Haskell for Miranda Programmers

Kevin Glynn
Bernard Pope
({keving,bjpop}@cs.mu.oz.au)

Technical Report TR1999/14

Department of Computer Science
and Software Engineering,
The University of Melbourne,
Parkville, Vic. 3052, Australia

June 22, 1999

1 Introduction

This document is designed to help programmers with a knowledge of Miranda[1] move to Haskell as quickly and as painlessly as possible. It aims for conciseness over completeness. Pointers to further information about Haskell can be found in section 11. In particular, you are encouraged to read *A Gentle Introduction to Haskell*, which covers the complete language and includes many examples.

According to the Haskell Language Report, the Haskell project started in 1987, at a meeting held at FPCA 87. It was believed that the advancement of functional programming was being stifled by the wide variety of languages available. There were more than a dozen lazy, purely functional languages and none had widespread support (except maybe Miranda, which was already old, a commercial product and therefore considered unsuitable for the purposes of research and teaching).

A committee was formed to design the language. The name *Haskell* was chosen in honour of the mathematician *Haskell Curry*, whose research forms part of the theoretical basis upon which functional languages are implemented. Haskell is a modern, consistent language with few rough edges. It is widely used and widely implemented within the functional programming community. It has largely superseded Miranda for teaching in universities, not least due to the robust, freely available implementations.

Haskell has provoked much research and discussion into the desirable properties of a modern functional language and it has undergone a number of revisions. This document describes version 1.4 of the language.

Version 1.4 has recently undergone some minor revisions aimed at making it more suitable for teaching. The resulting language is known as Haskell 98 and the first conformant implementations are now available. All major implementations of Haskell have committed to supporting Haskell 98 for the forseeable future, therefore it can be taught with confidence and it is hoped that this will encourage more teaching materials and text books to be developed.

The rest of this document describes differences, enhancements and omissions of Haskell 1.4 when compared with Miranda.

Appendix A gives the Haskell equivalent operators and functions for the functions in the standard Miranda environment.

2 All You Need to Know

In this section we aim to give a tutorial introduction to all the information Miranda programmers should need to get started programming in Haskell. Later sections cover important Haskell extensions in more detail.

Just like Miranda, Haskell is a lazy, functional language with polymorphic higher-order functions, algebraic data types and list comprehensions. Also like Miranda, it is layout-sensitive: it uses the so-called off-side rule to delimit definitions. There is a widely used, high quality interpreter available on many platforms (Hugs) and many compilers.

Unlike Miranda, Haskell supports overloading of function names (ad-hoc polymorphism via type classes) and has an extensive module system. Haskell is *pure*, even for IO. Haskell comes with a large number of modules including support for arrays, complex numbers, infinite precision integers, operating system interaction and concurrency. In addition, the syntax of Haskell offers a number of conveniences not found in Miranda (such as anonymous functions, let expressions, if-then-else expressions, case expressions, as patterns, user-defined operators, 'wild-card' parameters, etc.)

Haskell's standard prelude (equivalent to Miranda's standard environment) defines a large number of functions, types and operators. These are automatically available to Haskell programmers.

We now compare and contrast the two languages by first introducing a number of examples written in both languages.

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

Firstly tree insert in Miranda:

```
<1>  tree_type * ::= Null |
<2>                  Tree (tree_type *) * (tree_type *)
<3>
<4>         || bst_insert new bstree
<5>         || To insert a new item into its correct position in a BST.
<6>         || If the new item duplicates an existing item in the tree,
<7>         ||    insert into left subtree.
<8>
<9>  bst_insert :: * -> tree_type * -> tree_type *
<10> bst_insert new Null = Tree Null new Null          || tree is empty so create singleton tree
<11> bst_insert new (Tree ltree val rtree)
<12>       = Tree (bst_insert new ltree) val rtree,    if new <= val
<13>       = Tree ltree val (bst_insert new rtree),    otherwise
```

