FROM SIGPLAN NOTICES, 21(12):158-166, December 1986. ©D.A.Turner

An Overview of Miranda

David Turner
Computing Laboratory
University of Kent
Canterbury CT2 7NF
ENGLAND

Miranda is an advanced functional programming system which runs under the UNIX operating system[1]. The aim of the Miranda system is to provide a modern functional language, embedded in a convenient programming environment, suitable both for teaching and as a general purpose programming tool. The purpose of this short article is to give a brief overview of the main features of Miranda. The topics we shall discuss, in order, are:

Basic ideas
The Miranda programming environment
Guarded equations and block structure
Pattern matching
Currying and higher order functions
List comprehensions
Lazy evaluation and infinite lists
Polymorphic strong typing
User defined types
Type synonyms
Abstract data types
Separate compilation and linking
Current implementation status

Basic ideas

The Miranda programming language is purely functional – there are no side effects or imperative features of any kind. A program (actually we don't call it a program, we call it a "script") is a collection of equations defining various functions and data structures which we are interested in computing. The order in which the equations are given is not in general significant. There is for

[1]UNIX is a trademark of AT&T Bell Laboratories, Miranda is a trademark of Research Software Ltd.

example no obligation for the definition of an entity to precede its first use. Here is a very simple example of a Miranda script:

```
z = sq x / sq y
sq n = n * n
x = a + b
y = a - b
a = 10
b = 5
```

Notice the absence of syntactic baggage – Miranda is, by design, rather terse. There are no mandatory type declarations, although (see later) the language is strongly typed. There are no semicolons at the end of definitions – the parsing algorithm makes intelligent use of layout. Note that the notation for function application is simply juxtaposition, as in "`sq x`". In the definition of the `sq` function, "`n`" is a formal parameter – its scope is limited to the equation in which it occurs (whereas the other names introduced above have the whole script for their scope).

The most commonly used data structure is the list, which in Miranda is written with square brackets and commas, eg:

```
week_days = ["Mon","Tue","Wed","Thur","Fri"]
days = week_days ++ ["Sat","Sun"]
```

Lists may be appended by the "`++`" operator. Other useful operations on lists include infix "`:`" which prefixes an element to the front of a list, "`#`" which takes the length of a list, and infix "`!`" which does subscripting. So for example `0:[1,2,3]` has the value `[0,1,2,3]`, `#days` is 7, and `days!0` is `"Mon"`.

There is also an operator "`--`" which does list subtraction. For example `[1,2,3,4,5] -- [2,4]` is `[1,3,5]`.

There is a shorthand notation using "`..`" for lists whose elements form an arithmetic series. Here for example are definitions of the factorial function, and of a number "`result`" which is the sum of the odd numbers between 1 and 100 (`sum` and `product` are library functions):

```
fac n = product [1..n]
result = sum [1,3..100]
```

The elements of a list must all be of the same type. A sequence of elements of mixed type is called a tuple, and is written using parentheses instead of square brackets. Example

```
employee = ("Jones",True,False,39)
```

Tuples are analogous to records in Pascal (whereas lists are analogous to arrays). Tuples cannot be subscripted – their elements are extracted by pattern matching (see later).

The programming environment

The Miranda system is interactive and runs under UNIX as a self contained subsystem. The basic action is to evaluate expressions, supplied by the user at the terminal, in the environment established by the current script. For example evaluating "z" in the context of the first script given above would produce the result "9.0".

The Miranda compiler works in conjunction with an editor (by default this is "vi" but it can be set to any editor of the user's choice). Scripts are automatically recompiled after edits, and any syntax or type errors signalled immediately. The polymorphic type system permits a high proportion of logical errors to be detected at compile time.

There is quite a large library of standard functions. There is also an online reference manual. The interface to UNIX permits Miranda programs to take data from, and send data to, UNIX files and it is also possible to invoke Miranda programs directly from the UNIX shell and to combine them, via UNIX pipes, with processes written in other languages.

Guarded equations and block structure

An equation can have several alternative right hand sides distinguished by "guards" – the guard is written on the right following a comma. For example the greatest common divisor function can be written:

```
gcd a b = gcd (a-b) b, if a>b
        = gcd a (b-a), if a<b
        = a,           if a=b
```

The last guard in such a series of alternatives can be written "**otherwise**", instead of "**if** condition", to indicate a default case[2].

