, meta-variables and finally constants. The parser is given information about the allowed symbols for each of the first seven symbol classes. Thus a symbol that is syntactically of the first type above should appear either in the list of allowed quantifier symbols or in one of the three lists of allowed infix symbols. If it does not it is an unknown symbol and the parse will fail. If a symbol is syntactically of the second type, but does not appear in one of the three lists of prefix symbols, then it is assumed to be a variable. Anything obeying the appropriate syntactic conditions will be taken as a meta-variable or constant, except for the three meta-predicates Formula, Term and Var which would otherwise be taken as meta-variables.

This first phase of the parse does not take into account the relative precedences of the different symbols, except for that between different classes of symbol. We therefore have a second stage which involves reparsing those portions of the parse tree involving symbols that may be affected. These are the logical and function infix

symbols: the predicate infix operators need no relative precedence as it makes no sense to associate them.

Thus the complete parsing function takes lists of the allowed symbols for seven of the symbol classes and precedence information for two of the symbol classes. The parser returns a parse tree of type `structure` which encompasses all the desired forms of expression. Various functions are provided to test whether what the parser has returned is of the form desired. For instance the function `maybe_meta_formula` tests whether the parsed `structure` is a meta-formula. These functions are all prefixed by `maybe` as the presence of meta-variables means that the best they can do is give a negative result or a qualified positive result. Thus $A\&B$ would be accepted as a formula even though in a context where we had Term A this would not be true.

A sequent will be input as simply a list of meta-formulae for the antecedents and a single meta-formula for the consequent.

Sequents

The sequents, which represent the inference rules, are implemented as data structures. This is different from the approach used in systems such as LCF and in §4.1, where the inference rules are implemented as functions. A sequent can simply be regarded as a pairing of a list of meta-formulae, that represent the antecedents, and a single meta-formula, that represents the consequent. However, this is not quite versatile enough. This is because any sequent-creating functions will always face the possibility

of failing through inappropriate application. Thus, unless we simply want to call Miranda's **error** function and terminate execution, we need a better answer. The obvious one is to use a simple algebraic type which has only two constructors. One constructor will pair the antecedents and the consequent needed for a sequent and the other will allow the construction of an error representation of the correct type. We choose an error representation consisting of a list of strings and a list of structures; the intention is that interleaving the strings with the pretty-printed structures will result in an appropriate error message. This is a very simple form of formatted string output. We will call these error representations *bad sequents.*

The only drawback associated with using an algebraic type to represent sequents is that any applications of the constructors would produce values of the sequent type. We would like to ensure more security than this, therefore the sequent type is actually an abstract type whose implementation type is the algebraic type just described. We then include in the signature of the abstract type all the basic sequent creation and manipulation functions we will need. An abbreviated form of the abstract type definition is given in Figure 7. We will discuss each of these functions in details.

create_sequent This is the function used to create user defined sequents. In order to create a sequent, we need the list of antecedents and the consequent. We also need a name for the sequent, so that it can be referred to by the user. The only condition we need impose is that the structures provided as antecedents and

```
abstype sequent with
     create_sequent ::
          [char] -> [structure] -> structure -> sequent
     instance ::
          substitution structure -> sequent -> sequent
     consequent ::  sequent -> structure
     antecedents ::  sequent -> [structure]
     sequent_name ::  sequent -> [char]
     bad_sequent ::  [[char]] -> [structure] -> sequent
     sequent_OK ::  sequent -> bool
     pp_sequent ::
          (structure -> [sys_message]) -> sequent ->
          [sys_message]
     axiom_1_1 ::  structure -> sequent
     :
     axiom_5_7 ::  structure -> sequent
     rule_1 ::  structure -> sequent
     :
     rule_7 ::  structure -> sequent

sequent == sequent'

sequent' ::= Sequent [char] [structure] structure |
             Bad_sequent [[char]] [structure]
```

Figure 7: Sequents

consequent be meta-formulae. If this is not the case a bad sequent is created.

instance In order to take instances of sequents we need a representation for substitutions and a function for applying them. These are provided by my library script of unification and matching functions developed for my work on type checking [Lon87b]. We can simply apply the substitution to the antecedents and consequent of the sequent to get the required instance.

consequent We will often wish to examine the consequent of a sequent. For instance, for checking the suitability of a sequent when attempting a proof through a subgoaling technique.

antecedents We will similarly often wish to examine the antecedents of a sequent.

sequent_name The sequent name is needed in order that the user may refer to the sequent, for example, when selecting the next rule to apply in a proof.

bad_sequent When the conditions associated with a sequent creation function are not satisfied, we do not want to have to call Miranda's **error** function. Instead we allow the construction of a representation of the error. This representation consists of a list of strings and a list of structures which can be formatted in a simple fashion.

sequent_OK We need to be able to check whether we have a sequent or the representation of some error.

pp_sequent We include a simple pretty-printer for sequents. It must be given the current pretty-printer for structures which will depend on the symbols currently allowed and their precedences. This will also print the error message represented by a bad sequent.

axiom_1 ... The axioms of the system present an interesting problem. This is because the statements of the axioms are doubly schematic in that each axiom stands for an entire set of axioms, one for each symbol in the symbol class the axiom applies to. Thus, at first glance, it seems they should be represented as functions from symbols to sequents. This turns out to be rather complex, and, as in any use of these axioms we will know the desired consequent, we in fact implement them as functions that take the desired consequent and return a sequent. They therefore not only select the version of axiom appropriate for the symbol in the consequent, but also take the appropriate instance of the schematic form of the axiom.

There is a further consideration for four of the axioms, 2.3), 3.3), 4.1) and 4.3). These are the four axioms that do not obey the sub-expression rule in that there are meta-variables in their antecedents that do not occur in their consequent. These axioms must therefore be given more information before they can construct a sequent. These axioms are therefore implemented by functions of a slightly more complex type:

```
structure -> structure -> sequent
```

Applying any of these axiom functions to expressions to which they do not apply results in the construction of a bad sequent. As an example of the ease with which the axioms can be implemented the following shows the implementation of axiom 3.3). The type **structure** of meta-language expressions is a simple algebraic type with constructors corresponding to each form of expression.

```
axiom_3_3 (ST_non_free s2 s4) (ST_non_free s1 s3)
      = Sequent "Axiom 3.3)" [ST_Var s1,
                              ST_Var s2,
                              ST_equivalent s1 s2,
                              ST_equivalent s3 s4,
                              ST_non_free s1 s3]
                             (ST_non_free s2 s4)

axiom_3_3 s1 s2
      = Bad_sequent ["Axiom 3.3) not applicable to ",
                     " and\n","\n"]
                    [s1,s2]
```

The ease with which the implementation can be compared with the specification is clear.

rule_1 ... The additional rules of the system are implemented in the same manner as the axioms.

The implementations of the axioms and rules in Miranda are sufficiently concise that it is easy to confirm their correctness.

Proofs

Proofs are represented by trees in much the same way as derivations were in §4.1. The leaves of the tree are the assumed meta-formulae and the root of the tree is the conclusion of the proof. Each node of the tree is a meta-formula derived by the application of a sequent to the sub-proofs above that node. The name of the sequent that was applied is attached to each node.

We again use an abstract type whose implementation type is an algebraic type. The algebraic type now has three constructors as we must distinguish assumptions and proofs resulting from the application of an inference rule as well as coping with errors. We shall call these error representations *bad proofs*. We include in the signature of the abstype all the basic proof creation and manipulation functions we will need (there are only three functions that can create proofs). The definition of the abstract type for proofs is given is Figure 8; we now discuss some details of these functions.

assume The simplest form of proof is the assumption of a meta-formula. If the structure is not a meta-formula we construct a bad proof.

apply_sequent We first check that the sequent is not bad and that none of the sub-proofs are bad proofs. Then, if the antecedents of the sequent are present among the conclusions of the sub-proofs, we can derive a proof of the consequent of the sequent. This form of sequent application is justified by our rules for deriving new sequents. This proof step is tagged with the name of the sequent being

```
abstype proof with
     assume ::  structure -> proof
     apply_sequent ::  sequent -> [proof] -> proof
     extract_sequent ::  proof -> sequent
     conclusion ::  proof -> structure
     hypotheses ::  proof -> [structure]
     bad_proof ::  sequent -> [structure] -> proof
     proof_OK ::  proof -> bool
     pp_proof ::  (structure -> [sys_message]) -> proof ->
                  [sys_message]
     show_proof ::  (structure -> [sys_message]) -> proof ->
                    [sys_message]

proof == proof'

proof' ::= Proof [char] [proof'] structure |
           Assume structure                |
           Bad_proof sequent [structure]
```

Figure 8: Proofs

used.

extract_sequent The rules for deriving new sequents also justify this function. Given a proof, we can extract the hypotheses and the conclusion and construct a sequent from them. The hypotheses will form the antecedents of the sequent and the conclusion the consequent.

conclusion We will wish to extract the conclusion of a proof.

hypotheses We also wish to extract the assumptions upon which the proof is based. It should be noted that assumptions cannot be discharged at this level: this is only done at the entailment ($\vdash$) level of proofs.

bad_proof When the application of a proof creating function fails we will wish to construct some representation of the error. The main failure will be that of applying a sequent to an inappropriate set of sub-proofs. We therefore record the sequent and the conclusions of the sub-proofs as our error representation. This turns out to be enough to cope with all possible errors as we can encode any form of message as a bad sequent in this representation.

proof_OK We will need to check that a proof is not an error representation.

show_proof We can show proofs in a more or less readable form: this will also show the error message from a bad proof.

pp_proof This is a more sophisticated form of presentation of a proof. This function is again obtained by mapping `proof` trees into objects from the algebraic type `pretty_printing_tree` as described in §4.1.2. It presents the proof as a centered tree, where the conclusion of a proof step is centered under the sub-proofs. Unfortunately, due the nature of proofs in this system most proofs are too wide for usual display media.

Here again the conciseness of the Miranda code means that correctness of the implementations of the above functions is readily apparent.

Useful Functions

As well as the basic structure of sequents and proofs described above a variety of useful functions were implemented. Some of these would be essential if some form of theorem prover was to be implemented on top of this core system. We will discuss these functions in order of increasing complexity.

- There are two functions, `variables_in` and `meta_variables_in`, which return lists all the object-variables or meta-variables appearing anywhere in an expression.

- The functions `variant` and `meta_variant` return a variant of a variable or meta-variable. These functions take a variable and a list of variables, or meta-variable and list of meta-variables, and return a variant not present in the list.

- We have four functions, `maybe_meta_formula`, `maybe_Formula`, `maybe_Term` and `maybe_Var`, which test the form of an expression. As has already been mentioned they can only return a negative or qualified positive result due to the presence of meta-variables. They are defined recursively on the structure of expressions and make optimistic assumptions about any meta-variables encountered.

- `maybe_non_free :: structure -> structure -> value [structure]`

  The first argument is a variable or meta-variable and the second an expression. This function attempts to find out if the variable, or meta-variable, is non-free in the expression. The presence of meta-variables again means that we can only give a negative or qualified positive result. If the variable is definitely free in the expression it will return `None` and otherwise it will return a list of non-freeness conditions on the meta-variables appearing in the expression that would have to be satisfied for the variable, or meta-variable, to be non-free in the expression.

- `create_sequent' :: [structure] -> structure -> sequent`

  This function is actually defined in the signature of the abstract type of sequents. It creates a sequent from a list of antecedents and a consequent, but first uses the form of the antecedents and consequent to generate extra antecedents. In many cases the context of a meta-variable in these expressions will be enough to decide whether it should satisfy one of the three meta-predicates; Formula, Term

and Var. We use the following constraints, and the fact that any expression satisfying Var will also satisfy Term, when deriving these extra antecedents.

- In the expression $X \setminus A$ we can expect X to satisfy Var.
- In the expression $A == B$ we can expect both A and B to satisfy Term or to both satisfy Formula.
- In the expression $H_1, \ldots, H_n \vdash A$ we can expect all the $H_1, \ldots, H_n$ and A to satisfy Formula.
- In the expression $\forall X.A$ we can expect X to satisfy Var and A to satisfy Formula.
- In the expression $A \& B$ we can expect both A and B to satisfy Formula.
- In the expression $A = B$ we can expect both A and B to satisfy Term.
- In the expression $A + B$ we can expect both A and B to satisfy Term.
- In the expression $[V_1 := T_1, \ldots, V_n := T_n]$ we can expect $V_1, \ldots, V_n$ to satisfy Var and $T_1, \ldots, T_n$ to satisfy Term.

Of course these conditions derive must be consistent and also be consistent with any of the given antecedents involving Formula, Term or Var. If the derived conditions are not consistent then we simply form a sequent from what we were given.

This function makes the entry of sequents much easier because it is extremely tedious entering conditions for all the meta-variables used in the statement of a sequent.

- `rewrites :: structure -> structure -> structure -> structure -> ([structure], [(structure,substitution structure,[structure])])`

This function is used to generate all the possible rewrites of an expression for a given equality. This function is best explained by reference to §4.3.6. The first argument is the object variable z, the second argument is the left hand side of the equality, ϕ, and the third argument is the right hand side of the equality, ψ. The fourth argument is the expression ξ.

