From Springer Lecture Notes in Computer Science 201:1-16.

Miranda: A non-strict functional language with polymorphic types

D.A.Turner
Computing Laboratory
University of Kent
Canterbury, England

September 1985

Introduction

The last few years have seen much fruitful research into the nature of functional programming. Although there are still many important questions unanswered it seems clear that we have reached a point at which it is appropriate to make available what we have found out so far to a larger community, in the form of stable implementations of complete and well-documented programming languages.

Miranda[1] is a functional programming language which has been developed with this aim in mind. Given that there are still honest disagreements about some quite fundamental questions amongst researchers in the field, we cannot have a single vehicle. Miranda embodies one set of design decisions.

The basic ideas of Miranda are closely modelled on those of the earlier languages SASL [Turner 76, Richards 84] and KRC [Turner 82]. To arrive at a system more suitable for tackling large problems Miranda adds to this foundation (i) a polymorphic type system [Milner 78] and (ii) a library structure with type secure facilities for separate compilation and linking. The major part of this paper will be taken up with a discussion of the type system and in particular the facilities for user-defined types in Miranda, with other aspects of the language and its programming environment being only briefly sketched.

In general approach Miranda is quite similar to HOPE [Burstall McQueen & Sannella 1980]. The fundamental difference is that following SASL and KRC, Miranda has a **non-strict** (i.e. "lazy") semantics. There are two reasons for this decision. The first is the authors belief that a non-strict semantics is the only one fully consistent with the principle of referential transparency [this is argued in Turner 81]. The second and more practical reason is that the presence of non-strict functions and infinite data structures seems to yield a richer and more expressive language.

[1] 'Miranda' is a trademark of Research Software Ltd.

Miranda also has a somewhat terser style than HOPE — it uses fewer keywords, and leaves rather more to be deduced by the compiler (the main example of this being that the type system is based on inference rather than declaration). Whether you like this or not is very much a matter of personal taste.

Overview of the Language

Miranda is a purely functional language, with no imperative features of any kind. A program is a collection of definitions, of functions and other data objects, written in the form of recursion equations. The order in which the definitions are written is not in general significant — the definition of an object may come later in the program than its first use for example. It seems inappropriate to call a collection of equations a 'program', so we call it a 'script'. There is a nested block structure using **where**, and indentation of inner blocks is compulsory — as in SASL the compiler uses the *offside rule* to determine the scopes of local definitions.

```
foldr op z = g
             where
             g [] = z
             g (a : x) = op a (g x)

product = foldr (*) 1
sum = foldr (+) 0
and = foldr (&) True
or = foldr (\/) False

example:-  product[1,2,3,4] = 24

        FIGURE 1
(Using a higher order function)

Note that '[]' is the empty list, and that ':' is infix cons  on
lists.   Note  also  that in Miranda an operator, such as '*' or
'+' can be passed as a parameter by enclosing it in parentheses.
The  'foldr'  function  defined  above is closely related to the
reduce operator of Backus.
```

A typical script is shown in Figure 1, which defines a higher order function `foldr` and uses it to define a series of useful list-processing functions. Like KRC, Miranda uses guarded equations rather than conditional expressions to express case analysis. So the greatest common divisor function can be expressed:-

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

Case analysis can also be done by 'pattern matching' on the argument. The reader will see an example of this in the definition of `foldr` shown in Fig 1, where the auxiliary function `g` is defined by pattern matching.

Miranda also takes over from KRC the '`..`' notation for arithmetic series, so e.g. `[1..10]` is shorthand for the list `[1,2,3,4,5,6,7,8,9,10]`. In fact there is a fairly rich syntax for expressing sequences of various kinds in a succinct way. Some examples are given in Figure 2.

```
nats = [1..]

evens = [2,4..]

negs = [-1,-2..]

fibs = [a | (a,b) <- (1,1), (b,a+b)..]

primes = sieve [2..]
         where
         sieve(p : x) = p : sieve[n<-x | n rem p ~= 0]
```

FIGURE 2
(Some Miranda notation for infinite sequences)

Notations for both list-abstraction and set-abstraction are provided, and distinguished by using square brackets `[...]` for lists, and braces `{...}` for sets. As an example of the conciseness of set expressions we offer the following Miranda expression, which searches for counter-examples to Fermatś Last Conjecture:-

```
{ [a,b,c,n] | a,b,c,n <- [3..]; a^n + b^n = c^n }
```

The Miranda compiler is embedded in an interactive system (implemented under UNIX[2] providing access to a screen editor, an on line reference manual and an interface to UNIX. There is support for a program library structure with fully automatic and type secure facilities for separate compilation of library components. (Users do not need to be aware that object files exist, since they are kept up to date by the system.)