and now in Haskell:

```
<31> data TreeType a = Null |
<32>                   Tree (TreeType a) a (TreeType a)
<33>      deriving (Eq, Ord, Read, Show)
<34>
<35> {-      bst_insert new bstree
<36>         To insert a new item into its correct position in a BST.
<37>         If the new item duplicates an existing item in the tree,
<38>             insert into left subtree.
<39> -}
<40> bst_insert :: Ord a => a -> TreeType a -> TreeType a
<41> bst_insert new Null = Tree Null new Null           -- tree is empty so create singleton tree
<42> bst_insert new (Tree ltree val rtree)
<43>       | new <= val = Tree (bst_insert new ltree) val rtree
<44>       | otherwise  = Tree ltree val (bst_insert new rtree)
```

Comments

Haskell has two comment styles: block comments and line comments.

A block comment is introduced by {- and the comment finishes with a matching -} (i.e. block comments may contain block comments and they will behave as desired). See lines 35-39.

Line comments are introduced by --, a line comment finishes at the end of the line. Compare line 10 with line 41.

Type Variables

Haskell uses ordinary variable names for type variables (unlike Miranda which uses *, **, *** etc.). Haskell type variables must start with a lower case alphabetic character. By convention, they are a single character, `a`, `b`, `c` etc. Compare lines 1-2 with lines 31-32, and line 9 with line 40.

User-Defined Algebraic Types

Haskell introduces a user-defined algebraic type definition with the `data` keyword. Haskell type names and data constructors must start with an uppercase alphabetic character. Haskell uses `=` rather than Miranda's `::=` to separate the type name from its definition. Compare lines 1-2 with lines 31-32.

Type Classes

Haskell type classes allow the Haskell programmer to group related types into type classes, polymorphic functions can then be constrained to only work on types which belong to particular type classes. For example, on line 40 `bst_insert` is constrained to only work for types in the `Ord` type class. `Ord` is a built-in Haskell type class, it is the class of Haskell types that support an ordering. Read `bst_insert`'s type signature as:

"for all types `a` such that `a` is in the `Ord` class `bst_insert` takes an `a` and a tree of `a` and returns a tree of `a`".

Similarly, the Haskell type signature for a function to sort the elements in a list would be:

```
sort :: Ord a =>  [a] -> [a]
```

`sort` can sort lists of any type as long as that type is in `Ord`.

A type class has a name and a set of operations. For example, the `Ord` class has the operations `compare`, `<`, `<=`, `>=`, `>`, `max` and `min`. The Haskell programmer can declare a type to be an *instance* of a type class by telling the Haskell implementation which type-specific operation to use for each operation in the class. `bst_insert` can only be used on types in the `Ord` class because on line 43 the guard uses the `<=` operator to decide which sub-tree the new element should be inserted to.

Note that it would be an error to omit the class constraint in the type signature. `bst_insert` does *not* work for all types. If the constraint is omitted Hugs will complain:

```
ERROR  "bst_insert.hs" (line 11): Declared type too general
```

A type may be a member of many type classes.

A type class may be a sub-class of another type class. For example, `Ord` is a sub-class of the `Eq` class (for types with equality). So all types which are an instance of `Ord` are also an instance of `Eq`.

The type constraint in a signature is a list of constraints separated by commas. Each constraint constrains a type variable. A type variable may be constrained many times. For example:

```
bst_conv_toset :: (Eq a, Ord a, Show a, Eq b) => Tree a -> (a->b) -> [b]
```

For an extended example of programming with type classes see appendix B.

Many type classes are provided in the Haskell prelude. Haskell programmers can introduce new type classes but that is beyond the scope of this tutorial.

Derived Type Classes

For the Haskell prelude's built-in type classes (`Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read`) Hugs can automatically generate appropriate instance declarations for user-defined types.

On line 33 the user defined type definition is followed by the `deriving` keyword and a list of type classes. The Haskell report defines the rules for the automatically generated instance code (e.g. constructors are ordered lexicographically by their name).

We recommend that only derivations of `Eq` and `Show` should be done this way. Deriving `Eq` will allow `==` and `/=` to be used on values of that type. Deriving `Show` will allow the `show` operator to convert values of that type to a list of characters (which is required by Hugs to print out values with that type).