The function searches, breadth first, for all the sub-expressions ϕ' of ξ that are instances of ϕ. Let us suppose that the matching substitution of ϕ and ϕ' is ρ and that $\rho(\psi)$ is ψ'. Then if ϕ' is a free sub-expression of ξ and ψ' at the same position in ξ would be a free sub-expression of ξ, we can construct an ordered triple as follows. The first component is the structure $\xi(z/\phi')$ obtained by replacing ϕ' by z in ξ, the second is the matching substitution ρ, and the third is the list of non-freeness conditions that are required for ϕ' and ψ' to be free sub-expressions of ξ.

```
rewrites var l r s
      = (conditions,concat (rewrites' s l r var []))
        where
        conditions = map f (meta_variables_in s)
        f mv = ST_non_free var mv

rewrites' (ST_meta_variable name) l r var bound
      = [rewrite (ST_meta_variable name) l r var bound]
rewrites' (ST_quantifier name s1 s2) l r var bound
      = rewrite (ST_quantifier name s1 s2) l r var bound:
        join_subtrees
                  (generate_subtrees var (Op_quan name) [s1]
                                         [s2] l r (s1:bound))
                  , maybe_Var s1
      = [rewrite (ST_quantifier name s1 s2) l r var bound]
                  , otherwise
rewrites' s l r var bound
      = rewrite s l r var bound:
        join_subtrees
                  (generate_subtrees var (get_operator s) []
                                         (get_operands s) l r bound)

rewrite s l r var bound
      = [] ,   match_succeeds result  conditions_value = None
      = [(var,subst,conditions)] , otherwise
        where
        result = match s l
        Matches subst = result
        r' = instantiate subst r
        conditions_value
                  = foldr combine_conditions (Value []) (qmap f bound)
        Value conditions = conditions_value
        f v = combine_conditions (maybe_non_free v s) (maybe_non_free v r
```

Figure 9: Rewrites

```
generate_subtrees var op head (x:tail) l r bound
      = map (map f) sub_structure_rewrites :
        generate_subtrees var op (head ++ [x]) tail l r bound
        where
        sub_structure_rewrites = rewrites' x l r var bound
        f (s,subst,conditions)
                  = (reconstruct_expression op (head ++ s :  tail),
                     subst,conditions)
generate_subtrees var op head [] l r bound = []

join_subtrees = foldr (map2p id id (++)) []
```

Figure 10: Subtrees

The second element of the pair returned by the function is the list of all these tuples found by searching ξ, while the first is the list of non-freeness conditions required for the variable z.

Using this function one could write the portion of a sub-goal package that presents to the user all the possible rewrites of a goal. Its implementation is shown in Figure 9 and Figure 10. The result of the `rewrites` function is a pairing of a list of non-freeness conditions and a list of possible rewrites. The subsidiary function `rewrites'` returns a list of list of rewrites, each list contains all the rewrites for a given depth in the expression. To do this its needs to take the expression apart, search the sub-expressions for rewrites and rebuild the expression with each of the sub-expression rewrites. To do this we use the functions `generate_subtrees` and `join_subtrees`. The latter is defined in terms of a binary `map` function that applies default functions when its argument

lists are of different lengths.

- `match_axiom :: (structure -> sequent) -> (sequent,[structure])`

  This function can be used to match all but four of the axioms against a goal. In general when a sequent is matched against a goal we get not only the appropriate instance of the sequent but also a list of the meta-variables appearing in the antecedents that we may wish to instantiate. These meta-variables only occur in sequents that break the sub-expression condition. Thus for all the axioms this function applies to the second element of the result tuple will be an empty list.

- We also require separate functions for matching the four axioms `match_axiom_2_3`, `match_axiom_3_3`, `match_axiom_4_1` and `match_axiom_4_3`, as they break the sub-expression condition so that we must choose meta-variables to appear in the antecedents. These meta-variables will be the ones that we may wish to instantiate. For the first two axioms the functions need only choose a single meta-variable that does not occur in the desired consequent. The function for axiom 4.1) must choose more than one meta-variable but the number required is defined by the form of the desired consequent. Axiom 4.3) presents a bigger problem as even the number of meta-variables needed for the antecedents is unknown and so we must be given this information as an extra argument to the

function.

- `match_rule :: structure -> (sequent,[structure])`

  This is identical to the function for axioms as all the rules obey the sub-expression condition.

- ```
  match_sequents :: [sequent] -> structure ->
                    [(sequent,[structure])]

  match_sequent :: structure -> sequent ->
                   (sequent,[structure])
  ```

  These functions are more interesting than those above as they perform sequent matching for the user defined sequents. The first is defined in the obvious way in terms of the second. A sequent matches a desired consequent if the desired consequent is an instance of the consequent of the sequent. If it is, then we return the appropriate instance of the sequent. We find the meta-variables occurring in the antecedents of this resulting sequent that do not appear in the consequent, these are the meta-variables we may wish to instantiate. If the sequent does not match we return a bad sequent, all such bad sequents are removed from the list of results of the first function.

- We have four functions corresponding to four of the additional rules, 6.1), 6.4), 6.5) and 6.6), which match against a goal at the entailment level using an instance of axiom 4.3). This form of sequent matching is necessary as these
  ```

additional rules are stated as sequents where the number of hypotheses in the entailment in the consequent of the sequent is given explicitly. Thus any use of these rules must be done in association with axiom 4.3).

We proceed as follows. Given a desired consequent which involves entailment, we then look at the conclusion of the entailment, call this ϕ'. We then look for a rule whose consequent is an entailment with conclusion an expression ϕ where ϕ' is an instance of ϕ. Next, we apply the matching substitution to get an instance of the rule. We then create an instance of axiom 4.3) where the consequent is the desired consequent and the last antecedent is the consequent of the instance of the rule. The other antecedents are constructed accordingly. As an example we will use rule 6.1). Suppose the desired consequent is of the form:

$$F_1, \ldots, F_k \vdash D\&E$$

for some entailment with k hypotheses whose conclusion is a conjunction. We get, as the third element of the results tuple, this instance of rule 6.1):

$$\frac{\begin{array}{c}\text{Formula } D \\ \text{Formula } E\end{array}}{D, E \vdash D\&E}$$

We then get, as the first element of the results tuple, the following instance of axiom 4.3):

$$\frac{\begin{array}{c}\text{Formula } F_1 \ldots \text{Formula } F_k \\ \text{Formula } D \quad \text{Formula } E \\ \text{Formula } D\&E \\ F_1, \ldots, F_k \vdash D \\ F_1, \ldots, F_k \vdash E \\ D, E \vdash D\&E\end{array}}{F_1, \ldots, F_k \vdash D\&E}$$

The second element of the results tuple is the list of meta-variables we may wish to instantiate, which in this case is empty.

Now we can see that the above instance of axiom 4.3) does indeed match the desired consequent and so can be used to generate some sub-goals, but notice also that this instance has been chosen such that the last of the antecedents will be matched by the instance of rule 6.1) given above.

Only rule 6.5) produces a meta-variable that may be matched, as in all the other cases each of the meta-variables appears on the right of the entailment in the consequent.

- `match_entailment_sequents :: [sequent] -> structure -> [(sequent,[structure],sequent)]`
  `match_entailment_sequent :: sequent -> structure -> (sequent,[structure],sequent)`

These two function perform a similar matching at the entailment level for user defined sequents.

4.3.9 Sub-goaling

We now discuss some of the issues and problems that arise when the implementation of a sub-goal package for this logic is considered.

The major issue is the need to automate the proof of sub-goals concerning the simple meta-predicates Formula, Term, Var, \ and ==. This is because proofs involving the axioms for these meta-predicates are essentially trivial and are not what the user should spend time proving. However they are not as easy to automate as might be supposed, because it is difficult to ensure the termination of a function designed to automatically proof such goals.

There are no problems concerning axioms 1.1) to 1.7) as they all obey the the sub-expression property and also always produce simpler sub-goals. Thus we can automate the proof of goals involving Formula, Term and Var.

The axioms 2.1) to 2.7) for equivalence cause greater problems. The last four axioms, 2.4) to 2.7), cause no problems as they all obey the sub-expression condition and also produce simple sub-goals. Axiom 2.1) is also not a problem as it either succeeds or fails, but a simple attempt to automate 2.2) could easily lead to non-termination as the sub-goal is no simpler than the consequent. The transitivity axiom, 2.3), causes more problems because it does not obey the sub-expression condition and there is no obvious way to automatically instantiate the meta-variable B.

I think these problems can be solved but the resulting implementation would be

very slow. The idea is to use axioms 2.2) and 2.3) in a forward direction to derive proofs of all possible equivalences from the current hypotheses. This can be done, but not very quickly. We can then extract the sequents from these proofs and add them to the list of axiom sequents we can use while attempting to find a proof. In this rather expensive fashion we can avoid the problem of non-termination.

The axioms for non-freeness are very similar in that axioms 3.1), 3.2) and axioms 3.4) to 3.7) cause no problems and can be automated easily. The problem here is axiom 3.3) which does not obey the sub-expression condition. Thus we have a problem in instantiating the meta-variables A and C. Given the above solution to the problem for the equivalence axioms we can simply try all possible instantiations for which we know the equivalence sub-goals will succeed. This again could be a very expensive process. We must also be wary of non-termination if we simply keep attempting the same goal: this could be solved by keeping track of the goals already tried in order to avoid such a non-terminating cycle.

A final problem occurs with safe substitution: pushing a substitution down an expression is exceedingly tedious and of little interest to the user. If all the needed non-freeness conditions are present in the hypotheses then the application of the substitution will succeed with no problems.

The following is a simple presentation of a method by which a substitution might be applied to an expression. It results in a proof of an equivalence between the

application of the substitution to the original expression and the final expression obtained by applying the substitution.

The process depends on the form of the expression which selects the appropriate safe substitution axiom to be used. Suppose we are starting with $S\ A$ where S is the substitution and A the original expression. If axiom 5.1) is applicable we can then use it to prove:

$$S\ A == A$$

If axiom 5.2) applies the we can use it to get a proof of the following:

$$S\ A == T$$

Where T is the appropriate term from the substitution S. If one of axioms 5.4) to 5.7) apply then we use it. As an example we will assume A takes the form $L + R$ and use 5.4). We then get a proof of the following:

$$S\ A == S\ L + S\ R$$

We now recurse on $S\ L$ and $S\ R$ and will get proofs of:

$$S\ L == L' \quad \text{and} \quad S\ R == R'$$

We can then use axiom 2.4) to prove:

$$S\ L + S\ R == L' + R'$$

And finally using axiom 2.3) we can prove:

$$S\ A == L' + R'$$

If none of these axioms apply then if A is the application of a substitution then we can apply this process first and rewrite A to A' and then use axiom 2.3) to get the expression $S\ A'$ as a goal. If A is not an application of a substitution we can use axiom 2.1) to get a proof of $S\ A == S\ A$.

There are two problems with this simple scheme. The first is that it is not obvious how to decide if axiom 5.1) is in fact applicable. The second is that we have made no use of axiom 5.3) and this axiom will be needed. The problem is again the fact that this axiom does not satisfy the sub-expression condition and problems arise when trying to instantiate the meta-variables in the antecedents.

4.4 Conclusion

I believe there are two major issues involved in implementing logics:

- The clarity of expression of the target logic in the implementation.
- The security of the implementation and the ease with which one can convince oneself that one cannot prove untruths.

The first section, §4.1, demonstrated the clarity of expression possible in a functional language like Miranda. This section presumes we start with a specification of a

logic and wish to implement it in a way that allows us to convince others of its correctness. The strong type system and the abstract and algebraic types provide a powerful mechanism for implementing logics in a secure manner while maintaining the readability of the implementation. This approach not only reduces the extent of the code whose correctness must be argued but also makes this argument much easier.

In the second section, §4.2, we investigated the approach used by the Edinburgh Logic Framework. The theoretical foundations of this approach are, of course, far deeper than the implementation approach of §4.1. However, while its abstract nature does allow one to encompass more than one logic it representation can be hard to relate to the standard presentation. One is also left with the problem of tuning parsers and pretty printers for the LF for each logic represented. In the first approach one can tailor parsing and unparsing functions to suit the logic in question. For these reasons I prefer the approach of §4.1 to that used in the LF.

The logic presented by Paul Gardiner is still under development and all the following remarks must be seen in this light.

The 'B' system attempts to be a logic-independent system without a secure theoretical framework. I believe that the LF approach provides a far better basis for such a system.

The approach described in §4.3 has one major advantage: it gives a formalism for describing inference rules as sequents which naturally leads to an implementation of inference rules as data structures. This allows inference rules to be used as easily in the process of backward proof as forward proof. This is a distinct advantage over systems such as LCF where the set of inference rules is closed because they are implemented as functions, and where a completely different set of functions, called tactics, must be used during backwards proof.

The logic also isolates certain important issues that arise when attempting to implement a logic-independent system. The major issue is that of the representation of meta-variables and object variables and associated with this the problem of non-freeness side conditions. The logic gives a simple way of describing a usefully large range of inference rules.

The disadvantages seem to arise from the fact that any implementation would have to hide all the proof steps at the meta-level as only the proof steps at the object, entailment, level are of interest. It may be that this hiding is in fact not possible, as in the generation of extra antecedents from the derived rewriting rule.

Chapter 5

MOOSE[1]

5.1 Introduction

The debugging of imperative languages, and some "functional" languages, has long been aided by the availability of interactive debugging facilities. These may vary from simply tracing the execution of the program to halting the execution to examine various interesting values and even changing these values before continuing the execution. The essential facility provided is to allow the examination of the values bound to identifiers at various stages of the execution.

For most imperative languages this is a reasonable approach as the concept of

[1] "Miranda" Operationally Oriented Symbolic Evaluator.
Miranda is a Trademark of Research Software Ltd.
In this chapter we shall use a language resembling Miranda as our example language, and will refer to it as "Miranda". However, this chapter should not be taken as defining in any way the language Miranda, for such a definition one should see the documents from Research Software Ltd.

a "stage of the execution" is fairly clear, or can be made so through an abstract machine model. In general this is a achieved by a pairing of the current values assigned to variables in the state with the current position in the source code. This information may however be quite difficult to interpret. Firstly the entire state is open to modification and thus may need examination and secondly the language may allow unstructured jumps or even coroutining which obscure the relationship to the source code.

For applicative languages the situation initially seems simpler as we need not concern ourselves with examination of the state but merely with the values of the arguments of functions as they are applied and the only control structure is function application. This will be true for strict functional languages such as, if we ignore its assignment statement, ML [GMW79]. But for lazy functional languages such as Miranda the situation is much more complex.

The initial problem is that of displaying incompletely evaluated expressions when they occur as the arguments of a function application. A more serious problem arises from the various implementation techniques employed to ensure maximum sharing of computation. These involve compiling the source into some form of graph, usually of combinator applications, and it is this graph which is then executed. At any stage in the execution all we have is the graph which may be far removed from the initial source program. A description of the complex transformations needed to construct

such a graph is provided by Simon Peyton Jones [Jon87].

Richard Kieburtz, in his proposal for an interactive debugger for ML [Kie85], claims that an operational view of the evaluation of an applicative expression is not useful. Instead he proposes a system for testing functions on various "selected values" which are obtained using the exception mechanism of ML. We do not feel that this provides enough insight into the possible cause of a run-time error and, further, that an operational presentation could be useful.

In their debugging tool Toyn and Runciman [TR85] provide the user with a snapshot of the execution at the point of failure. They however use the underlying evaluation mechanism of the existing implementation annotated with references to the source program. This does allow them to provide a snapshot that relates at least in part to the source program. However, the form of the snapshots provided is somewhat cryptic and does not seem to provide much information about the context in which the error occurred. A more serious problem arises with their snapshot tool for lazy languages. There is a certain class of definitions, which they refer to as *pathological cases*, for which their debugger will attempt a non-terminating computation. This is because they must partially reduce the combinator graph in order to collect the annotations to provide the appropriate reference to the source program. A similar problem would arise with the Kieburtz approach if applied to lazy ML as the user must provide display functions for the selected values and these will be forced to

evaluate their arguments in order to display them.

Both these problems are due to the fact that these debuggers take the implementation as given, only modifying it slightly, and thus have no ability to deal with partially evaluated expressions. We believe that we can present the lazy evaluation strategy of "Miranda" in an operational fashion in terms of the source script[2] and then provide a more useful trace of the computation in terms of this operational model. We believe that an implementation based on the source program, though it will lose some efficiency, must provide a better description of the error than one based on combinator graphs. We further believe that a, user specified, trace provides the context lacking in a snapshot or an examination of selected values.

The major problem with this approach, is that the implementation used may fail to correctly model the details of the standard implementation. This is particularly true for a language like Miranda where the details of the lazy evaluation of expressions are expressed by subtle interactions of the combinators [Tur79]. There are indeed some subtle details of the sharing of computation as described here for "Miranda" that differ from the evaluation strategy of Miranda. We feel that these differences, explained in §5.6, are sufficiently minor as to not invalidate this approach.

Our basic idea is that when a computation fails its is within some particular function definition and that what we really need is a trace of the details of the computation

[2]It is this close relationship to the source script which explains our description of this approach as a symbolic evaluator.

as driven by selected "suspect" functions. A function drives the computation in two ways, firstly by forcing evaluation of its arguments through pattern matching and secondly by the application of functions in its defining right hand side expression. We believe that tracing both these forms of evaluation whilst hiding the internal details of the application of non-suspect functions may provide a useful insight into the cause of errors.

The basis of the evaluation algorithm for an applicative language is the process of applying a function to some arguments. This involves simply substituting the values of these arguments for the free occurrences of the functions formal parameter variables in its defining right hand side expression. The fact that the functions use lazy pattern matching to select between their different clauses and that sharing of computation occurs adds some details that will be explained in §5.4 and §5.6.

5.2 Expressions

There are only four basic sorts of expressions that will be encountered when evaluating an expression.

- A delayed application of some expression to some sequence of expressions. If there are sufficient arguments then we will be able to evaluate the application, otherwise no further evaluation will be possible. All applications are initially delayed and are only evaluated when necessary.

- An expression representing some functional object, that is something that can be applied to some arguments, there are four such expressions.

  1) A variable which is the name of some function defined either in the standard library or in the script.

  2) One of the built-in prefix or infix operator symbols. The infix notation for some of the built-in operators is regarded as merely a syntactic convenience expressing the operator's application to its two operands.

  3) A constructor defined by an algebraic type definition in the script or the standard library.

  4) A partial application of a function, built-in operator or constructor.

- The structure that results from the application of some constructor to some sequence of arguments, we shall call this a construct. Except when an expression evaluates to a function application with insufficient arguments it will always evaluate to some construct. Given the following delayed application of a constructor of arity n to n arguments:

  $$Con\ x_1 \cdots x_n$$

  we will write the construct that results from the application of the constructor, assuming it is non-strict, as:

  $$\mathbf{Con}\ x_1 \cdots x_n$$

Unless the strictness notation is employed all constructors defined in the script will be non-strict and hence the resulting constructs may have partially evaluated sub-expressions.

These constructors may be nullary and we can then regard them as representing constants. We can regard the built-in types **char** and **num** as being hidden algebraic types whose nullary constructors are referred to by a particular syntactic shorthand. Thus each character and number can be thought of as representing some nullary constructor of the appropriate type, and will therefore be regarded as a construct.

There are also non-nullary constructors represented by the list and tuple notations. We could regard the list notation:

$$[x_1, \ldots, x_n]$$

as representing the delayed nested application of n infix CONS constructors, where we use : for CONS and [] for the empty list.

$$x_1 : \cdots : x_n : []$$

This expression would then require evaluation of each infix application of CONS before the required list construct can be obtained. However, as the infix CONS constructor is non-strict in both its arguments we choose instead to regard this

as representing the construct that would result from evaluating such a nested application. Similarly we could regard the tuple notation:

$$(x_1, \ldots, x_n)$$

as representing the delayed application of a tuple forming constructor, Tuple say, of the correct type:

$$\text{Tuple } x_1 \cdots x_n$$

We also choose to regard this as the construct that would result from evaluating such an application, as tupling is also non-strict in all its arguments.

As a final point, strings are simply regarded as lists of characters.

- A variable representing some expression. These variables are bound to expressions either by pattern and function definitions or by appearing in the formal parameter patterns of a function definition where the pattern matching invoked by function application binds them to some expression.

5.3 Environments

As we wish to provide a useful tool for understanding the run-time behaviour of "Miranda" we choose to describe the evaluation in terms of expression rewriting

described at the level of "Miranda" itself. We can thus relate any particular stage of the evaluation to some part of the script.

The basic evaluation strategy is to repeatedly evaluate the applications present in the initial expression, this may involve invoking some of the functions defined in the script. The variables in the initial expression are provided with values by an environment defined by the script. The application of a function will rewrite to the right hand side expression of the function definition. Such an expression may contain identifiers whose values will be provided by the bindings defined for the formal parameter pattern variables, by pattern matching, and by any sub-definitions in a "where" clause.

We will actually maintain a list of environments each of which will correspond to a particular variable binding expression in the script. Each environment will have three components:

1. A unique environment identifier by which the appropriate variables in the expression can be associated with the environment.

2. A list of variable bindings, each variable binding will associate an expression with a variable. We shall write such bindings as:

   $$var \mapsto exp$$

   It will, in fact, be necessary to allow more than one expression to be bound to

a variable at once as will be explained in §5.5.2. In this case we will represent a multiple-valued binding as:

$$var \mapsto exp_1, \ldots, exp_n$$

This form of binding will be treated as the simple binding,

$$var \mapsto exp_1$$

everywhere except §5.5.2.

3. A list of definitions. There are two types of definition in a script; conformal, or pattern, definitions and function definitions. Both forms of definition will, as needed, be used to produce variable bindings.

Thus the initial environment will have an environment identifier, *id0* say, an initially empty list of variable bindings and a definitions list containing all the top level definitions in the script. As the evaluation proceeds these definitions will be used to provide variable bindings as required and these will be added to the bindings list for this initial environment.

All the variables in the current expression being evaluated will have some environment identifier associated with them. For instance, all the variables in the initial, user provided, expression will have *id0* associated with them.

The above form of environment might be implemented by the following algebraic type definition:

```
environment ::= Environment identifier
                              [(variable,[expression])]
                              [definition]
```

5.4 Rewriting

We regard expression evaluation as a process of rewriting. This rewriting is done in an environment as described in §5.3. As we intend to produce a tool that enables novice users of "Miranda" to debug their programs we require that the description of the evaluation algorithm serve as a tutorial for these novice users. Thus rather than describing the process of selecting the next redex and then its reduction we choose to use a simple recursive description. This presentation should allow the user to track the evaluation of an expression far more easily. This is because the "To evaluate X we must first evaluate Y and Z" style of description is closer to the user's model of the evaluation. The basic recursive rewrite function is $\mathcal{R}$ and we have two subsidiary iterative rewrite functions. The function $\mathcal{RC}$ is used in certain cases where we know, due to the type correctness of the script, that the expression must rewrite to a construct. It uses $\mathcal{R}$ to rewrite an expression until a construct is obtained. The complete evaluation of an expression involves applying $\mathcal{R}$ repeatedly, and recursively, as long as it is possible to do so. This is the definition of the function $\mathcal{R}^*$. Each of these functions takes an expression and the current environment list and returns the rewritten expression and a new, updated, environment list. It is necessary to

return an updated environment list as we may rewrite to an expression containing variables for which we previously had no environment. This occurs when we rewrite a function application to the appropriate right hand side expression. There are also two subsidiary functions, **bindings** and **conformality**, which are used when dealing with pattern and function definitions.

If an expression exp is rewritten to exp' in environment ρ returning the new environment list ρ' we will write this as:

$$exp' \diamond \rho' \longleftarrow \mathcal{R}[\![exp]\!]\rho$$

We will often refer to a rewrite as resulting in an expression or construct without mentioning the environment list that also results when we are not particularly concerned with the details of the new environment list.

5.4.1 $\mathcal{R}$

This is the basic recursive rewrite, we will describe it by cases. We will not present all the cases of the algorithm but a selection that displays the variety of recursive evaluations that are required.

1) $\mathcal{R}[\![x\ op\ y]\!]\rho$

   The infix application of a built-in operator. The expressions x and y may be further evaluated depending on the particular operator. As we can regard each operator as belonging to a group of operators whose rewrites are similar we only

give the details for one example operator from each such group. Only those operators which examine the detailed structure of constructs involve complicated rewrites. These are the equalities and the three list operators. The prefix operators $\sim$ and $-$ can be included in the groups for logical and arithmetic operators respectively. We must include a final case for the prefix operator # which, being a list operator, is more complex.

(a) $x : y$

This is *not* a construct but the application of the infix CONS operator : to the arguments x and y. We need not evaluate either x or y and can rewrite this to the following list construct:

$$x \ : \ y \diamond \rho$$

The infix function composition operator is rewritten similarly.

(b) $x + y$

This is the application of the infix sum operator. We must evaluate both x and y to constructs before we can apply the + operator. Therefore we get:

$$x' \diamond \rho' \longleftarrow \mathcal{RC}[\![x]\!]\rho$$
$$y' \diamond \rho'' \longleftarrow \mathcal{RC}[\![y]\!]\rho'$$

We can now calculate the sum of x' and y' as being constructs they must be numbers. If we call this sum n we have rewritten to:

$$n \diamond \rho''$$

The other infix arithmetic operators are rewritten similarly.

(c) $x \;\&\; y$

This is an application of the infix logical conjunction operator. This operator only evaluates its arguments x and y as required. We first evaluate the expression x to a construct:

$$x' \diamond \rho' \longleftarrow \mathcal{RC}[\![x]\!]\rho$$

If we have $x' \equiv \mathbf{False}$ then there is no point evaluating y and we can rewrite to:

$$\mathbf{False} \diamond \rho'$$

Otherwise we must rewrite y to a construct and the result we want will thus be the result of the following rewrite:

$$\mathcal{RC}[\![y]\!]\rho'$$

The infix logical disjunction is rewritten similarly.

(d) $x = y$

This is the application of the infix equality operator. We expect the arguments to be constructs, if they are not then this error terminates the

execution. We must first evaluate both arguments to get constructs before we can do the comparison. Therefore we get:

$$x' \diamond \rho' \longleftarrow \mathcal{RC}[\![x]\!]\rho$$
$$y' \diamond \rho'' \longleftarrow \mathcal{RC}[\![y]\!]\rho'$$

Assuming we have $x' \equiv \mathbf{Con}\ a_1 \cdots a_n$ and $y' \equiv \mathbf{Con'}\ b_1 \cdots b_m$ then we can compare the two constructors.

If we have $\mathbf{Con} \equiv \mathbf{Con'}$ then we must evaluate the following equalities, in sequence, to check that the sub-expressions are equal.

$$\begin{aligned} t_1 \diamond \rho_1 &\longleftarrow \mathcal{R}[\![a_1 = b_1]\!]\rho'' \\ &\vdots \\ t_n \diamond \rho_n &\longleftarrow \mathcal{R}[\![a_n = b_m]\!]\rho_{n-1} \end{aligned}$$

If for some t_i we have $t_i \equiv \mathbf{False}$ then we get:

$$\mathbf{False} \diamond \rho_i$$

Otherwise we get:

$$\mathbf{True} \diamond \rho_n$$

If we don't have $\mathbf{Con} \equiv \mathbf{Con'}$ then we get:

$$\mathbf{False} \diamond \rho''$$

Given the implied ordering on constructors, produced from the algebraic type specification, and using lexicographic ordering of the sub-expressions the inequalities can be rewritten in a similar fashion.

(e) $x!y$

This is the application of the infix list subscripting operator. We must first evaluate the subscript:

$$y' \diamond \rho' \longleftarrow \mathcal{RC}[\![y]\!]\rho$$

We must also evaluate the list to get a construct:

$$x' \diamond \rho'' \longleftarrow \mathcal{RC}[\![x]\!]\rho'$$

If we have $x' \equiv [\,]$ then this is an error and the execution terminates, otherwise we must have $x' \equiv a : b$. If we have $y' \equiv 0$ then we get:

$$a \diamond \rho''$$

Otherwise we must evaluate more of the list, if we let y'' be the number obtained by decrementing y' then we want the result of the following rewrite:

$$\mathcal{R}[\![b!y'']\!]\rho''$$

(f) $x +\!+ y$

This is the application of the infix list concatenation operator. We rewrite the first argument to get a construct:

$$x' \diamond \rho' \longleftarrow \mathcal{RC}[\![x]\!]\rho$$

If we have $x' \equiv [\,]$ then we get:

$$y \diamond \rho'$$

Otherwise we must have $x' \equiv a : b$ and we get:

$$a : (b +\!+ y) \diamond \rho'$$

(g) $x --y$

This is the application of the infix list difference operator[3]. We must evaluate the first argument to a construct:

$$x' \diamond \rho' \longleftarrow \mathcal{RC}[\![x]\!]\rho$$

If we have $x' \equiv [\,]$ then we get:

$$[\,] \diamond \rho'$$

Otherwise we must have $x' \equiv a : b$ and we must check whether or not a is a member of y. To do this we assume the existence of a list membership operator:

$$t \diamond \rho'' \longleftarrow \mathcal{RC}[\![a \in y]\!]\rho'$$

If we have $t \equiv$ **True** then we get the result of the following rewrite:

$$R[\![b --(y --[a])]\!]\rho''$$

[3]The evaluation described here is *not* the one used in Miranda.

Otherwise we get:

$$a : (b --(y --[a])) \diamond \rho''$$

(h) $\#x$

We must first evaluate the expression x to get a construct:

$$x' \diamond \rho' \longleftarrow \mathcal{RC}[\![x]\!]\rho$$

If we have $x' \equiv [\,]$ then we get:

$$0 \diamond \rho'$$

Otherwise we must have $x' \equiv a : b$ and we keep rewriting b until we get a completed list construct. Suppose this repeated rewriting of the tail of the list results in the following construct:

$$a_1 : \cdots : a_n : [\,] \diamond \rho''$$

we have therefore rewritten to:

$$n \diamond \rho''$$

2. $\mathcal{R}[\![\ [x, y\ ..\ z]\ ]\!]\rho$

This is a representation of a numeric list generating function. We must first evaluate x and y in order to calculate the difference between elements of the list:

$$x' \diamond \rho' \longleftarrow \mathcal{RC}[\![x]\!]\rho$$

$$y' \diamond \rho'' \longleftarrow \mathcal{RC}[\![y]\!]\rho'$$

We can now calculate the difference, d say, and its sign tells us in which sense to test against the bound z. We must first evaluate z:

$$z' \diamond \rho''' \longleftarrow \mathcal{RC}[\![z]\!]\rho''$$

If d is negative we test if x' is less than z' otherwise we test if x' is greater than z'. If the test fails then we rewrite to:

$$[\,] \diamond \rho'''$$

Otherwise we rewrite to:

$$x'\colon [x'', y'' \,..\, z'] \diamond \rho'''$$

where x'' is the sum of x' and d and y'' is the sum of y' and d.

3. $\mathcal{R}[\![var]\!]\rho$

This variable, *var*, will have an environment identifier associated with it, *id* say, and we can use this to select the appropriate environment from the environments list ρ. If *var* is defined by a function definition then we cannot rewrite this expression. If it is defined by a pattern definition there are two possibilities.

- If the variable *var* currently has no binding in its environment we can use its pattern definition to provide one.

  If the definition for *var* is:

  $$pat = exp$$

  then we first associate the environment identifier *id* with all the free variables in *exp* that have no associated environment identifier. These will be those identifiers bound in the definition by appearing in the pattern. This produces a new version of *exp* which we shall call *exp'*.

  We next use the function **bindings** to create a list of variable bindings, one for each variable occurring in the pattern.

  $$new\text{-}bindings \leftarrow \textbf{bindings}(pat, exp')$$

  We then extend the environment *id* by adding this bindings list to the already existing bindings list, this gives us a new list of environments ρ'.

  We must next check that the expression *exp'* does indeed evaluate to a structure whose shape corresponds to that of the pattern *pat.* This is called a conformality check, it may involve evaluating the expression *exp'* and so may alter the environment.

  $$\rho'' \leftarrow \textbf{conformality}(pat, exp', \rho')$$

This conformality check may fail due to a conformality error and if this happens then the evaluation terminates with this error. If it succeeds we will have produced a variable binding for *var* in environment *id*. If this binding is:

$$var \mapsto x$$

Then we have rewritten to:

$$x \diamond \rho''$$

- If we already have the variable binding:

$$var \mapsto y$$

then we can rewrite to:

$$y \diamond \rho$$

4. $\mathcal{R}[\![\text{"abc"}]\!]\rho \;\; \mathcal{R}[\![\text{'c'}]\!]\rho \;\; \mathcal{R}[\![123]\!]\rho$

These are all completely evaluated constructs and cannot be further evaluated.

5. $\mathcal{R}[\![x\ x_1 \cdots x_n]\!]\rho$

We first evaluate the expression x as much as possible:

$$x' \diamond \rho' \longleftarrow \mathcal{R}^*[\![x]\!]\rho$$

The type correctness of the script means that x cannot be a construct. Thus by examining the definition of $\mathcal{R}^*$ it can be seen that x' must be either a function

name or a partial application with insufficient arguments. Thus in general we will have a left nested application such as:

$$(\cdots((f\ x_1^1 \cdots x_{n_1}^1)\ x_1^2 \cdots x_{n_2}^2)\cdots)\ x_1^m \cdots x_{n_m}^m$$

Where f is either a function name, constructor or built-in operator. This left nesting is irrelevant to the essential evaluation of the expression so we can regard this more simply as an expression of the form:

$$f\ y_1 \cdots y_k$$

where y_1 is x_1^1 and y_k is $x_{n_m}^m$. There are thus three cases to consider.

(a) f is a constructor, *Con* say. If there are enough y_i to provide all the fields of the constructor then, due to the type correctness of the script, all of them must be required and we rewrite to the following construct:

$$\mathbf{Con}\ y_1' \cdots y_k' \diamond \rho''$$

Where y_i' is simply y_i unless field i of the constructor *Con* is strict. If field i is strict then y_i' is the result of completely evaluating the expression y_i. If none of the y_i needs further evaluation then ρ'' is simply ρ', otherwise it is the modified environment produced by these further rewrites.

If there are not enough arguments then we simply get:

$$\mathrm{Con}\ y_1 \cdots y_k \diamond \rho'$$

(b) f is a function name. If the first clause of the definition of f is:

$$f\ pat_1 \cdots pat_j = rhs$$

Then we will only require j of the k available arguments. We must pattern match the y_i against the pat_i to check that this clause of the function definition is applicable. We first create a variable bindings list:

$$new\text{-}bindings \leftarrow \mathbf{bindings}(pat_1, y_1) \oplus \cdots \oplus \mathbf{bindings}(pat_j, y_j)$$

We can now create a new environment for the formal parameter variables. It will have a new environment identifier, *new-id* say, its variable bindings list will be *new-bindings* and its definitions list will be empty. We add this new environment to the environment list ρ to get the new environment list ρ'.

We can now perform a conformality check of the y_i against the pat_i.

$$\rho'' \leftarrow \mathbf{conformality}(pat_1, y_1, \ldots, pat_j, y_j, \rho')$$

We can regard this conformality checking of multiple pattern/expression pairs as a conformality check of the tuple of patterns against the tuple of expressions:

$$\mathbf{conformality}((pat_1, \ldots, pat_j), (y_1, \ldots, y_j), \rho')$$

This works because of the lazy product matching employed on tuples, see §5.5. These conformality checks may fail in which case we cannot use this

clause of the function definition and try again with the next clause. If this is the last clause of the function definition then this is an error and the evaluation terminates.

If the conformality check succeeds then we next make a new version of *rhs* by associating the environment identifier of f with all the free variables in *rhs* that have no associated environment identifier. Those variables occurring in the formal parameter patterns are regarded as bound whilst doing this.

We next associate *new-id* with all the occurrences in *rhs* of the variables bound by the formal parameter patterns. We can call this new version of the right hand side of this clause of the function definition rhs'.

This new right hand side will need evaluation if it has sub-definitions, for which we must add an environment to the environments list, or guards that must be evaluated to select the appropriate right hand side expression. If the right hand side is simply an expression this extra evaluation need not be done and we get:

$$rhs'\ y_{j+1} \cdots y_k \diamond \rho''$$

Otherwise we want the result of the following rewrite:

$$rhs''\ \diamond \rho''' \longleftarrow \mathcal{R}[\![rhs']\!]\rho''$$

and the entire application is rewritten to:

$$rhs''\ y_{j+1} \cdots y_k \diamond \rho'''$$

If there are not enough arguments then we simply get:

$$f\ y_1 \cdots y_k$$

(c) The built-in operators are dealt with in the same way as function definitions.

6. $\mathcal{R}[\![expl \text{ where } md]\!]\rho$

Here we make a new environment from the sub-definitions. It has a new environment identifier, *new-id* say, its variable bindings list is empty and its definitions list is the sub-definitions *md*. We add this new environment to ρ to get a new environments list ρ'. We can now rewrite the guarded expression list in this new environment:

$$\mathcal{R}[\![expl]\!]\rho'$$

7. A sequence of m guarded right hand side expressions:

$$\begin{array}{lcl} exp_1 & , & grd_1 \\ \vdots & \vdots & \vdots \\ exp_m & , & grd_m \end{array}$$

We evaluate the first guard:

$$grd'_1 \diamond \rho'_1 \longleftarrow \mathcal{RC}[\![grd_1]\!]\rho$$

If we have $grd'_1 \equiv$ **True** then we rewrite to the guarded expression:

$$exp_1 \diamond \rho'_1$$

If instead we have $grd'_1 \equiv$ **False** then we go on to evaluate the second guard in environment ρ'_1. We continue until we find a guard that evaluates to **True**, if none of them does then this is an error and the evaluation terminates.

5.4.2 $\mathcal{RC}$

This is a multiple step rewrite, it rewrites an expression repeatedly until a construct is obtained. As this function is only applied where we know, due to the type correctness of the script, that the result must be a construct it can safely repeat the basic recursive rewrite until it gets a construct. Notice that the resulting construct may have incompletely evaluated sub-expressions.

$$\mathcal{RC}[\![exp]\!]\rho$$

We rewrite the expression once:

$$exp' \diamond \rho' \longleftarrow \mathcal{R}[\![exp]\!]\rho$$

If we have $exp' \equiv \textbf{Con}\ x_1 \cdots x_n$ for some constructor *Con* then we have succeeded in rewriting to a construct and have rewritten to:

$$\textbf{Con}\ x_1 \cdots x_n \diamond \rho'$$

Otherwise we must continue with the evaluation of exp' which should result in a construct:

$$\mathcal{RC}[\![exp']\!]\rho'$$

5.4.3 $\mathcal{R}^*$

This is a multiple step rewrite, it rewrites an expression completely if possible.

$$\mathcal{R}^*[\![exp]\!]\rho$$

We first attempt to rewrite the expression once:

$$exp' \diamond \rho' \longleftarrow \mathcal{R}[\![exp]\!]\rho$$

If we failed to rewrite the expression at all then we get:

$$exp \diamond \rho$$

Otherwise we must look at the structure of exp'. If exp' is not a construct then we must continue rewriting, we want the result of the following rewrite:

$$\mathcal{R}^*[\![exp']\!]\rho'$$

Otherwise we have $exp' \equiv \mathbf{Con}\ x_1 \cdots x_n$ and we must completely evaluate the sub-expressions:

$$\begin{array}{lcl} x'_1 \diamond \rho_1 & \longleftarrow & \mathcal{R}^*[\![x_1]\!]\rho' \\ & \vdots & \\ x'_n \diamond \rho_n & \longleftarrow & \mathcal{R}^*[\![x_n]\!]\rho_n \end{array}$$

We then finally get:

$$\mathbf{Con}\ x'_1 \cdots x'_n \diamond \rho_n$$

5.5 Pattern Matching

"Miranda" performs pattern matching on two occasions. The first is when providing variable bindings from a pattern definition and the second is when selecting the appropriate clause of a multiple clause function definition. "Miranda" implements a form of lazy pattern matching [Jon87, pages 51–81]. Lazy pattern matching essentially involves noting that expressions of a product type can only evaluate to constructs formed by that type's single constructor. It is therefore not necessary to evaluate such an expression to ensure it is of the correct shape as, due to the type correctness of the script, it must be. This is because product types, with non nullary constructors, correspond to the direct product domain construction. Patterns formed only of product type constructors and unrepeated variables cannot fail to be matched by an expression of the correct type and are thus called irrefutable patterns. Product types

whose single constructor is nullary are not regarded as irrefutable patterns as we wish to distinguish these constant values from the bottom element of their type.

Pattern matching proceeds in two stages; in the first bindings are provided for the variables in the pattern and in the second the shape of the expression is compared against the pattern.

5.5.1 Variable Bindings

We now describe how we construct a variable bindings list as part of the pattern matching process. We are given a pattern, which may contain variables, and an expression.

bindings(*pat*, *exp*)

We bind each variable to the, delayed, application of a sequence of selector functions to the expression. When these selector functions are applied they will extract the desired sub-expression from the expression, evaluating the expression just sufficiently to do so. These selectors will only be applied if the value of the variable is required, if it never is required then the selectors will not be applied and the expression will not suffer unnecessary evaluation.

We now describe how these variable bindings are constructed. We have three cases to consider depending on the form of the pattern.

1. If the pattern is a variable, *var* say, then we form the singleton variable bindings list:

   $$[var \mapsto exp]$$

   Variables are a special case of irrefutable patterns.

2. If the pattern is an irrefutable pattern then it must be a construct of the form:

   $$(pat_1, \ldots, pat_k)$$

   where the pat_i are irrefutable patterns with no variables in common. We can simply calculate the variable bindings list for each of the pat_i and then concatenate these lists as there cannot be multiple bindings for any variable.

   $$\textbf{bindings}(pat_1, \text{selector}_1^k\ exp) +\!\!+ \cdots +\!\!+ \textbf{bindings}(pat_k, \text{selector}_k^k\ exp)$$

   The selector functions are defined such that selector_i^j extracts the i^{th} field from a j-field product type constructor. These selector functions will not need to check the shape of their argument when they are applied as the type correctness of the script ensures that it must evaluate to something of the correct shape.

3. If the pattern is a refutable pattern then it can involve constructors other than product type constructors and may involve repeated variables.

   $$\textbf{Con}\ pat_1 \cdots pat_k$$

We again calculate the bindings lists for each of the pat_i but this time we must take into account the repeated variables. This is done by the special list concatenation operator $\oplus$ which constructs a multiple binding for each repeated variable.

$$\textbf{bindings}(pat_1, \text{selector-Con}_1\ exp) \oplus \cdots \oplus \textbf{bindings}(pat_k, \text{selector-Con}_k\ exp)$$

The selector functions are defined such that when selector-Con_i is applied it will test its argument to check it is a construct formed by the application of the constructor *Con*. If the argument is not of the correct shape this indicates that the pattern matching has failed. If it succeeds it will return sub-expression i of the construct.

As already mentioned the operator $\oplus$ takes care of repeated variables, if it is given two variable bindings lists that both contain bindings for a variable *var*:

$$var \mapsto exp_1, \ldots, exp_j$$

$$var \mapsto exp_k, \ldots, exp_n$$

then it coalesces these into a single binding for *var*:

$$var \mapsto exp_1, \ldots, exp_j, exp_k, \ldots exp_n$$

5.5.2 Conformality Checking

When we attempt a conformality check we may have to evaluate an expression to check that its shape matches a pattern. While performing a conformality check we keep track of the variables we have seen in the pattern, this is so that we can deal correctly with repeated variables in the pattern. We will require that all occurrences of a variable in a pattern be bound to the same expression. We are given the pattern, the expression and the environment list in which to evaluate the expression.

conformality(pat, exp, ρ)

There are only two cases to check:

1. If the pattern is a variable or irrefutable pattern then the expression must evaluate to a structure of the correct shape and we need not evaluate the expression. However we must check that the variable, or variables for a pattern, have not already been seen. If any variable in this pattern has been seen then it is a repeated variable and we check that the expressions associated with it at its multiple occurrences are equal. For such a repeated variable *var* we will find that it has a multiple binding:

$$var \mapsto exp_1, exp_2, \ldots, exp_n$$

The value exp_1 is that associated with all previous occurrences of *var* in the entire pattern being checked, exp_2 is the value associated with the, single, occurrence of *var* in *pat*. Thus we must test the following equality:

$$b \diamond \rho' \longleftarrow \mathcal{R}[\![exp_1 = exp_2]\!]\rho$$

If we have $b \equiv$ **False** then the conformality check has failed. Otherwise we have $b \equiv$ **True** and the conformality check has succeeded. We now update the environment ρ' by replacing the binding for *var* with the binding:

$$var \mapsto exp'_2, \ldots, exp_n$$

where exp'_2 is the evaluated form of exp_2 produced by the evaluation of the equality. We can then return this updated environment list. As can be seen this multiple binding is one shorter than the original, thus when we have checked the entire pattern all the variables will have a single binding.

2. If the pattern is refutable then it will take the form:

   Con $pat_1 \cdots pat_k$

   We must first evaluate the expression to get a construct:

   Con' $exp_1 \cdots exp_j \diamond \rho' \longleftarrow \mathcal{RC}[\![exp]\!]\rho$

If we don't have $\mathbf{Con} \equiv \mathbf{Con}'$ the conformality check has failed. Otherwise we must check the pat_i against the exp_i, if the constructors are the same we must have $j = k$:

$$\begin{array}{lcl} \rho'_1 & \leftarrow & \mathbf{conformality}(pat_1, exp_1, \rho') \\ \vdots & \vdots & \vdots \\ \rho'_k & \leftarrow & \mathbf{conformality}(pat_k, exp_k, \rho'_{k-1}) \end{array}$$

If all of these conformality checks succeed then we can return the environment ρ'_k.

5.6 Sharing Computation

The above description defines a lazy evaluation strategy in the simple sense that nothing is evaluated unless it is needed. However there are other ways of avoiding evaluation which derive from avoiding duplicated evaluations. A full discussion of the various ways in which computation can be shared in the implementation of functional languages is presented by Arvind, Kathail and Pingali [AKP85]. There are three basic cases where sharing is possible and we discuss these in order of increasing complexity.

1. One basic form of sharing is to avoid unnecessary copying of expressions. The simplest form of this sharing is that occasioned by multiple occurrences of a formal parameter variable in a right hand side expression. Consider the following

simple function definition:

$$f\ x = x * x$$

Now if we attempt to evaluate $f\ (2+3)$ we don't want this to result in the evaluation of $(2+3)*(2+3)$ when it is clear that $(2+3)$ need only be evaluated once. The basic idea is to make both occurrences of x in the expression $x * x$ refer to the *same* expression so that when the first x is evaluated the second x also enjoys the fruits of this evaluation.

The simplest way to do this would be to elaborate the form of expressions slightly by allowing variables to be annotated as "evaluated". Then any evaluation of a sub-expression once referenced by a variable would still effect the binding for that variable through the continuing presence of the annotated variable. This approach will work but has the disadvantage that long strings of annotated variables might arise and they will maintain unnecessary references to unneeded environments.

The neater solution is to use pointers, this will also be of use for the second and third forms of sharing. The use of pointers allows us to very simply make both occurrences of x refer to the same expression. This is done by binding x not to the expression $(2+3)$ but instead to a pointer to the expression $(2+3)$. Thus whenever a binding is added to the environment it will always bind the variable

to a pointer. Any evaluation of the pointed-to expression is then apparent to all things pointing to it. This use of a pointers also allows the bindings produced by the function **bindings** to all refer to the same argument expression rather than multiple copies.

2. The second way of introducing sharing is to note that, for any function definition with constant terms in its right hand side expression, it is not necessary to evaluate these expressions at each application, as their value will always be the same. Generalising this idea we arrive at the concept of the maximal free sub-expression [AKP85, page 25]. A free sub-expression of the right hand side expression of a function definition is one containing no free occurrences of the formal parameter pattern variables. A maximal free sub-expression is one that is not a sub-expression of any other free sub-expression. These maximal free sub-expressions will always evaluate to the same value regardless of the value of the arguments for any particular application of the function. They therefore need only be evaluated once; we can again use pointers to achieve this effect.

   Suppose we have the following function definition:

   $$f\ x = x * x + 2 * y$$

   where y is defined elsewhere. Instead of adding this definition to the definitions list of an environment in the environments list we add a modified version. This

modified version is obtained by replacing the sub-expression $2 * y$ with a pointer to this sub-expression. Thus when we copy the right hand side during applications of this function we will copy this pointer and no matter how many times the function is applied the sub-expression $2 * y$ will only be evaluated once.

Thus in general all the maximal free sub-expressions in the right hand side of a definition are replaced with pointers to those sub-expressions.

3. The third form of sharing arises from the fact that all functions are curried and that partial application is allowed. Suppose we have the following function definition:

$$f\ x\ y = x * x + y$$

Now consider the following definition:

$$z = g\ 2 + g\ 3 \text{ where } g = f\ 7$$

The right hand side expression of the definition of z will be evaluated in an environment containing the following binding:

$$g \mapsto f\ 7$$

Thus the right hand side expression will evaluate to the following expression:

$$\underline{(f\ 7)}2 + \underline{(f\ 7)}\ 3$$

Where both sub-expressions $(f\ 7)$ are in fact the same, pointed-to, expression. However, this sub-expression cannot be evaluated because it is a partially applied function needing one more argument. Thus the first operand of $+$ must be evaluated as a left nested application giving rise to:

$$7 * 7 + 2$$

And its second operand, having gained nothing from the evaluation of the first, gives rise to:

$$7 * 7 + 3$$

As can be seen, we are evaluating the sub-expression $7 * 7$ twice; we would like to avoid this.

The solution is to extend the concept of maximal free sub-expressions to cope with the partial application of functions. The first thing to note is that, when we evaluate the first operand of $+$ we will first attempt to evaluate the sub-expression $(f\ 7)$ before we evaluate the entire left nested application. This evaluation will not be possible due to the lack of a second argument. We can notice this and proceed to ensure that the double occurrence of this particular partial application of f will not result in the double evaluation of $7 * 7$. Given that we know the first argument for f then it is clear that any sub-expressions of the expression defining f involving only x and constants will evaluate to the

same value at all applications of this partial application, regardless of the value of the second argument. We therefore associate a *further* modified version of the definition of f with the partial application bound to g. This modified version of f has the sub-expression $x * x$ replaced by a pointer to this sub-expression. Now when the left nested applications in the operands of $+$ are evaluated it is this definition that is used and so the copying of the right hand sides of this definition will not produce two copies of the sub-expression $x * x$ but two pointers to this single sub-expression. Now the evaluation of the first operand will evaluate this pointed to sub-expression to the value 49 which will then be available for the evaluation of the second operand.

Thus we can, in general, notice a partial application and associate with it a further modified version of the function definition. This modified definition is obtained by replacing maximal free sub-expressions with pointers as in case 2) but we now regard those formal parameter variables associated with the available arguments in the partial application as constants when determining the maximal free sub-expressions.

This is where this implementation differs from that of Miranda. This is because in Miranda patterns are compiled into nested lambda abstractions with more opportunities for partial applications, and hence sharing, than we can achieve. In particular, if a function has a multiple clause definition then we cannot invoke

this procedure as we cannot tell which clause will be matched at any eventual application.

5.7 Implementation

The evaluation algorithm presented above was used as the basis for the implementation of an interactive symbolic evaluator for "Miranda" written in Miranda.

Given a source script we can evaluate expressions that involve definitions from that script and the standard library. The symbolic evaluator requires the source script to be type checked and thus requires that the `typecheck` program of §3.7 be run on the source script first. We are therefore reusing both the "Miranda" parser and type checker from this earlier piece of work. The interactive user interface to this program is again written using interactions as described in [Lon89]. We use a modified form of interactions which provides a form of exception handling. This is required to cope with errors during the evaluation of expressions or premature termination by the user. As with the type checker the user interface was not our main area of concern. Indeed, as will be explained, we spent even less effort on it in this case as it soon became clear that there are certain intractable problems in tracing lazy evaluations. The implementation of the evaluation algorithm itself also uses interactions to enable it to cope with the various exceptional conditions which may arise. A couple of cases which give a flavour of this implementation are given in