It is also permitted to introduce local definitions on the right hand side of a definition, by means of a "**where**" clause. Consider for example the following definition of a function for solving quadratic equations (it either fails or returns a list of one or two real roots):

```
quadsolve a b c = error "complex roots",    if delta<0
                = [-b/(2*a)],               if delta=0
                = [-b/(2*a) + radix/(2*a),
                   -b/(2*a) - radix/(2*a)], if delta>0
                  where
                  delta = b*b - 4*a*c
                  radix = sqrt delta
```

[2]In early versions of Miranda they keyword "**if**" was not required.

Where clauses may occur nested, to arbitrary depth, allowing Miranda programs to be organised with a nested block structure. Indentation of inner blocks is compulsory, as layout information is used by the parser.

Pattern matching

It is permitted to define a function by giving several alternative equations, distinguished by the use of different patterns in the formal parameters. This provides another method of doing case analysis which is often more elegant than the use of guards. We here give some simple examples of pattern matching on natural numbers, lists and tuples. Here is (another) definition of the factorial function, and a definition of Ackermann's function:

```
fac 0 = 1
fac (n+1) = (n+1) * fac n

ack 0 n = n+1
ack (m+1) 0 = ack m 1
ack (m+1) (n+1) = ack m (ack (m+1) n)
```

Here is a (naive) definition of a function for computing the n'th Fibonacci number:

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

Here are some simple examples of functions defined by pattern matching on lists:

```
sum [] = 0
sum (a:x) = a + sum x

product [] = 1
product (a:x) = a * product x

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

Accessing the elements of a tuple is also done by pattern matching. For example the selection functions on 2-tuples can be defined thus

```
fst (a,b) = a
snd (a,b) = b
```

As final examples we give the definitions of two Miranda library functions, take and drop, which return the first n members of a list, and the rest of the list without the first n members, respectively

```
take 0 x = []
take (n+1) [] = []
take (n+1) (a:x) = a : take n x

drop 0 x = x
drop (n+1) [] = []
drop (n+1) (a:x) = drop n x
```

Notice that the two functions are defined in such a way that that the following identity always holds – "`take n x ++ drop n x = x`" - including in the pathological case that the length of `x` is less than `n`.

Currying and higher order functions

Miranda is a fully higher order language – functions are first class citizens and can be both passed as parameters and returned as results. Function application is left associative, so when we write "`f x y`" it is parsed as "`(f x) y`", meaning that the result of applying f to x is a function, which is then applied to y. The reader may test out his understanding of higher order functions by working out what is the value of "`answer`" in the following script:

```
answer = twice twice twice suc 0
twice f x = f (f x)
suc x = x + 1
```

Note that in Miranda every function of two or more arguments is actually a higher order function. This is very useful as it permits partial application. For example "`member`" is a library function such that "`member x a`" tests if the list `x` contains the element `a` (returning `True` or `False` as appropriate). By partially applying member we can derive many useful predicates, such as

```
vowel = member ['a','e','i','o','u']
digit = member ['0','1','2','3','4','5','6','7','8','9']
month = member ["Jan","Feb","Mar","Apr","Jun","Jul","Aug",
                "Sep","Oct","Nov","Dec"]
```

As another example of higher order programming consider the function foldr, defined

```
foldr op k [] = k
foldr op k (a:x) = op a (foldr op k x)
```

All the standard list processing functions can be obtained by partially applying foldr. Examples

```
sum = foldr (+) 0
product = foldr (*) 1
reverse = foldr postfix []
          where postfix a x = x ++ [a]
```

List comprehensions

List comprehensions give a concise syntax for a rather general class of iterations over lists. The syntax is adapted from an analogous notation used in set theory (called “set comprehension”). A simple example of a list comprehension is:

```
[ n*n | n <- [1..100] ]
```

This is a list containing (in order) the squares of all the numbers from 1 to 100. The above expression can be read as “list of all `n*n` such that `n` is drawn from the list 1 to 100”. Note that “`n`” is a local variable of the above expression. The variable-binding construct to the right of the bar is called a “generator” – the “`<-`” sign denotes that the variable introduced on its left ranges over all the elements of the list on its right. The general form of a list comprehension in Miranda is:

```
[ body | qualifiers ]
```

Each qualifier is either a generator, of the form `var<-exp`, or else a filter, which is a boolean expression used to restrict the ranges of the variables introduced by the generators. When two or more qualifiers are present they are separated by semicolons. An example of a list comprehension with two generators is given by the following definition of a function for returning a list of all the permutations of a given list,

```
perms [] = [[]]
perms x  = [ a:y | a <- x; y <- perms (x--[a]) ]
```

The use of a filter is shown by the following definition of a function which takes a number and returns a list of all its factors,

```
factors n = [ i | i <- [1..n div 2]; n mod i = 0 ]
```

List comprehensions often allow remarkable conciseness of expression. We give two examples. Here is a Miranda statement of Hoare’s “Quicksort” algorithm, as a method of sorting a list,

```
sort [] = []
sort (a:x) = sort [ b | b <- x; b<=a ]
             ++ [a] ++
             sort [ b | b <- x; b>a ]
```

Next is a Miranda solution to the eight queens problem. We have to place eight queens on chess board so that no queen gives check to any other. Since any solution must have exactly one queen in each column, a suitable representation for a board is a list of integers giving the row number of the queen in each successive column. In the following script the function “`queens n`” returns all safe ways to place queens on the first n columns. A list of all solutions to the eight queens problem is therefore obtained by printing the value of (queens 8)

```
queens 0 = [[]]
queens (n+1) = [q:b | b <- queens n; q <- [0..7]; safe q b]
safe q b = and [ ~checks q b i | i <- [0..#b-1] ]
checks q b i =  q = b!i \/ abs (q - b!i) = i+1
```

Lazy evaluation and infinite lists

Miranda's evaluation mechanism is "lazy", in the sense that no subexpression is evaluated until its value is known to be required. One consequence of this is that is possible to define functions which are non-strict (meaning that they are capable of returning an answer even if one of their arguments is undefined). For example we can define a conditional function as follows,

```
cond True x y = x
cond False x y = y
```

and then use it in such situations as "`cond (x=0) 0 (1/x)`".

The other main consequence of lazy evaluation is that it makes it possible to write down definitions of infinite data structures. Here are some examples of Miranda definitions of infinite lists (note that there is a modified form of the ".." notation for endless arithmetic progressions)

```
ones = 1 : ones
repeat a = x
           where x = a : x
nats = [0..]
odds = [1,3..]
squares = [ n*n | n <- [0..] ]
perfects = [ n | n <- [1..]; sum (factors n) = n ]
primes = sieve [ 2.. ]
         where
         sieve (p:x) = p : sieve [n | n <- x; n mod p > 0]
```

One interesting application of infinite lists is to act as lookup tables for caching the values of a function. For example our earlier naive definition of "fib" can be improved from exponential to linear complexity by changing the recursion to use a lookup table, thus

```
fib 0 = 1
fib 1 = 1
fib (n+2) = flist!(n+1) + flist!n
            where
            flist = map fib [ 0.. ]
```

Another important use of infinite lists is that they enable us to write functional programs representing networks of communicating processes. Consider for example the Hamming numbers problem – we have to print in ascending order all numbers of the form $2^a \times 3^b \times 5^c$, for $a, b, c \geq 0$. There is a nice solution to this problem in terms of communicating processes, which can be expressed in Miranda as follows

```
hamming = 1 : merge (f 2) (merge (f 3) (f 5))
          where
          f a = [ n*a | n <- hamming ]
          merge (a:x) (b:y) = a : merge x (b:y), if a<b
                            = b : merge (a:x) y, if a>b
                            = a : merge x y,     otherwise
```

Polymorphic strong typing

Miranda is strongly typed. That is, every expression and every subexpression has a type, which can be deduced at compile time, and any inconsistency in the type structure of a script results in a compile time error message. We here briefly summarise Miranda's notation for its types.

There are three primitive types, called num, bool, and char. The type num comprises integer and floating point numbers (the distinction between integers and floating point numbers is handled at run time – this is not regarded as being a type distinction). There are two values of type bool, called True and False. The type char comprises the Latin-1 character set – character constants are written in single quotes, using C escape conventions, e.g. `'a'`, `'$'`, `'\n'` etc.

If `T` is type, then `[T]` is the type of lists whose elements are of type `T`. For example `[[1,2],[2,3],[4,5]]` is of type `[[num]]`, that is list of lists of numbers. String constants are of type `[char]`, in fact a string such as `"hello"` is simply a shorthand way of writing `['h','e','l','l','o']`.

If `T1` to `Tn` are types, then `(T1, ··· ,Tn)` is the type of tuples with objects of these types as components. For example `(True,"hello",36)` is of type `(bool,[char],num)`.

If `T1` and `T2` are types, then `T1->T2` is the type of a function with arguments in `T1` and results in `T2`. For example the function `sum` is of type `[num]->num`. The function `quadsolve`, given earlier, is of type `num->num->num->[num]`. Note that "`->`" is right associative.

Miranda scripts can include type declarations. These are written using "::" to mean *is of type*. Example