Guards

In Haskell, guards immediately follow the arguments and precede the body expression. A guard is introduced by the `|` character. Compare lines 12 and 13 with lines 43 and 44. Note that in Haskell, `otherwise` is just a function that returns `True`.

The next sample function checks a list for adjacent, equal elements.

In Miranda:

```
<1>       || dup list
<2>       || Returns true if two adjacent elements of the list
<3>       || are equal.
<4>
<5> dup :: [*] -> bool
<6> dup [] = False
<7> dup [a] = False
<8> dup (a:a:xs) = True
<9> dup (a:b) = dup b
```

In Haskell:

```
<30> {-       dup list
<31>          Returns true if two adjacent elements of the list
<32>          are equal.
<33> -}
<34> dup :: Eq a => [a] -> Bool
<35> dup [] = False
<36> dup [_] = False
<37> dup (a:t@(b:_)) = if a==b then True
<39>                           else dup t
```

As-patterns

Haskell allows any component of a pattern to be given a name. In line 37 the tail of the input list is given the name `t`. `t` can then be used in the rest of the definition to refer to this part of the list.

The following table gives examples of pattern matching:

Pattern	Value	Matches
xs	[1,2,3]	xs = [1,2,3]
(x:xs)	[1,2,3]	x = 1 xs = [2,3]
(a:t@(b:c))	[1,2,3,4]	a = 1 t = [2,3,4] b = 2 c = [3,4]

Wild Cards

In Haskell, it isn't necessary to give names to parameters or components of patterns that aren't required on the right hand side. The programmer can use an underscore instead, as on line 37.

Note that in the Miranda version we had to give names to `xs` on line 8 and `a` on lines 7 and 9 even though we weren't interested in them. The Haskell version uses underscores instead.

Wild cards and As-Patterns can often be usefully combined, e.g. `s@([_])` matches any list with only one element.

Linear Patterns

In Haskell, parameter names can only appear once in patterns. Thus the implicit equality check in line 8 is not possible in Haskell. Note that multiple underscores *are* allowed, each underscore can match with anything.

Conditional Expressions

Haskell allows conditional expressions: `if` exp_0 `then` exp_1 `else` exp_2.

exp_0 is a boolean expression and exp_1 and exp_2 must have the same type. The result of this expression is either exp_1 or exp_2, depending on the value of exp_0. See lines 37-39.

Haskell also supports `case` expressions, see later.

Note that in Haskell `dup` can only be used on lists of elements for which equality is defined, hence the `Eq` constraint on line 34.

Our next example defines a function to return the word with most vowels from a sentence.
In Miranda:

```
<1>  word == [char]
<2>  sentence == [word]
<3>
<4>          || max_vowels sentence
<5>          || max_vowels returns the word from the sentence with the
<6>          || most vowels
<7>
<8>  max_vowels :: sentence -> (word, num)
<9>  max_vowels []          = ("",0)
<10> max_vowels (word:sent) = (word, num_vowels) , if num_vowels >= best_v
<11>                        = (best_w, best_v)   , otherwise
<12>     where
<13>       (best_w, best_v) = max_vowels sent  || max found in rest of sentence
<14>    || isvowel :: char -> bool
<15>       isvowel c = "aeiou" $member c
<16>       num_vowels = # filter isvowel word
```

In Haskell:

```
<30> type Word = [Char]
<31> type Sentence = [Word]
<32>
<33> {-     max_vowels sentence
<34>        max_vowels returns the word from the sentence with the
<35>        most vowels
<36> -}
<37> max_vowels :: Sentence -> (Word, Int)
<38> max_vowels []     = ("",0)
<39> max_vowels (word:sent) = let num_vowels = count_vowels word in
<40>                          if num_vowels >= best_v
<41>                          then (word, num_vowels)
<42>                          else prev
<43>    where
<44>      prev@(_, best_v) = max_vowels sent -- max found in rest of sentence
<45>      count_vowels :: Word -> Int
<46>      count_vowels w = length (filter (\c -> c `elem` "aeiou") w)
```

Type Synonyms

In Haskell, type synonyms are introduced with the `type` keyword. Compare lines 1-2 with lines 30-31. Again, Haskell type names must start with a capital letter.