The library structure is quite powerful and has some similarities to [McQueen 84] but within the framework of a somewhat simpler mechanism. A full description of the language and its programming environment is in preparation.

The type system

In Miranda it is not required to declare the types of functions and other objects. The language is, however, strongly typed. The types are deduced by the

[2]UNIX is a trademark of Bell Laboratories.

compiler from the information in the script. This is done using a method based on [Milner 78].

Miranda notation for types (which is very close to that of the original version of ML [Gordon et al 1979]) is shown in Figure 3. Note that there is a single type 'num', which comprises both integers and floating point numbers, the distinction between these being handled at run time. (This was done to avoid the necessity of having two sets of arithmetic operators, given that it would greatly complicate the algorithm for inferring types if it had to cope with overloaded operators.)

The primitive types, in addition to `num`, are `bool` (comprising `True` and `False`) and `char`, which is the ascii character set, written using the same conventions as in "C".

Notice that a strongly typed language has to distinguish between lists, such as `[1,2,3,4]`, whose elements must all be of the same type, and tuples such as `(1,True,'@')`, whose elements can be of mixed type. List processing operations work only on lists — with a tuple, you can do little, except pass it around and extract its elements by pattern matching. Tuples correspond to what are usually called 'records' in an imperative language.

Types can in general be polymorphic, so for example if we define the identity function:-

```
I x = x
```

the type deduced for this will be (we use '`::`' to mean 'is of type')

```
I :: *->*
```

where the '`*`' stands for an arbitrary or unknown type.

It is permitted to specify the types of functions and other objects in the script, and if this is done the compiler will check that the types are correctly given. Very occasionally it is necessary to give the system type information it could not deduce for itself (a situation of this kind is discussed in [Mycroft 84]).

A certain level of type information in the script is desirable for documentation purposes, even though the compiler does not need it. One of the options available is to have the compiler insert the types for you, after it has deduced them.

An interesting consequence of letting the compiler discover the type of your function, is that it may tell you that it has a more general type than you realised. For example the type deduced by the system for 'foldr' (see Figure 1) is

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

which reveals that the operator supplied to `foldr` can be unsymmetrical, which is something that most people miss when they are shown its definition.

The type system in Miranda serves two purposes. First it permits a significant proportion of programming errors to be detected at compile time. Secondly it provides a foundation on which to erect a system of user-defined types. It is these that are the main subject of this paper.

primitive types

```
        num     bool    char
```

function types

```
        T1 -> T2

examples:- sqrt :: num->num
              + :: num->num->num
```

list types

```
        [T]

examples:- [char]       ||strings
           [[num]]      ||matrices
           [num->num]   ||a list of functions like ‘sqrt’
```

tuple types

```
        (T1, ... ,Tn)

example:- (1,True,'@') :: (num,bool,char)
```

generic type variables

```
        *   **   ***   ...   etc.

examples:- map :: (*->**)->[*]->[**]
           reverse :: [*]->[*]
```

FIGURE 3
(Miranda notation for types)

Note that ‘->’ is right associative, and functions of >1 argument are normally regarded as being ‘curried’, e.g. see the type of ‘+’ above. Note also that types can be enclosed in parentheses for grouping. See for example the type of ‘map’ above. There is no conflict between this convention and the use of parentheses in tuple types, because Miranda has no concept of a 1-tuple.

User defined types

We now discuss the facilities available in Miranda for user-defined types. There are three mechanisms — (i) type synonyms, (ii) algebraic types, (iii) abstract types.

Type synonyms

This permits the user to introduce a name for an already existing type (we use '`==`' for these definitions, to distinguish them from ordinary value definitions) thus

```
string == [char]
```

makes `string` a synonym for the type `[char]`. We can also introduce names for operators over types, this is done by using generic type variables as formal parameters, as in

```
f2 * == *->*->*
```

So now `f2 string` will mean `[char]->[char]->[char]`, for example. Obviously type operators of any arity can be defined in this style by introducing enough generic type variables into the definition.