```
eval' (ST_Identifier "True") expr
      = push (new_structure (ST_construct "True") [])
                $then
        change_state set_rw

eval' (ST_expression "&") expr
      = push expr
                $then
        eval_const arg1
                $then
        test    (state_condition is_False)
                (popn 2 build_result)
                (pop (const identity)
                             $then
                 eval_const arg2
                             $then
                 popn 2 build_result
                )
                $then
        change_state set_rw
        where
        [arg1,arg2] = substructures_of expr
        build_result [expr,arg] = push arg
```

Figure 11: Evaluation

Figure 11. As can be seen they correspond fairly closely to the specification of the evaluation algorithm. Unfortunately the implementation of the tracing and breaking of applications means that its implementation is far from clear.

The evaluator takes the type checked version of a Miranda script and then allows the user to enter expressions to be evaluated in the context provided by that script. Thus the top-level user interface is very like that of Miranda itself. During the evaluation of the input the user is presented with a trace of the progress of the evaluation. This trace shows the evaluation of all the applications of operators and functions and their operands in the input expression. The evaluation resulting from the applications of the functions is only shown if that function is being traced, otherwise we merely see the result of the application. We may also choose to break on the application of certain functions, this allows us to examine the arguments to the function and also to change the functions being traced or that we wish to break on.

The basic problem we encountered is that of describing the current context in the evaluation and relating the current expression to that context. The description of the function `R` is misleading in that it seems to imply that the evaluation of an expression can easily be related to that of its parent expression. This is not true because most of the evaluation is driven by the pattern matching and these evaluations are hidden inside the definition of the function **`conformality`**. Thus the definition of a function which is returned from its application may be passed around through a great many

functions completely unevaluated. When some pattern matching eventually forces its evaluation it is almost impossible to relate it to the function definition or application that produced it. This also means that there is no obvious way of measuring the "depth" of the evaluation and thus no way to use indentation to present it in a more readable form. Even worse, because we don't substitute expressions for variables until we require their value we can build up complex unevaluated expressions with multiple occurrences of variables of the same name but from different contexts. As these contexts must include not only the function definition but its particular application there is no simple way to annotate the variables. Because the user interface we implemented is so crude it does not serve as a convincing demonstration of these points, being independently incomprehensible.

A further problem with this implementation is more mundane. Because we built the system on top of the type checker the internal representation of a source script is far larger than is needed for the symbolic evaluation. This combined with the large amount of structure copying implied by our model of environments means that only small, relatively uninteresting, scripts can be evaluated.

5.8 Conclusion

While an operational source level description of "Miranda" can be produced and used as the basis of an implementation in a fairly straightforward way the inherent

complexity of the evaluations makes it impossible to trace them. While it is true that little time was spent on the user interface and it might be argued that with more a effort a useable tool could be produced I do not think this is the case. I believe that the problems of describing the current state of the evaluation and of relating partially evaluated expressions back to the script are insurmountable.

This example does show that the interactions of [Lon89] can be used in any context where exceptional conditions must be handled, not simply in user interfaces.

Appendix A

"Miranda" Syntax

The following is the full source of the YACC script used to generate the "Miranda" parser of Chapter 3. The details of this procedure are given in [Lon87a].

```
/*    The syntax of Miranda as input to Yacc.