```
sq :: num -> num
sq n = n * n
```

The type declaration is not necessary, however. The compiler is always able to deduce the type of an identifier from its defining equation. Miranda scripts often contain type declarations as these are useful for documentation (and they provide an extra check, since the typechecker will complain if the declared type is inconsistent with the inferred one).

Types can be polymorphic, in the sense of Milner [1978]. This is indicated by using the symbols `*`, `**`, `***`, etc. as an alphabet of generic type variables. For example, the identity function, defined in the Miranda library as

```
id x = x
```

has the following type

```
id :: * -> *
```

this means that the identity function has many types. Namely all those which can be obtained by substituting an arbitrary type for the generic type variable, eg "`num->num`", "`bool->bool`", "`(*->**) -> (*->**)`" and so on.

We illustrate the Miranda type system by giving types for some of the functions so far defined in this article

```
fac :: num -> num
ack :: num -> num -> num
sum :: [num] -> num
month :: [char] -> bool
reverse :: [*] -> [*]
fst :: (*,**) -> *
snd :: (*,**) -> **
foldr :: (*->**->**) -> ** -> [*] -> **
perms :: [*] -> [[*]]
```

User defined types

The user may introduce new types. This is done by an equation in "`::=`". For example a type of labelled binary trees (with numeric labels) would be introduced as follows,

```
tree ::= Nilt | Node num tree tree
```

This introduces three new identifiers – "`tree`" which is the name of the type, and "`Nilt`" and "`Node`" which are the constructors for trees – note that constructors must begin with an upper case letter. Nilt is an atomic constructor, while Node takes three arguments, of the types shown. Here is an example of a tree built using these constructors

```
t1 = Node 7 (Node 3 Nilt Nilt) (Node 4 Nilt Nilt)
```

To analyse an object of user defined type, we use pattern matching. For example here is a definition of a function for taking the mirror image of a tree

```
mirror Nilt = Nilt
mirror (Node a x y) = Node a (mirror y) (mirror x)
```

User defined types can be polymorphic – this is shown by introducing one or more generic type variables as parameters of the "`::=`" equation. For example we can generalise the definition of tree to allow arbitrary labels, thus

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

this introduces a family of tree types, including `tree num`, `tree bool`, `tree (char->char)`, and so on[3].

The types introduced by "`::=`" definitions are called "algebraic types". Algebraic types are a very general idea. They include scalar enumeration types, eg

```
color ::= Red | Orange | Yellow | Green | Blue |
          Indigo | Violet
```

and also give us a way to do union types, for example

```
bool_or_num ::= Left bool | Right num
```

It is interesting to note that all the basic data types of Miranda could be defined from first principles, using "`::=`" equations. For example here are type definitions for bool, (natural) numbers and lists,

```
bool ::= True | False
nat ::= Zero | Suc nat
list * ::= Nil | Cons * (list *)
```

Having types such as "`num`" built in is done for reasons of efficiency - it isn't logically necessary.

[3]In versions of Miranda before release two (1989) it was possible to associate "laws" with the constructors of an algebraic type, which are applied whenever an object of the type is built. For details see Turner [1985], Thompson [1986]. This feature was little used and has since been removed from the language.

Type synonyms

The Miranda programmer can introduce a new name for an already existing type. We use "==" for these definitions, to distinguish them from ordinary value definitions. Examples

```
        string == [char]
        matrix == [[num]]
```

Type synonyms are entirely transparent to the typechecker – it is best to think of them as macros. It is also possible to introduce synonyms for families of types. This is done by using generic type symbols as formal parameters, as in

```
        array * == [[*]]
```

so now eg 'array num' is the same type as 'matrix'.

Abstract data types

In addition to concrete types, introduced by "::=" or "==" equations, Miranda permits the definition of abstract types, whose implementation is "hidden" from the rest of the program. To show how this works we give the standard example of defining stack as an abstract data type (here based on lists):