[**Note** In Miranda the '`==`' symbol is always used when we have to bind an identifier to a type. This turns up in two other places — when implementing an abstract type (see below) and also when passing a type into a parameterised script via an **include** directive (this is part of the library mechanism, not discussed here). A type binding not associated with one of these special contexts is treated as introducing a synonym — these are entirely transparent to the typechecker.]

The ability to introduce abbreviations for types is likely to be less important in Miranda than it might be in some other languages, because the type system is primarily based on inference rather than explicit declaration. The type synonym facility was included, however, because it seemed that the language would be in some sense incomplete without it. Being able to rename types is also sometimes useful for documentation purposes.

Algebraic data types

The basic method of introducing a new concrete data type, as in a number of other languages, is to declare a free algebra. In Miranda this is done by an equation using the symbol '`::=`',

```
tree ::= Niltree | Node num tree tree
```

being a typical example. This introduces three new identifiers, '`tree`' which is a new type, '`Niltree`' which is a tree-constructor of no arguments (i.e. an atomic

tree), and '`Node`' which is a tree-constructor of type: `num->tree->tree->tree`. The idea of using free algebras to define data types has a long and respectable history [Landin 64], [Burstall 69], [Hoare 75]. We call it a free algebra, because there are no associated laws, such as a law equating a tree with its mirror image. Two trees are equal only if they are constructed in exactly the same way.

Note that we do not need names for the recognisers and selectors corresponding to each constructor (such as 'isnode', 'label', 'left', 'right' for nodes) because we can do pattern matching on the constructors. For example we can define a function of type (`tree->num`) for summing all the labels in tree, thus

```
sumlabs Niltree = 0
sumlabs (Node a x y) = a + sumlabs x + sumlabs y
```

Since an '`::=`' can introduce a type with any number (≥ 1) of constructors, each of which has any arity (≥ 0), algebraic types are a very general idea. Among the possibilities subsumed, are enumerated scalar types (as in PASCAL)

```
day ::= Mon | Tue | Wed | Thur | Fri
```

and also unions

```
boolint ::= Left bool | Right int
```

This is a labelled disjoint union. The other possible kind of union (coalesced union, as in Algol 68), is not present in Miranda, and (at least in the author's view) is not missed.

The subject of an algebraic type definition can be made a family of types, rather than just a single type, by introducing an appropriate number of generic type variables as formal parameters of the definition. So we can revise our earlier definition of trees to allow trees with labels of arbitrary type, built with polymorphic constructors, by writing

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

This introduces an infinite family of types, of which '`tree int`', '`tree bool`', '`tree(num->num)`', are typical members. This definition of trees is exploited in Figure 4 to define a polymorphic version of the well known tree-sort algorithm. (Note the use of '`foldr`', defined earlier.)

It is interesting to note that algebraic type definitions could be used to define all the basic data types of the language, e.g.

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

gives us respectively (natural) numbers, booleans and lists with arbitrary elements. That we make numbers and certain other data types primitive to Miranda is therefore from considerations of convenience and efficiency, not logical

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

sort :: [*]->[*]; build :: [*]->tree *; flatten :: tree *->[*];
|| these types would be deduced by the compiler anyway, but
|| we include them as documentation

sort = flatten.build

build = foldr insert Niltree
        where
        insert b Niltree = Node b Niltree Niltree
        insert b (Node a s t) = Node a (insert b s) t, b<=a
                              = Node a s (insert b t), b>a

flatten Niltree = []
flatten(Node a s t) = flatten s ++ [a] ++ flatten t
```

FIGURE 4
(The tree sort algorithm in Miranda)

Note that in Miranda the comparison functions, `> >= = ~= <= <`, are polymorphic and impose a built in ordering on every (finite) type. On numbers and characters this is the natural ordering, and on other data types it is defined lexicographically (in the case of [char] for example it gives us the usual dictionary order). The tree sort algorithm given above is therefore polymorphic, and will sort lists with any finite element type. The infix operator '++' denotes the append operation on lists. The infix operator '.' is functional composition.

necessity. The primitive data types of the language may be regarded as having been introduced by '`::=`' definitions, but with some special syntax that allows us to write '`3`' instead of `Suc(Suc(Suc Zero))`, '`[]`' instead of `Nil`, and so on.

The apparatus briefly rehearsed above (many-sorted free algebras, plus polymorphism) is in no way peculiar to Miranda. Exactly the same apparatus (up to trivial differences in syntax) is to be found both in the functional language HOPE and in the procedural language ML.

In Miranda, however, the power of the apparatus is increased because the constructor functions are (in default of information to the contrary) taken to be non-strict. As a result, recursively defined types, such as '`tree`', here include infinite objects. As an object of type '`tree num`' we have