Typed Local Definitions

Local definitions introduced by where clauses can be given types in Haskell. Compare line 45 with line 14 (the Miranda type definition is commented out!)

Let Expressions

Let expressions introduce one or more local definitions. Let introduced objects can be used in the right hand sides of the definitions or in the body (the expression following the `in` keyword).

Unlike a where clause, which is a syntactic entity and can only be attached to function definitions, a let expression can be used wherever an expression can appear.

An example of `let` appears on line 39.

Anonymous Functions

In Miranda, all functions have to be named. In Haskell, a function object can be written using lambda notation. The function definition consists of "\" followed by a list of parameters, a "`->`" and a body expression. The anonymous function (or "lambda expression") on line 46 takes a character as argument and returns true if it is an element of the string "aeiou".

Functions as Operators

To use a two-argument function as an operator in Haskell surround it with backquote characters, compare the use of `elem` on line 46 with the use of member on line 15.

Again, note the use of as-patterns and wild cards on line 44. `prev` refers to the whole tuple result of the recursive call. The Haskell version of the function can return `prev` if the current word has less vowels than our current best. In the Miranda version a new tuple is constructed with the same contents. Compare line 42 with line 11.

Now, summing the values in a list of abstract expressions.
In Miranda:

```
<1>  expr ::= Num num
<2>           | Plus num num
<3>           | Minus num num
<4>           | Times num num
<5>
<6>         || sum_list expr_list
<7>         || Evaluate each expression in the list and return the total.
<8>
<9>  sum_list :: [expr] -> num
<10> sum_list []
<11>     = 0
<12> sum_list (x:xs)
<13>     = xval x + sum_list xs
<14>       where
<15>         xval (Num v)       = v
<16>         xval (Plus v1 v2)  = v1 + v2
<17>         xval (Minus v1 v2) = v1 - v2
<18>         xval (Times v1 v2) = v1 * v2
```

In Haskell:

```
<30> data Expr = Num Int
<31>           | Plus Int Int
<32>           | Minus Int Int
<33>           | Times Int Int
<34>
<35>
<36> {-     sum_list expr_list
<37>        Evaluate each expression in the list and return the total.
<38> -}
<39> sum_list :: [Expr] -> Int
<40> sum_list []
<41>     = 0
<42> sum_list (x:xs)
<43>     = (case x of
<44>          Num v -> v
<45>          Plus v1 v2 -> v1 + v2
<46>          Minus v1 v2 -> v1 - v2
<47>          Times v1 v2 -> v1 * v2) + sum_list xs
```

Case Expressions

In Haskell, case takes an expression and executes one of a number of alternative expressions, depending on the expression's value. As in this example the case expression can perform pattern matching to select an alternative. In the Miranda version it was necessary to introduce the auxiliary function `xval`.

Finally, list comprehensions differ slightly:
A function to calculate a list of Pythagorean Triples in Miranda:

```
<1>       || Calculates a list of pythagorean triples with sides <= n
<2>       || In a pythagorean triple the sum of the squares of the first two
<3>       || integers equals the square of the third. (cf. right angled triangles)
<4>
<5>   pyTriple :: num -> [(num, num, num)]
<6>   pyTriple n = [ (a,b,c) | a <- [2 .. n]; b <- [a+1 .. n]; c <- [b+1 .. n];
<7>                            c*c = a*a + b*b]
```

and in Haskell:

```
<30>      {- Calculates a list of pythagorean triples with sides <= n
<31>         In a pythagorean triple the sum of the squares of the first two
<32>         integers equals the square of the third. (cf. right angled triangles)
<33>      -}
<34> pyTriple :: Int -> [(Int, Int, Int)]
<35> pyTriple n = [ (a,b,c) | a <- [2 .. n], b <- [a+1 .. n], c <- [b+1 .. n],
<36>                           c*c == a*a + b*b]
```

List Comprehensions

In Haskell, the generators and filters following the vertical bar (|) are seperated by commas, rather than semi-colons.