```
        abstype stack *
        with  empty :: stack *
              isempty :: stack * -> bool
              push :: * -> stack * -> stack *
              pop :: stack * -> stack *
              top :: stack * -> *

        stack * == [*]
        empty = []
        isempty x = (x=[])
        push a x = (a:x)
        pop (a:x) = x
        top (a:x) = a
```

We see that the definition of an abstract data type consists of two parts. First a declaration of the form "**abstype** ... **with** ...", where the names following the "**with**" are called the *signature* of the abstract data type. These names are the interface between the abstract data type and the rest of the program. Then a set of equations giving bindings for the names introduced in the abstype declaration. These are called the *implementation equations*.

The type abstraction is enforced by the typechecker. The mechanism works as follows. When typechecking the implementation equations the abstract type and its representation are treated as being the same type. In the whole of the rest of the script the abstract type and its representation are treated as two separate and completely unrelated types. This is somewhat different from the usual mechanism for implementing abstract data types, but has a number of advantages. It is discussed at somewhat greater length in [Turner 85].

Separate compilation and linking

The basic mechanisms for separate compilation and linking are extremely simple. Any Miranda script can contain one or more directives of the form

```
%include "pathname"
```

where `"pathname"` is the name of another Miranda script file (which might itself contain include directives, and so on recursively - cycles in the include structure are not permitted however). The visibility of names to an including script is controlled by a directive in the included script, of the form

```
%export names
```

It is permitted to export types as well as values. It is not permitted to export a value to a place where its type is unknown, so if you export an object of a locally defined type, the typename must be exported also. Exporting the name of a "`::=`" type automatically exports all its constructors. If a script does not contain an export directive, then the default is that all the names (and typenames) it defines will be exported (but not those which it acquired by "%include" statements).

It is also permitted to write a *parametrised script*, in which certain names and/or typenames are declared as "free". An example is that we might wish to write a package for doing matrix algebra without knowing what the type of the matrix elements are going to be. A header for such a package could look like this:

```
%free { element :: type
        zero, unit :: element
        mult, add, subtract,
                   divide :: element->element->element
      }

%export matmult determinant eigenvalues eigenvectors ...
|| here would follow definitions of matmult, determinant,
|| eigenvalues, etc. in terms of the free identifiers
|| zero, unit, mult, add, subtract, divide
```

In the using script, the corresponding "**%include**" statement must give a set of bindings for the free variables of the included script. For example here is an instantiation of the matrix package sketched above, with real numbers as the chosen element type:

```
%include "matrix_pack"
         { element == num; zero = 0; unit = 1
           mult = *; add = +; subtract = -; divide = /
         }
```

The three directives "**%include**", "**%export**" and "**%free**" provide the Miranda programmer with a flexible and type secure mechanism for structuring larger pieces of software from libraries of smaller components.

Separate compilation is administered without user intervention. Each file containing a Miranda script is shadowed by an object code file created by the system and object code files are automatically recreated and relinked if they become out of date with respect to any relevant source. (This behaviour is similar to that achieved by the UNIX program "**make**", except that here the user is not required to write a makefile – the necessary dependency information is inferred from the "**%include**" directives in the Miranda source.)

Current implementation status

An implementation of Miranda is available for a range of UNIX machines including SUN-4/Sparc, DEC Alpha, MIPS, Apollo, Sequent Symmetry, Sequent Balance, Silicon Graphics, IBM RS/6000, HP9000, PC/Linux. This is an interpretive implementation which works by compiling Miranda scripts to an intermediate code based on combinators. It is currently running at 550 sites (as of August 1996).

Licensing information can be obtained from the world wide web at
`http://miranda.org.uk`

REFERENCES

Milner, R. "A Theory of Type Polymorphism in Programming" Journal of Computer and System Sciences, vol 17, 1978.

Thompson, S.J. "Laws in Miranda" Proceedings 4th ACM International Conference on LISP and Functional Programming, Boston Mass, August 1986.

Turner, D.A. "Miranda: A non-strict functional language with polymorphic types" Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy France, September 1985 (Springer Lecture Notes in Computer Science, vol 201).

*[**Note** – this Overview of Miranda first appeared in SIGPLAN Notices, December 1986. It has here been revised slightly to bring it up to date.]*