```
big = Node 1 big big
```

the (complete) infinite binary tree, with `1` as the label at every node.

A more interesting example is that from the BNF of a context free language we can easily write some equations to construct a tree containing all possible sentences of the language (which we can then search for sentences with a given property). The presence of infinite data structures has a far reaching effect on programming style, the ramifications of which we have hardly begun to explore. (Consider for example the possibility of representing real numbers as infinite Cauchy sequences.) The author is convinced that the possibilities opened up by the use of infinite data structures form an essential part of functional programming.

Infinite objects are not always required in a recursively defined data type. If we are setting up an account of natural numbers from first principles, the infinite object defined by '`x = Suc x`' is an unwelcome intrusion into the theory. Miranda provides a means to control this, by writing '`!`' after a field in an algebraic type definition, to indicate that the constructor is strict in that field. For example ordinary LISP type lists (which must be finite) may be introduced by the definition

```
lisplist * ::= Nil | Lispcons *! (lisplist *)!
```

which makes '`Lispcons`' strict in both its arguments. Notice that either of the strictifying symbols may be present or absent independently of the other, giving us four possible accounts of the list data type, which comprise in addition to ordinary finite lists and fully lazy lists, two interesting intermediate forms of partial laziness.

[**Note** Readers familiar with domain theory will have recognised that there is a simple isomorphism between a '`::=`' definition and the corresponding domain equation, and that '`!`' is an operator which cuts the bottom element from a domain before including it in the construction. This gives the Miranda programmer fine control over the structure of his data types. It is intended that future implementations of Miranda will exploit this extra information to construct more efficient representations of data objects.]

The naive user, who does not use '`!`' in his type definitions, gets the failsafe option in which all the infinite objects are available if he wants to use them.

Unfree algebras

Miranda also offers a facility for defining algebraic data types with associated laws. Laws are special equations using '`=>`' which can accompany an algebraic type definition. This greatly extends the scope of algebraic types. We take an example — suppose we wish to define from first principles a type containing the integers (positive, negative and zero). We could write

```
int ::= Zero | Suc int | Pred int
```

As a free algebra this is wrong. We have e.g. '`Zero`', '`Suc (Pred Zero)`' and '`Pred(Suc  Zero)`' as three different objects whereas we really want them all to be the same. We have too many objects in the type. We can fix this by writing after the type definition, the following laws

```
Suc (Pred n) => n
Pred (Suc n) => n
```

These laws are simplification rules which will be used (if applicable) whenever an object of type 'int' is created. The effect of the laws is that certain objects which would otherwise have been distinct are reduced to a common form. The object '`Suc(Pred Zero)`' is no longer part of the data type, because it will be simplified to '`Zero`' as soon as we try to form it.

[**Note** Readers acquainted with domain theory will again recognise a familiar idea at work here, namely that we can form one type from another by means of a *retract*. Here the retract is built into the definition of the type in such a way that it is the retracted type, rather than the original one, which comes into being.]

All the standard textbook examples of data types with constraints (balanced trees, ordered lists etc) can be represented in Miranda as algebraic data types with appropriate laws. For example we can define ordered lists as follows

```
ordlist ::= Onil | Ocons num ordlist
Ocons a (Ocons b x) => Ocons b (Ocons a x), a>b
```

Note the use of the guard '`a>b`' in the statement of the law. Objects of type '`ordlist`' automatically retain themselves in ascending order as they are built. (The effect of the law is that each element ripples down to the right place in the list as it is added.) To see how this works, it is necessary to understand that the law is invoked whenever '`Ocons`' is applied, including therefore on the rhs of the law itself.

[The exact effect of the mechanism can be explained as follows. Let us introduce a different identifier, '`OCONS`' say, to refer to the constructor of the original free algebra. Then we define a function '`ocons`' as follows