      This grammar is a restricted version of the full Miranda grammar.
      We don't allow user defined infix functions or constructors.
*/

/*    The lexical tokens
*/
```

```
/*    Operator precedence.
*/



/*    The start non-terminal.
*/



script
      : decl_s0
      ;

decl
      : tform TK_synonym TK_offside_begin type TK_offside_end
      | tform TK_comprises TK_offside_begin constructs TK_offside_end
      /*    We will actually require that all the laws associated with an
            algebraic type follow immediately after the declaration of
            the type.
      */
      | law
      | TK_abstype tform_list TK_with TK_offside_begin signature TK_offside_en
      | def
      | spec
      | libdir
      ;

def
      : lhs TK_equal rhs
      ;

/*    We require the lhs for a law to be a pattern.
*/
law
      : lhs TK_law lawrhs
```

```
        ;

signature
        : spec spec_s0
        ;

/*      In the first of these two we require the left hand side to be a list
        of identifiers.
*/
spec
        : tform_list TK_of_type TK_offside_begin type TK_offside_end
        | tform_list TK_of_type TK_offside_begin TK_type TK_offside_end
        ;

constructs
        : construct TK_alternative constructs
        | construct
        ;

construct
        : TK_Identifier field_s0
        ;

field
        : type1
        | type1 TK_exclamation
        ;

type
        : type0
        | type0 TK_right_arrow type
        ;

type0
        : TK_identifier type1_s0
        | type2
        ;

type1
        : TK_identifier
```

```
      | type2
      ;

type2
      : TK_type_variable
      | TK_left_bracket TK_right_bracket
      | TK_left_bracket type_list TK_right_bracket
      | TK_left_square_bracket TK_right_square_bracket
      | TK_left_square_bracket type_list TK_right_square_bracket
      ;

tform
      : TK_identifier type_variable_s0
      ;

lhs
      : lhs0 TK_cons lhs
      | lhs0 TK_plus TK_number
      | lhs0
      ;

/*    If the lhs is a sequence of lhs1 then we require all but the first
      to be patterns, if this is under a ":" or "+" then they
      must all be patterns.
      If the tails sequence of lhs1's is non-empty then the first must
      be either be an identifier or a constructor, or be a sequence of lhs1
      that satisfies this condition.
      If we have an identifier then this must be a function definition,
      otherwise it is a pattern definition.
*/
lhs0
      : lhs1 lhs1_s0
      ;

lhs1
      : TK_identifier
      | TK_Identifier
      | literal
      | TK_left_bracket TK_right_bracket
      | TK_left_bracket lhs_list TK_right_bracket
```

```
      | TK_left_square_bracket TK_right_square_bracket
      | TK_left_square_bracket lhs_list TK_right_square_bracket
      ;

rhs
      : TK_offside_begin exp TK_offside_end
      | TK_offside_begin exp TK_where def def_s0 TK_offside_end
      | TK_offside_begin exp TK_comma exp TK_offside_end
      | TK_offside_begin exp TK_comma TK_otherwise TK_offside_end
      | TK_offside_begin exp TK_comma exp TK_offside_end TK_equal rhs
      | TK_offside_begin exp TK_comma exp TK_where def def_s0 TK_offside_end
      | TK_offside_begin exp TK_comma TK_otherwise
        TK_where def def_s0 TK_offside_end
      ;

lawrhs
      : TK_offside_begin exp TK_offside_end
      | TK_offside_begin exp TK_where def def_s0 TK_offside_end
      | TK_offside_begin exp TK_comma exp TK_offside_end
      | TK_offside_begin exp TK_comma TK_otherwise TK_offside_end
      | TK_offside_begin exp TK_comma exp TK_offside_end TK_law lawrhs
      | TK_offside_begin exp TK_comma exp TK_where def def_s0 TK_offside_end
      | TK_offside_begin exp TK_comma TK_otherwise
        TK_where def def_s0 TK_offside_end
      ;

exp
      : exp0
      | operator
      ;

exp0
      : exp0 TK_cons exp0
      | exp0 TK_append exp0
      | exp0 TK_difference exp0
      | exp0 TK_or exp0
      | exp0 TK_and exp0
      | TK_not exp0 %prec PREFIX_not
      | exp0 TK_greater exp0
      | exp0 TK_greater_or_equal exp0
```

```
      | exp0 TK_equal exp0
      | exp0 TK_not_equal exp0
      | exp0 TK_less exp0
      | exp0 TK_less_or_equal exp0
      | exp0 TK_plus exp0
      | exp0 TK_minus exp0
      | TK_minus exp0 %prec PREFIX_minus
      | exp0 TK_multiply exp0
      | exp0 TK_divide exp0
      | exp0 TK_div exp0
      | exp0 TK_mod exp0
      | exp0 TK_power exp0
      | exp0 TK_compose exp0
      | TK_length exp0 %prec PREFIX_length
      | exp0 TK_exclamation exp0
      | application
      ;

application
      : simple simple_s0
      ;

simple
      : TK_identifier
      | TK_Identifier
      | literal
      | TK_show
      | TK_left_bracket TK_right_bracket
      | TK_left_bracket exp_list TK_right_bracket
      | TK_left_square_bracket TK_right_square_bracket
      | TK_left_square_bracket exp TK_right_square_bracket
      | TK_left_square_bracket exp TK_dot_dot exp_opt
        TK_right_square_bracket
      | TK_left_square_bracket exp TK_comma exp TK_dot_dot exp_opt
        TK_right_square_bracket
      | TK_left_square_bracket exp TK_comma exp TK_right_square_bracket
      | TK_left_square_bracket exp TK_comma exp TK_comma exp_list
        TK_right_square_bracket
      | TK_left_square_bracket exp TK_alternative qualifier_list
        TK_right_square_bracket
```

```
      | TK_left_square_bracket exp TK_diagonalise qualifier_list
        TK_right_square_bracket
      ;

qualifier
      : exp
      /*In these next three cases the expressions to the left
        of the arrow must in fact be a patterns.
      */
      | exp TK_comma exp_list TK_left_arrow exp
      | exp TK_left_arrow exp
      | exp TK_left_arrow exp TK_comma exp TK_dot_dot
      ;

literal
      : TK_number
      | TK_string
      | TK_character
      ;

operator
      : prefix
      | infix
      ;

infix
      : TK_cons
      | TK_append
      | TK_difference
      | TK_or
      | TK_and
      | TK_greater
      | TK_greater_or_equal
      | TK_equal
      | TK_not_equal
      | TK_less_or_equal
      | TK_less
      | TK_plus
      | TK_multiply
      | TK_divide
```

```
        | TK_div
        | TK_mod
        | TK_power
        | TK_compose
        | TK_exclamation
        ;

prefix
        : TK_minus
        | TK_not
        | TK_length
        ;

/*    Sequence (*), optional (?)  and list productions
*/

type1_s0
        : /* empty */
        | type1 type1_s0
        ;

decl_s0
        : /* empty */
        | decl decl_s0
        ;

tform_list
        : tform
        | tform TK_comma tform_list
        ;

spec_s0
        : /* empty */
        | spec spec_s0
        ;

type_list
        : type
        | type TK_comma type_list
        ;
```

```
lhs_list
      : lhs
      | lhs TK_comma lhs_list
      ;

lhs1_s0
      : /* empty */
      | lhs1 lhs1_s0
      ;

field_s0
      : /* empty */
      | field field_s0
      ;

type_variable_s0
      : /* empty */
      | TK_type_variable type_variable_s0
      ;

def_s0
      : /* empty */
      | def def_s0
      ;

simple_s0
      : /* empty */
      | simple simple_s0
      ;

exp_list
      : exp
      | exp TK_comma exp_list
      ;

exp_opt
      : /* empty */
      | exp
      ;
```

```
qualifier_list
      : qualifier
      | qualifier TK_semi_colon qualifier_list
      ;

/*    New stuff to cope with library directives.
*/

libdir
      : TK_include TK_offside_begin environment TK_offside_end
      | TK_export TK_offside_begin part_s0 TK_offside_end
      | TK_free TK_left_squiggly_bracket signature TK_right_squiggly_bracket
      ;

environment
      : fileid binder_opt alias_s0
      ;

binder
      : TK_left_squiggly_bracket binding binding_s0 TK_right_squiggly_bracket
      ;

binding
      : TK_identifier TK_equal TK_offside_begin exp TK_offside_end
      | tform TK_synonym TK_offside_begin type TK_offside_end
      ;

part
      : TK_identifier
      | fileid
      | TK_plus
      ;

fileid
      : TK_string
      | TK_libpath
      ;

alias
```

```
        : TK_identifier TK_divide TK_identifier
        | TK_Identifier TK_divide TK_Identifier
        | TK_minus TK_identifier
        ;

binder_opt
        : /* empty */
        | binder
        ;

binding_s0
        : /* empty */
        | binding binding_s0
        ;

part_s0
        : /* empty */
        | part part_s0
        ;