```
ocons a (OCONS b x) = ocons b(ocons a x), a>b
ocons a x = OCONS a x    ||otherwise
```

```
rational ::= Ratio num num
Ratio a b => error "bad ratio", b=0 \/ ~integer a \/ ~integer b
          => Ratio (-a) (-b), b<0
          => Ratio (a div h) (b div h), h>1
             where
             h = hcf (abs a) (abs b)
             hcf a b = hcf (a-b) b, a>b
                     = hcf a (b-a), a<b
                     = a,           a=b

Ratio a b $rat_plus Ratio c d = Ratio (a*d+c*b) (b*d)
Ratio a b $rat_minus Ratio c d = Ratio (a*d-c*b) (b*d)
Ratio a b $rat_mult Ratio c d = Ratio (a*c) (b*d)
Ratio a b $rat_div Ratio c d = Ratio (a*d) (b*c)
rat_recip (Ratio a b) = Ratio b a
whole (Ratio a b) = (b=1)
|| etc...
```

FIGURE 5
(Rational numbers: an algebraic data type with laws)

Note the use of the function ‘error’ in the first law, to eliminate malformed rationals. The result of applying ‘error’ to a string is an object which possesses every type (i.e. is of type ‘*’ and is semantically indistinguishable from bottom (i.e. non-termination). Note also that there is in Miranda a class of user defined infix operators, of the form ‘$identifier’. Any identifier can be turned into an infix, at any time, simply by prefixing it with the ‘$’ sign.

Throughout the rest of the program, whenever ‘`Ocons`’ occurs on the left, in a pattern match, we read it as the constructor ‘`OCONS`’, but if it occurs on the right it is a reference to the function ‘`ocons`’ defined above.]

A definition in Miranda of a more complex algebraic type, rational numbers, is shown in Figure 5. The effect of the laws is to ensure that a rational is always represented in its lowest terms.

Abstract data types

Many of the data types which in other languages would have to be expressed as abstract data types can be represented in Miranda as algebraic data types with associated laws. Nevertheless there is still a need for abstract data types, as may be seen from the following example (which is based on a use of abstract data types in the LCF system [Gordon et al 79]).

Suppose we are interested in writing programs to derive theorems in a formal system of inference. Such a system would typically be organised as follows. There is a class of *wffs* (well formed formulae), which are correctly formed propositions of the theory. These can be defined by giving a grammar, say. *Theorems* are a distinguished subset of wffs, which are generated inductively from axioms by using rules of inference. For example in the standard formulation of propositional logic, there is an axiom which says that for any wffs A B, it is a theorem that: A implies (B implies A). There are two more axioms, and a single rule of inference (modus ponens) which enables us to derive new theorems from existing ones.

We would like to use the type system to guarantee that a well typed program cannot, even accidentally, make an invalid inference. One way to do this would be to define proof as an algebraic data type (a proof is a kind of tree with instances of the axioms as its leaves) and we could easily do this in such a way that only valid proofs are permitted by the type definitions. Proofs are rather large objects, however, so let us suppose we decide not to do things this way. Instead of collecting proofs we decide to collect theorems. Now we need an abstract data type.

`Theorem` is an abstract data type based on `wff`. A theorem looks like a wff, but has been lifted to a higher world (think of it as being dyed blue). The entrances to this higher world are closely guarded (as in general are the exits, although that is not relevant in this example.) The only way create a blue object is either by using an axiom, or by applying a rule of inference to objects that are already blue.

This is illustrated in Figure 6, which is a complete implementation of the notion of theorem-of-propositional-logic as an abstract data type in Miranda. First we define *wff* as an algebraic data type. Then we declare that *theorem* is an abstract data type, and give a list of the identifiers which implement it together with their types (this list is called the signature of the abstract data type).

Notice that the information that `theorem` is based on `wff` is not given in the **abstype** declaration, nor any other information about how the objects mentioned in the signature are implemented. (Methodologically, this is surely right — in order to take delivery of an abstract object, it is not necessary to know anything about its internal representation.)

The implementation equations (the fact that a `theorem` is really a `wff`, the definitions of `axiom1`, `axiom2`, and so on) are given separately. This does not have to be immediately after the **abstype** declaration, it could be somewhere else in the same script. (Nor do the bindings for `theorem`, `axiom1`, `axiom2`, etc have to be given together — they could be scattered about the script, although stylistically that would be bad practice.)

Although the idea of an abstract data type is now standard, the reader will see from the example that the way in which they are presented in Miranda (compared with say ML or HOPE) involves some innovations. The first, and minor deviation is that we have separated the declaration of the signature of an abstract data type from the statement of how it is implemented, treating these