alias_s0
        : /* empty */
        | alias alias_s0
        ;
```

Appendix B

Type Checking

B.1 Using the *Typecheck* Program

How to run the `typecheck` and `interact` programs described in Chapter 3. There are two shell scripts in my area that run the system.

- The *typecheck* program.

  ~ml1/typecheck file

  Should attempt to type check the Miranda file named. This program produces a large output file, in the directory of the source file, for use by the next *interact* program.

- The *interact* program.

~ml1/interact file

Allows you to interactively examine the type checked version of Miranda file. This program looks for a file produced by the *typecheck* program.

B.2 Using the *Interact* Program

B.3 Introduction

The *interact* program of Chapter 3 is an interactive type browser for "Miranda"[1] scripts. It should be used in conjunction with the *typecheck* program.

When one runs the *interact* program on a "Miranda" script it looks for the type checked version of that script that would be be produced by the *typecheck* program. If this file does not exist, either in the same directory as the source script or in the current directory, then the program cannot proceed. If the file does exist then we can use the script browsing facilities of the *interact* program to examine the types deduced for *all* the expressions in the script. This includes all the sub-expressions and not simply the top-level definitions.

The script browser views any script as a tree of expressions about which one

[1] Miranda is a Trademark of Research Software Ltd.
In this document we shall use a language resembling Miranda as our example language, and will refer to it as "Miranda". However, this document should not be taken as defining in any way the language Miranda, for such a definition one should see the documents from Research Software Ltd.

may navigate. A script is itself regarded as a single expression where the top-level definitions and specifications in the script form its list of sub-expressions. This tree of expressions view of a script differs from the syntactic structure of scripts and is therefore explained in detail in section §B.4. As one moves around the script one may examine various properties of the current expression. The most interesting property of an expression is the type deduced for it by the *typecheck* program. If the current expression failed to type check the details of the type error can be displayed and by examining its sub-expressions one may attempt to diagnose the cause of the type error.

Sections §B.5, §B.6 and §B.7 explain the commands provided by the *interact* program.

B.4 The Tree Structure of a Script

The following describes the sub-expression structure of a script. We describe this in a top-down manner starting with the script itself. For each form of expression we also detail which of the commands that change the current expression will be appropriate.

The script A script is simply a list of things. The following things may occur at the top-level of a script.

- A library directive; **%export**, **%include** or **%free**

- A top-level function or pattern definition.
- A synonym, algebraic or abstract type definition.
- A type specification for variables or placeholder types.

$$def_1 \cdots def_N$$

The only things that have sub-expressions you can examine are multiple clause function definitions, single clause function definitions and pattern definitions. If the script contains N things then the following commands are appropriate:

$1, \ldots, N$, Find, Down = 1

A multiple clause function definition This is simply a list of the individual clauses of the definition.

$$clause_1 \cdots clause_N$$

If there are N clauses then the following commands are appropriate:

$1, \ldots, N$, Down = 1

If this is not a single sub-definition or the only thing in the script then these commands are also appropriate:

Next, Previous, Find

A single clause function definition This has two sub-expressions, the left hand side formal parameter patterns and the defining right hand side.

$$f\ pat_1 \cdots pat_j = rhs$$

The following commands are appropriate:

Left, Right, Down = Right

If this is not a single sub-definition or the only thing in the script then these commands are also appropriate:

Next, Previous, Find

A pattern definition This has two sub-expressions, the left hand side pattern and the defining right hand side.

$$pat\ = rhs$$

The following commands are appropriate:

Left, Right, Down = Right

If this is not a single sub-definition or the only thing in the script then these commands are also appropriate:

Next, Previous, Find

One clause of a function definition This has two sub-expressions, the left hand side formal parameter patterns and the defining right hand side.

$$f\ pat_1 \cdots pat_j = rhs$$

The following commands are appropriate:

Left, Right, Next, Previous, Down = Right

The left hand side of a definition The left hand side of a single clause function definition, pattern definition or of one clause of a multiple clause function definition is simply an *exp*. The commands are those appropriate for such an expression. (Note: The left hand sides of definitions are treated as applications)

The right hand side of a definition This can be one of three things.

- A right hand side with sub-definitions introduced by a "where" clause.
- A list of alternate right hand side expressions.
- A single right hand side expression.

The commands are those appropriate to whichever of these is found on the right hand side.

A "where" clause This has two sub-expressions, the defining right hand side expressions and the sub-definitions introduced by the "where" clause.

rhs-exp **where** *defs*

The first may be either a list of alternate right hand side expressions or a single right hand side expression. The second may be either a list of sub-definitions or a single sub-definition. The following commands are appropriate:

Left, Right, Down = Left

A list of alternate right hand side expressions This is simply a list of the alternate single right hand side expressions.

$alt_1 \cdots alt_N$

If there are N alternate expressions then the following commands are appropriate:

$1, \ldots, N$, Down = 1

A single right hand side expression This is either a right hand side expression with a guard or simply an *exp*. The commands are those appropriate for whichever of these is found.

A list of sub-definitions This is simply a list of the sub-definitions.

$$def_1 \cdots def_N$$

If there are N sub-definitions then the following commands are appropriate:

$$1, \ldots, N, \text{Down} = 1$$

A single sub-definition This is either a multiple clause function definition, a single clause function definition or a pattern definition. The commands are those appropriate for whichever of these is found.

A right hand side expression with a guard This has two sub-expressions, the guarded expression and the guard.

$$exp \ , \ grd$$

The following commands are appropriate:

$$\text{Left, Right, Down} = \text{Left}$$

exp: **An application** An application is essentially a list of sub-expressions.

$$exp_1 \ exp_2 \cdots exp_N$$

The first is the expression being applied the second the first argument of the application and so on. If there are N sub-expressions, that is $N-1$ arguments, then the following commands are appropriate:

1, ..., N, Down = 1

If the expression above is a list of sub-expressions, including this expression, then these commands will also be appropriate:

Next, Previous

exp: **An infix operator expression** This has two sub-expressions, the left and right operands.

exp-left op exp-right

The following commands are appropriate:

Left, Right, Down = Left

If the expression above is a list of sub-expressions, including this expression, then these commands will also be appropriate:

Next, Previous

exp: **A prefix operator expression** This has a single sub-expression.

op exp

The following command is appropriate:

Down

If the expression above is a list of sub-expressions, including this expression, then these commands will also be appropriate:

Next, Previous

exp: **A tuple** A tuple may be empty in which case it will have no sub-expressions. Otherwise it will have N elements.

$(exp_1, \ldots, exp_N)$

If it is not empty and has N elements then the following commands will be appropriate:

1, ..., N, Down = 1

If the expression above is a list of sub-expressions, including this expression, then these commands will also be appropriate:

Next, Previous

exp: **A list** A list may be empty in which case it will have no sub-expressions. Otherwise it will have N elements.

$[exp_1, \ldots, exp_N]$

If it is not empty and has N elements then the following commands will be appropriate:

1, ..., N, Down = 1

If the expression above is a list of sub-expressions, including this expression, then these commands will also be appropriate:

Next, Previous

exp: **A numeric list expression** This is a list of sub-expressions, it may have one, two or three sub-expressions. If there is only one sub-expression it will be the initial expression.

[*exp*..]

The following commands are appropriate:

1, Down = 1

If there are two sub-expressions they are either the initial and secondary expressions or the initial and limit expressions.

[*exp*, *exp'*..]

[*exp*..*exp''*]

In these cases the following commands are appropriate:

1, 2, Down = 1

The final possibility is that there are three sub-expressions. These being the initial, secondary and limit expressions.

$[exp, exp'..exp'']$

In this case the following commands are appropriate:

1, 2, 3, Down = 1

In all three cases if the expression above is a list of sub-expressions, including this expression, then these commands will also be appropriate:

Next, Previous

exp The following expressions all have no sub-expressions.

An identifier

A constructor

A number

A string

A character

An operator

The keyword **show**

The keyword **otherwise**

If the expression above is a list of sub-expressions, including this expression, then these commands will be appropriate:

Next, Previous

B.5 Input

The *interact* program is an interactive program that keeps prompting the user for commands to be executed. Each input line can contain any number of commands separated by commas. Each command is made up of a sequence of words separated by white space. The syntax of commands is given in sections §B.6 and §B.7. If an inappropriate command is found in a list of commands then all remaining commands from that input line are discarded.

B.6 Command Syntax

We present the syntax of the available commands, the command names can always be abbreviated to the shortest non-ambiguous prefix. We also allow a range of names for each command. The case of letters in command names is irrelevant except for the Quit command which must start with a capital. We shall present all commands with an initial upper case character.

The following twelve commands are general and can be used wherever you are

in the script. They display either general information or information specific to the current expression.

Quit We quit the program.

Help A short help message is given.

Alternate names: ?

Explain A long help message explaining the various commands is given.

Information A description of where we are in the script is provided. This contains a detailed description of the current expression followed by its type if it has one and then details of the enclosing sub-definitions and definition if there are any. If pretty printing is switched on then the current expression will be pretty printed. If commands listing is switched on then the appropriate commands will be listed.

If a command takes you to a new expression then this information is automatically displayed for the new expression.

Alternate names: Where, @

Variables The bound and free variables of this expression are shown. Most expressions in a script will only have free variables.

Type-errors Any type errors that arose whilst type checking this expression and its sub-expressions are displayed.

Show *context?* The current expression is pretty printed. The default is to simply show the current expression. However, if a context is given then the current expression is high-lighted within the pretty printing of the expression designated by that context.

Show On *context?* Pretty printing is switched on for the **Information** command. If a context is given then it is the high-lighted form of pretty printing that is used.

Show Off The pretty printing for the **Information** command is switched off.

Commands The list of appropriate commands for the current expression is displayed.

Commands On The listing of appropriate commands is switched on for the **Information** command.

Commands Off The listing of appropriate commands for the **Information** command is switched off.

The next nine commands are used to move around the script. They will not all be appropriate for all the expressions in the script. The **Commands** command will tell you which are appropriate for the current expression. The basic commands take you to some sub-expression of the current expression. As you descend to a sub-expression the current expression is remembered and that sub-expression becomes

the new current expression. This trace of your path down through the script allows you to back-track up out of an expression.

Up *context?* We can back-track up out of the expression we are in. The simplest version of this command, that with no context, simply takes you to the expression immediately enclosing the current expression. If a context is given then we go up to the enclosing expression designated by that context.

Alternate names: ^

Down *num?* This command will take you the sub-expression which is the "most interesting". Which of the sub-expressions is regarded as the most interesting depends on the particular form of the current expression. When there are other commands that may take you to a sub-expression then this command will be equivalent to one of these other commands. The details of the action of this command are explained in §B.4. If a number is present then that many **Down** commands are attempted.

Alternate names: !

Left *num?* If the current expression has essentially two sub-expressions this command will take you to the first of them. If a number is present then that many **Left** commands are attempted.

Alternate names: <

Right *num?* If the current expression has essentially two sub-expressions this command will take you to the second of them. If a number is present then that many **Right** commands are attempted.

Alternate names: **>**

num If the current expression has a list of sub-expressions this selects one of them as the new expression. A list of sub-expressions is always numbered from one.

Next *num?* If the current expression is an element of a list of expressions then this command takes you to the num^{th} following expression in that list. If the number is absent it defaults to one. The new expression *replaces* the current expression.

Alternate names: **+**

Previous *num?* If the current expression is in a list of expressions then this command takes you to the num^{th} preceding expression in that list. If the number is absent it defaults to one. The new expression *replaces* the current expression.

Alternate names: **-**

Find *name?* If the current expression has a list of sub-expressions that bind identifiers, constructors or type names then this command takes you to the first of these sub-expressions that binds the name given. The only such expressions are the script itself and a list of sub-definitions in a "where" clause. This command will find the first binding occurrence of a name whose prefix is *name*. If a name

is not given then the last name searched for is used.

Furthermore, if the current expression is an element of such a list of binding expressions then this command will find the next binding occurrence of the name given in that list of binding expressions. In this latter case the new expression *replaces* the current expression.

Alternate names: **/**

Again Repeats the last command that changed the current expression.

Alternate names: **&**

B.7 Context Syntax

Each expression in a script can be regarded as existing inside some larger enclosing expressions which provide a context for the expression. The **Show** and **Up** commands allow one to use a *context* to refer to an expression enclosing the current expression. A context is alway taken relative to the current expression and thus may sometimes fail to designate any enclosing expression.

Expression This refers to the largest enclosing expression corresponding to the Miranda syntactic class *exp*.

Alternative This refers to the enclosing alternative right hand side expression.

Clause This refers to the enclosing clause of a function definition or the enclosing pattern definition.

Definition *num?* If the number is absent it defaults to one and this refers to the definition or sub-definition enclosing the current expression. If the number is bigger than one then this refers to the definition or sub-definition that many definitions or sub-definitions above the current expression. If there are not that many then this refers to the enclosing top-level definition.

Script This refers to the top-level expression in the script which is the script itself.

num The expression *num* expressions above this one. This command uses the trace of the route by which we reached the current expression.

B.8 Tutorial Session

The following is a tutorial example of the use of the `interact` program of Chapter 3.

```
% interact tutorial
Interacting with tutorial.m
Please wait for type checked file to be loaded ...
```

If the type checked version of the script has already been compiled then these two lines won't appear.

```
compiling %STATE%tutorial.m
checking types in %STATE%tutorial.m
```

If you have recently invoked the interact program on this source script then these two lines won't appear.

```
compiling /tmp/include.state.m
checking types in /tmp/include.state.m
```

Having loaded the type checked version of the source script we can now browse through it. We first get a description of the script.

```
Miranda Interactive Type Investigator
The script, containing 2 things.
```

We do a Show command to see what the source script looks like.

```
Commands: Show
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
reverse0 (x:l)
        = reverse0 l ++ [x]

reverse0 []
        = []


(test1,test2)
        = (value1,value2)
          where
          value1 = reverse0 [1,2,3]
          value2 = reverse0 "ABC"
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

We find out which commands are appropriate for the current expression.

```
Commands: Commands
The following commands are appropriate:
Help,Down,1,2,Find
```

We go to the first sub-expression in the script.

```
Commands: 1
Doing: 1.
```

We are given a description of the new current expression. Notice that the deduced type for the reverse0 function is polymorphic as we would expect.

```
The 1st thing in the script.
A 2 clause function definition of reverse0.
This expression has the following type:
:: [*] -> [*]
```

We switch on the listing of appropriate commands.

```
Commands: Commands On
Commands now On.
The following commands are appropriate:
Help,Down,1,2,Next,Find
```

We go to the first clause of the function definition. Notice that the appropriate commands are listed.

```
Commands: 1
Doing: 1.
The 1st clause of a function definition.
A single clause of a multiple clause function definition of reverse0.
This expression has the following type:
:: [*] -> [*]
The following commands are appropriate:
Help,Down,Left,Right,Next
```

We show the current expression.

```
Commands: Show
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
reverse0 (x:l)
        = reverse0 l ++ [x]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

We go to the right hand side of this clause.

```
Commands: Right
Doing: Right.
The defining right hand side of a function definition.
A single right hand side expression.
Infix operator expression for "++".
It occurs at the following type:
[*] -> [*] -> [*]
This expression has the following type:
:: [*]
In definition of:
reverse0 (x:l)
The following commands are appropriate:
Help,Down,Left,Right
```

We go to the left operand of the operator.

```
Commands: Left
Doing: Left.
The left operand of an infix application of "++".
An application with 1 argument.
This expression has the following type:
:: [*]
In definition of:
reverse0 (x:l)
The following commands are appropriate:
Help,Down,1,2
```

We switch on showing.

```
Commands: Show On
Showing now On.
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
          reverse0 l
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

We ask for the information about the current expression, this will now include pretty printing of the current expression.

```
Commands: Information
The left operand of an infix application of "++".
An application with 1 argument.
This expression has the following type:
:: [*]
In definition of:
reverse0 (x:l)
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
          reverse0 l
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The following commands are appropriate:
Help,Down,1,2
```

We go to the first argument of the application.

```
Commands: 2
Doing: 2.
The 1st argument of an application.
The identifier: l
This expression has the following type:
:: [*]
In definition of:
```

```
reverse0 (x:l)
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
                    l
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The following commands are appropriate:
Help,Previous
```

We go up to the largest enclosing expression.

```
Commands: Up Expression
Doing: Up Expression.
The defining right hand side of a function definition.
A single right hand side expression.
Infix operator expression for "++".
It occurs at the following type:
[*] -> [*] -> [*]
This expression has the following type:
:: [*]
In definition of:
reverse0 (x:l)
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
        = reverse0 l ++ [x]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The following commands are appropriate:
Help,Down,Left,Right
```

We pretty print the current expression high-lighted within the enclosing definition.

```
Commands: Show Definition
Doing: Show Definition.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
reverse0 (x:l)
@@@@@@@@@
        = reverse0 l ++ [x]
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

```
reverse0 []
        = []
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

We switch showing off.

```
Commands: Show Off
Showing now Off.
```

We go up to the top-level of the script.

```
Commands: Up Script
Doing: Up Script.
The script, containing 2 things.
The following commands are appropriate:
Help,Down,1,2,Find
```

We go to the second definition in the script.

```
Commands: 2
Doing: 2.
The 2nd thing in the script.
A pattern definition.
It declares the following identifiers:
test1, test2
This expression has the following type:
:: ([num],[char])
The following commands are appropriate:
Help,Down,Left,Right,Previous,Find
```

We go to its defining right hand side.

```
Commands: Right
Doing: Right.
The defining right hand side of a pattern definition.
A right hand side with sub-definitions introduced by a "where" clause.
This expression has the following type:
:: ([num],[char])
In definition of:
(test1,test2)
The following commands are appropriate:
Help,Down,Left,Right
```

We wish to look at the sub-definitions.

```
Commands: Right
Doing: Right.
A list of 2 sub-definitions.
In definition of:
(test1,test2)
The following commands are appropriate:
Help,Down,1,2,Find
```

We go to the first sub-definition.

```
Commands: 1
Doing: 1.
The 1st sub-definition.
A pattern definition.
It declares the following identifiers:
value1
This expression has the following type:
:: [num]
In definition of:
(test1,test2)
The following commands are appropriate:
Help,Down,Left,Right,Next,Find
```

We switch on showing.

```
Commands: Show On
Showing now On.
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
          value1 = reverse0 [1,2,3]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

We go to the defining right hand side.

```
Commands: Right
Doing: Right.
The defining right hand side of a pattern definition.
A single right hand side expression.
An application with 1 argument.
This expression has the following type:
:: [num]
In sub-definition of:
          value1
In definition of:
(test1,test2)
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
                  = reverse0 [1,2,3]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The following commands are appropriate:
Help,Down,1,2
```

We go to the function being applied. Notice that at this application the function reverse0 takes an instance of its deduced type.

```
Commands: 1
Doing: 1.
An expression that is applied to 1 argument.
The identifier: reverse0
This expression has the following type:
:: [num] -> [num]
```

```
In sub-definition of:
          value1
In definition of:
(test1,test2)
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
                    reverse0
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The following commands are appropriate:
Help,Next
```

We go up to the enclosing definition.

```
Commands: Up Definition
Doing: Up Definition.
The 1st sub-definition.
A pattern definition.
It declares the following identifiers:
value1
This expression has the following type:
:: [num]
In definition of:
(test1,test2)
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
          value1 = reverse0 [1,2,3]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The following commands are appropriate:
Help,Down,Left,Right,Next,Find
```

We go the next sub-definition.

```
Commands: Next
Doing: Next.
The 2nd sub-definition.
A pattern definition.
It declares the following identifiers:
```

```
value2
This expression has the following type:
:: [char]
In definition of:
(test1,test2)
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
          value2 = reverse0 "ABC"
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The following commands are appropriate:
Help,Down,Left,Right,Previous,Find
```

We now do two commands on one line to get to the second application of the reverse0 function. Notice again that at this application the function reverse0 takes an instance of its deduced type.

```
Commands: Right,1
Doing: Right.
The defining right hand side of a pattern definition.
A single right hand side expression.
An application with 1 argument.
This expression has the following type:
:: [char]
In sub-definition of:
          value2
In definition of:
(test1,test2)
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
                  = reverse0 "ABC"
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The following commands are appropriate:
Help,Down,1,2
Doing: 1.
An expression that is applied to 1 argument.
The identifier: reverse0
This expression has the following type:
```

```
:: [char] -> [char]
In sub-definition of:
          value2
In definition of:
(test1,test2)
Doing: Show.
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
                    reverse0
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The following commands are appropriate:
Help,Next
```

We quit the interact program.

```
Commands: Quit
Quitting.
```

Appendix C

Type Checking Proofs

We describe the approach to type checking proofs mentioned in Chapter 4. I first describe the type system used by LCF and why I think certain of the decisions were made. I then present a simpler approach.

C.1 LCF

LCF uses a rather strange form of type checking. The basic decision which seems to have forced the other choices made in the LCF system, is to not allow anonymous type variables.

The type checker needs to generate anonymous type variables when type checking and has to make up its own names for these type variables. The LCF system insists that the user supply explicit names for these internally generated type variables.

The user does this by adding type annotations to the expressions they input into the system in such a way as to allow the system to associate explicit type variable names with the internally generated type variables. Exactly what type annotations are needed for an expression can be quite hard to decide and the temptation is to initially supply none and add them, in a fairly arbitrary fashion, only if the system complains.

This need to constantly supply type annotations is very irritating so the system includes the concept of "sticky" types. This means that the system remembers the type of the last use of an identifier and failing any explicit type information in the expression will use this type. We therefore have a fairly complex type environment in which expressions are type checked and this can cause unexpected problems. Apparently in the Edinburgh LCF system these "sticky" types can cause the type checking to fail, a problem which is remedied in Larry Paulson's Cambridge LCF system.

A final decision which adds to the confusion is to let all the free variables in a proof take different types at their different occurrences[1]. Thus unless you know their types you cannot tell whether two occurrences of a free variable name are in fact the same variable. This decision is useful from a practical point of view as it means you can put expressions together without having to worry about the types of their free variables being consistent.

[1] **let** bindings.

All the above provides a very complex and I believe confusing type system. The only case where you seem to need access to explicit names for internally generated type variables is in rule which allows instantiation of the types of a theorem.

C.2 A Simpler Approach

C.2.1 Type Checking

I propose to use the full power of the Milner type checking algorithm. The user need provide no type information at all, the system will deduce the most general type for any expression and hence for any proof. This means we won't know which type variable names will be used for the internally generated type variables by the type checking, but then we don't need to.

When constructing a new proof we must now consider the free variables of the sub-proofs. I propose that the free variables of a proof take the same type at all their occurrences in the proof[2]. This is to avoid the confusion about whether or not occurrences of a variable name refer to the same variable. This now demands that the the type checker do more work and also that you have to be a bit careful about your use of variable names but neither of these is a problem.

The type checker must now check that a variable appearing free in the sub-proofs

[2]**lambda** bindings

has types, in these sub-proofs, that can be unified. If these types don't unify the construction of the new proof will fail. The user should not find this clash of free variables a problem as a simple renaming of the offending variable in some of the sub-proofs will remove the problem[3].

There is one final point to notice, if there are free variables occurring in the sub-proofs, but not in the assumptions or conclusions of the sub-proofs, then if there was a type clash on these free variables the construction of the new proof would fail. If however we took the theorem form of the sub-proofs, where we only have the assumptions and conclusions, then the construction of the new proof would succeed. A more subtle problem can also arise where the proofs constructed from proofs and theorems both succeed but that based on proofs has a conclusion of more restricted type. This may be an argument for always dealing with entire proofs.

This is all you need to do and it provides a clear simple type discipline.

C.2.2 Display

As a practical matter I believe there is a way of presenting the type information in a proof without displaying the type of everything. We can annotate the variables and constants with types as follows.

1. All bound variables are annotated with their types at their binding occurrence.

[3]We will of course have to check for capture of the new variable and if it already occurs free in a sub-proof that its type and that of the variable being replaced can be unified

2. All the free variables in the assumptions are annotated with their types at all their occurrences in the assumptions, in the rest of the proof they have no type annotations.

3. All free variables in the proof not occurring free in the assumptions are annotated with their types at all their occurrences in the proof.

4. Constants can be annotated with their types at all their occurrences in the proof, or be completely without type annotations. I believe we can take the second course as the type environment along with the generic type that the user must have provided for the constant should be enough.

Bibliography

[AJ89] L. Augustsson and T Johnsson.

The Chalmers lazy-ML compiler.

The Computer Journal, 32(2):127–141, 1989.

[AKP85] Arvind, Vinod Kathail, and Keshav Pingali.

Sharing of computation in functional language implementations.

In Lennart Augustsson, John Hughes, Thomas Johnsson, and Kent Karlsson, editors, *Proceedings of the Workshop on Implementation of Functional Languages*, pages 15–58. Programming Methodology Group, February 1985.

[Ale88] Heather Alexander.

Comments on "Formal specification of user interfaces: A comparison and evaluation of four axiomatic approaches".

IEEE Transactions on Software Engineering, 14(4):438–439, April 1988.

[BB85] Erret Bishop and Douglas Bridges.

Constructive Analysis.

Springer-Verlag, 1985.

[BD77] R.M. Burstall and John Darlington.

A transformation system for developing recursive programs.

Journal of the ACM, 24(1):44–67, January 1977.

[BGS0] R.M. Burstall and J.A. Goguen.

The semantics of Clear, a specification language.

In *Proceedings 1979 Copenhagen Winter School on Abstract Software Specification*, February 1980.

[BH87] R. S. Bird and John Hughes.

The alpha-beta algorithm: An exercise in program transformation.

Information Processing Letters, 24:53–57, January 1987.

[Bir80] R.S. Bird.

Tabulation techniques for recursive programs.

ACM Computing Surveys, 12(4):403–417, December 1980.

[Bir84a] R.S. Bird.

The promotion and accumulation strategies in transformational programming.

ACM Transactions on Programming Languages and Systems, 6(4):487–504, October 1984.

[Bir84b] R.S. Bird.

Using circular programs to eliminate data traversal.

Acta Informatica, 21:239–250, 1984.

[Bir85] R.S. Bird.

Addendum : The promotion and accumulation strategies in transformational programming.

ACM Transactions on Programming Languages and Systems, 6(3):490–492, July 1985.

[Bir89] R. S. Bird.

Algebraic identities for program calculation.

The Computer Journal, 32(2):122–126, 1989.

[BM75] R.S. Boyer and J.S. Moore.

Proving theorems about LISP functions.

Journal of the ACM, 22(1):129–144, January 1975.

[Boe89] Hans-J. Boehm.

Type inference in the presence of type abstraction.

ACM SIGPLAN Notices, 24(7):192–206, July 1989.

[Chi85] Uli H. Chi.

Formal specification of user interfaces: A comparison and evaluation of four axiomatic approaches.

IEEE Transactions on Software Engineering, 11(8):671–685, August 1985.

[CW85] Luca Cardelli and Peter Wegner.

On understanding types, data abstraction, and polymorphism.

ACM Computing Surveys, 17(4):471–522, December 1985.

[Dam85] Luis Manuel Martins Damas.

Type assignment in programming languages.

Thesis CST–33–85, University of Edinburgh, April 1985.

[Dij76] E. W. Dijkstra.

A Discipline of Programming.

Prentice-Hall, 1976.

[dV88] Fer-Jan de Vries.

A functional program for the fast Fourier transform.

ACM SIGPLAN Notices, 23(1):67–74, January 1988.

[Dwe89] Andrew Dwelly.

Functions and dynamic user interfaces.

In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 371–381, September 1989.

[Fai82] Jon Fairbairn.

Ponder and its type system.

Computer Laboratory Technical Report 31, University of Cambridge, November 1982.

[Fai87] Jon Fairbairn.

Making form follow function: An exercise in functional programming style.

Software — Practice and Experience, 17(6):379–386, June 1987.

[FL89] R. Frost and J Launchbury.

Constructing natural language interpreters in a lazy functional language.

The Computer Journal, 32(2):108–121, 1989.

[Fle90] A. C. Fleck.

A case study comparison of four declarative programming languages.

Software — Practice and Experience, 20(1):49–65, January 1990.

[GMW79] M. J. C. Gordon, R. Milner, and C. P. Wadsworth.

Edinburgh LCF: A Mechanised Logic of Computation, volume 78 of *LNCS*.

Springer Verlag, 1979.

[HHP87] R. Harper, F. Honsell, and G. Plotkin.

A framework for defining logics.

In *Proceedings of the Second Symposium on Logic in Computer Science*, 1987.

[Hil89] Stephen A. Hill.

Functional Programming Techniques.

PhD thesis, University of Kent at Canterbury, 1989.

[Hin69] R. Hindley.

The principal type scheme of an object in combinatory logic.

Transactions of the American Mathematical Society, 146:29–60, 1969.

[HSW85] David M. Harland, Martyn W. Szyplewski, and John B. Wainwright.

An alternative view of polymorphism.

ACM SIGPLAN Notices, 20(10):23–35, October 1985.

[Hud89] Paul Hudak.

Conception, evolution and application of functional programming languages.

ACM Computing Surveys, 21(3):359–411, September 1989.

[Hug89] J. Hughes.

Why functional programming matters.

The Computer Journal, 32(2):98–107, 1989.

[Joh85] Thomas Johnsson.

Lambda lifting: Transforming programs to recursive equations.

In Lennart Augustsson, John Hughes, Thomas Johnsson, and Kent Karlsson, editors, *Proceedings of the Workshop on Implementation of Functional Languages*, pages 165–180. Programming Methodology Group, February 1985.

[Joh87] Thomas Johnsson.

Attribute grammars as a functional programming paradigm.

In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture: LNCS 274*, pages 154–173. Springer Verlag, 1987.

[Jok89] Matti O. Jokinen.

A language-independent prettyprinter.

Software — Practice and Experience, 19(9):839–856, September 1989.

[Jon85] Simon L. Peyton Jones.

Yacc in Sasl — an exercise in functional programming.

Software — Practice and Experience, 15(8):807–820, 1985.

[Jon86a] C. B. Jones.

Systematic Software Development Using VDM.

Prentice-Hall, 1986.

[Jon86b] Richard Jones.

Flex — an experience of miranda.

UKC Computing Laboratory Report 38, University of Kent at Canterbury, June 1986.

[Jon87] Simon L. Peyton Jones.

The Implementation of Functional Programming Languages.

International Series in Computer Science. Prentice-Hall, 1987.

[Kie85] Richard B. Kieburtz.

A proposal for interactive debugging of ML programs.

In Lennart Augustsson, John Hughes, Thomas Johnsson, and Kent Karlsson, editors, *Proceedings of the Workshop on Implementation of Functional Languages*, pages 151–155. Programming Methodology Group, February 1985.

[Koo87] Pieter W. M. Koopman.

Interactive programs in a functional language: A functional implementation of an editor.

Software — Practice and Experience, 17(9):609–622, September 1987.

[Lon87a] M. Longley.

Generating parsers in Miranda.

UKC Computing Laboratory Report 49, University of Kent at Canterbury, November 1987.

[Lon87b] M. Longley.

Type checking "Miranda".

UKC Computing Laboratory Report 44, University of Kent at Canterbury, April 1987.

[Lon89] M. Longley.

Continuations → Continuations = Interactions.

UKC Computing Laboratory Report 59, University of Kent at Canterbury, March 1989.

[MG90] Carroll Morgan and P. H. B. Gardiner.

Data refinement by calculation.

Acta Informatica, 27:481–503, 1990.

[Mor88] Carroll Morgan.

The specification statement.

ACM Transactions on Programming Languages and Systems, 10(3):403–419, July 1988.

[Myc84] Alan Mycroft.

Polymorphic type schemes and recursive definitions.

In *International Symposium on Programming: LNCS 167*, pages 217–228. Springer Verlag, 1984.

[NRR86] Klaus Nökel, Robert Rehbold, and Michael M. Richter.

Remarks on SASL and the verification of functional programmaing languages.

In *Computation Theory and Logic: LNCS 270*, pages 265–276. Springer Verlag, 1986.

[O'D85] John T. O'Donnell.

Dialogues: A basis for constructing programming environments.

ACM SIGPLAN Notices, 20(7):19–27, July 1985.

Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments.

[OG89] James William O'Toole Jr. and David K. Gifford.

Type reconstruction with first-class polymorphic values.

ACM SIGPLAN Notices, 24(7):207–217, July 1989.

[Opp80] Derek C. Oppen.

Prettyprinting.

ACM Transactions on Programming Languages and Systems, 2(4):465–483, October 1980.

[Pra73] Dag Prawitz.

Ideas and results in proof theory.

In Jens E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–307. North Holland, 1973.

[Rob65] J. A. Robinson.

A machine-oriented logic based on the resolution principle.

Journal of the ACM, 12(1):23–41, 1965.

[Rub83] Lisa F. Rubin.

Syntax-directed pretty printing — a first step towards a syntax-directed editor.

IEEE Transactions on Software Engineering, 9(2):119–127, March 1983.

[Spi85] John Michael Spivey.

Understanding Z : A Specification Language and Its Formal Semantics.

PhD thesis, Wolfson College Oxford, 1985.

[TDR87] Ian Toyn, Alan Dix, and Colin Runciman.

Performance polymorphism.

In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture: LNCS 274*, pages 325–346. Springer Verlag, 1987.

[TR85] Ian Toyn and Colin Runciman.

Adapting combinator and SECD machines to display snapshots of functional computations.

Computer Science Report 79, University of York, 1985.

[Tur79] D.A. Turner.

A new implementation technique for applicative languages.

Software — Practice and Experience, 9(1):31–49, January 1979.

[Tur81] D.A. Turner.

Recursion equations as a programming language.

In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications*, pages 1–28. Cambridge University Press, 1981.

[Tur85] D.A. Turner.

Miranda — a non-strict functional language with polymorphic types.

In Jouannand, editor, *Functional Programming Languages and Computer Architecture: LNCS 201*, pages 1–16. Springer Verlag, September 1985.

[Wad85] Philip Wadler.

Views: A way for elegant definitions and efficient representations to coexist.

In Lennart Augustsson, John Hughes, Thomas Johnsson, and Kent Karlsson, editors, *Proceedings of the Workshop on Implementation of Functional Languages*, pages 247–258. Programming Methodology Group, February 1985.

[WF89] S. C. Wray and J. Fairbairn.

Non-strict languages — programming and implementation.

The Computer Journal, 32(2):142–151, 1989.