```
wff ::= Var[char] | wff $Implies wff | Not wff

abstype theorem
with axiom1, axiom3 :: wff->wff->theorem
     axiom2 :: wff->wff->wff->theorem
     modus_ponens :: theorem->theorem->theorem
     contents :: theorem->wff

theorem == wff
axiom1 a b = a $Implies (b $Implies a)
axiom2 a b c = (a $Implies (b $Implies c))
                $Implies ((a $Implies b) $Implies (a $Implies c))
axiom3 a b = Not (a $Implies b) $Implies (Not b $Implies Not a)
modus_ponens a (a $Implies b) = b
contents x = x
```

FIGURE 6
(Data type abstraction in Miranda - creating 'theorem' from 'wff')

as distinct syntactic acts.

More significant is that the abstract data type mechanism used in Miranda does not require the division of the program into two regions (the inside and outside of a capsule, say) such that in one region the the programmer has access to conversion functions (called '`abs_theorem`' and '`rep_theorem`', say) permitting him to move at will between the abstract type and its representation, while in the other region these are hidden from him. The mechanism used in Miranda is **transparent**, in that all the identifiers involved are visible throughout the script. The security of the abstract data type here depends, not on the hiding of declared identifiers, but on the fact that explicit acts of conversion between the abstract type and its representation are **nowhere** permitted.

The idea behind the mechanism is that all the information necessary to carry out the abstraction can be deduced from the signature, together with the concrete binding given for the abstract type. By substituting the latter into the former we can infer a second signature. Call these the abstract and the concrete signature, respectively. By comparing them systematically the compiler can infer what type conversions are necessary to support the abstraction, and where in the text they must be inserted. (The actual implementation technique is less clumsy than this, but has the same effect. See *Further remarks* below)

The advantages (in terms of security and convenience) of having the conversion functions installed by the compiler rather than the user, should be clear. It might be argued however, that this is pushing just too much onto the compiler and will lead to difficulties (perhaps that users will not understand the implications of what they are doing). The method of defining abstract data types must be regarded as one of the more experimental features of Miranda, and only after a period of experience with the language will we be in a position to say where the balance of advantage lies.

```
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
```

FIGURE 7
(A polymorphic type abstraction in Miranda)

We here define 'stack' to be an abstract type based on lists. Outside of the implementation equations, any attempt to access a stack using list operators will be rejected as illegal.

A second and more familiar example of an abstract data type is given in Figure 7, where we give an **abstype** declaration for stacks, together with the obvious implementation in terms of lists. Note that in this example the abstract type is a polymorphic one.

It is also permitted following an **abstype** to introduce several abstract data types — in this case, after the **with** we must give a collective signature for all of them. It is necessary to be able to do this because in general, abstract data types occur in groups (think of e.g. 'graph', 'node', 'edge').

A nice fact about data type abstraction in Miranda is that it involves no run-time penalty. The responsibility for enforcing distinctions such as that between '`theorem`' and '`wff`' can be discharged entirely at compile time. At run time a `theorem` is just another `wff`, and is represented in the same way. So we can build layer upon layer of abstract data types in our programming, without thereby incurring any loss of efficiency.

Note that the mechanism for data type abstraction which is presented here is inextricably bound up with strong (i.e. compile time) typing. There would seem to be no equivalent mechanism available in a language which delays its type checking until run time. By contrast, the traditional account of data type abstraction as an act of encapsulation would appear to be equally applicable to both strongly and weakly typed languages.

By 'encapsulation' (as a method of defining an abstract data type) we mean the process of defining within a local scope, some functions or other objects which involve a locally defined type, and then exporting them to a place where the type, or some of the facts about the type, are unknown. In Miranda to create an abstract data type in this way is always an illegal act. (This is a

consequence of the scope rules for type identifiers — the rules are quite simple, but we will not go into them here.)

If you want to define an abstract data type, you must use the **abstype** mechanism.

Further remarks on type abstraction

Since the presentation of data type abstraction in Miranda is somewhat non-standard we offer some additional explanatory remarks.

How to typecheck a script containing an **abstype** declaration (sketch):- First we use the binding of the abstract type to its representation type to compute the concrete signature from the abstract signature. The binding of the abstract type to the representation type is then suppressed — from now on 'theorem' and 'wff' (or 'stack' and 'list', or whatever) are treated as two distinct and unrelated types throughout the script. Each identifier in the signature now has two types, a concrete type and an abstract type. When typechecking the implementation equations each such identifier is regarded as having been declared with its concrete type; when typechecking the rest of the script (i.e. outside the implementation equations) it is regarded as having been declared with its abstract type. All other identifiers in the script (i.e. those not listed in the signature) are treated as having the same type everywhere. (end of sketch)

It is interesting to note that if you take the complete Miranda script containing an abstract date type declaration like that of figure 6 and remove from it just the '**abstype** ... **with** `<signature>`' part, leaving everything else intact, including the implementation equations, the resulting script is still well-typed and describes exactly the same computations as before, but now has a coarser type structure — '`theorem`' has collapsed back into '`wff`'.

This observation seems to throw light on the real purpose of introducing type abstractions into our program. It is to provide the compiler with more information about what we are doing, so that it can impose a finer type structure on the program. (So we see that here, data type abstraction should not be thought of as a matter of hiding information — quite the reverse.)

A good question is whether it is always the case that a well typed Miranda script remains well typed when you remove all **abstype** statements from it. The answer is no, because of the existence of mutually recursive acts of data type abstraction, such as when two abstract data types are declared, each having an implementation based on the other. (Although this sounds pathological, it is possible to produce useful examples of this phenomenon, and they are handled correctly by the compiler.) In such a case, when the **abstype** information is removed, so that the abstract types and their representations are collapsed back together, the resulting script will contain an infinite cycle in its type structure, which will be rejected by the compiler. (It was a deliberate decision in the design of Miranda, following the example of ML, that the creation of cycles in the type structure is not permitted except via an **abstype** or '::=' declaration.)

A final note — there is a systematic way of translating an ML-style **abstype** declaration, using encapsulation, into an equivalent Miranda script, using the

transparent style of **abstype** declaration (and vice versa). [We leave this as an exercise for the reader!]

Acknowledgements

Thanks are due to Simon Thompson, of the University of Kent, who drew the author's attention to a serious difficulty with an earlier version of the **abstype** mechanism described here (which would have rendered it unimplementable as a practical mechanism), and also to Mark Scheevel of Burroughs Corporation, for revealing to the author the true nature of 'foldr'

Note

This paper appeared in the Proceedings of IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France, September 1985 (Springer Lecture Notes in Computer Science, vol 201:1-16).

REFERENCES

R.Burstall "Proving properties of programs by structural induction" Computer Journal, vol 12, 1969.

R.Burstall D.McQueen & D.Sannella "HOPE: An experimental applicative language" Proceedings 1st International LISP conference, Stanford 1980.

M.Gordon, R.Milner & C.Wadsworth "Edinburgh LCF" Springer Lecture Notes in Computer Science vol 78, 1979.

C.A.R.Hoare "Recursive data structures" International Journal of Computer and Information Sciences, June 1975.

P.J.Landin "The Mechanical Evaluation of Expressions" Computer Journal, January 1964.

D.McQueen "Modules for Standard ML" Proceedings 3rd International Conference on LISP and functional programming, Austin Texas, August 1984.

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

A.Mycroft "Polymorphic type schemes and recursive definitions" Proceedings 6th International Conference on Programming, Toulouse April 1984. (Springer Verlag LNCS vol 167).

H.Richards "An overview of ARC SASL" SIGPLAN Notices October 1984.

D.A.Turner "SASL Language Manual" St Andrews University Technical Report, December 1976.

D.A.Turner "Aspects of the implementation of programming languages" D. Phil Thesis, Oxford 1981.

D.A.Turner "Recursion equations as a programming language" in Functional Programming and Its Applications, ed Darlington et al., CUP 1982.