As in Miranda, the generators are applied left to right, the leftmost changing most slowly. There is no equivalent of Miranda's *diagonalisation* operator which allows this order to be modified.

3 Numbers

In Miranda there is only one numeric type, `num`, which includes both arbitrary precision integers and floating point numbers. Miranda's numeric functions and operators are overloaded so that they can operate on any combination of numeric types.

In Haskell the situation is more complicated. There are a number of built-in numeric types. Type classes are used to overload numeric functions and operators appropriately. The following table from the Haskell Report summarizes the available numeric types:

Type	Class	Description
Integer	Integral	Arbitrary-precision integers
Int	Integral	Fixed-precision integers
Float	RealFloat	Real floating-point, single precision
Double	RealFloat	Real floating-point, double precision

Table 1: Numeric Types

All numeric types derive from class `Num`. This is a sub-class of `Eq` and `Show`, so all numeric types can be compared for equality and displayed.

The `Integral` class is for 'whole-number' types, it is a sub-class of `Num`. There are two instances of `Integral`, `Int`s are fixed precision integers (guaranteed to be at least in the range -2^{29} to $2^{29}-1$) and `Integer`s are arbitrary-precision integers.

The `RealFloat` class is for numbers represented by a machine's floating point representation, i.e. their precision is limited by the underlying representation. There are two instances of `RealFloat`, `Float` uses single precision and `Double` uses double precision. In Hugs the precision of `Double` is the same as the precision of `Float`.

Integer literals (e.g. 1, 3898, etc.) in the program text are automatically converted to the correct numeric type depending on context. So in `3 + 2.73` the `3` will be converted to a `Float` before the addition.

The Haskell report also defines classes and types for Complex Numbers and Rational Numbers (i.e. numbers represented by a ratio of two whole numbers). These are available by importing the appropriate module.

4 Modules

Although not required by Hugs, all Haskell programs should consist of modules. A module consists of a module declaration followed by the module's definitions. The module declaration names the module and (optionally) lists the names of the module's top-level definitions which are to be visible to importing scripts:

```
module Stack(Stack, empty, push, top, pop, isEmpty) where

data Stack a = MkStack [a]

empty :: Stack a
empty = MkStack []

isEmpty :: Stack a -> Bool
isEmpty (MkStack []) = True
```

```
isEmpty _            = False

push :: a -> Stack a -> Stack a
push x (MkStack xs) = MkStack (x:xs)

top :: Stack a -> a
top (MkStack []) = error "top: empty stack"
top (MkStack (x:_)) = x

pop :: Stack a -> Stack a
pop (MkStack []) = error "pop: empty stack"
pop (MkStack (_:xs)) = MkStack xs

count :: Stack a -> Int
count (MkStack xs) = length xs
```

Only the definitions in parenthesis following the module name will be visible to importing scripts. Therefore, importing scripts can only create stacks via `empty` and `push` (the stack constructor, `mkStack` isn't exported[2]) and they can't see the `count` function.

Modules are used to achieve the same effect as Miranda's Abstract Data Types (i.e. `abstype` declarations). As in the above example, put the data type and its operators in a module and only export its name and externally visible operators.

It is possible to omit the names in parenthesis from the module declaration, in this case all top-level definitions in the module are exported.

Modules are imported to scripts by the `import <module name>` declaration. The definitions imported can be specified by:

- Following the module name with a list of definitions to be imported:

  ```
  import Stack (Stack, top, isEmpty)
  ```

- 'Hiding' particular names:

  ```
  import Stack hiding (empty, push, top)
  ```

A module may import other modules. The names thus imported are top-level definitions and may be re-exported as above.

To avoid name clashes (or to make a script clearer) a module may be imported `qualified`. All uses of names from that module must be preceded by the module name and a full stop. For example:

```
import qualified Stack

initStack = Stack.push (-99) Stack.empty
```

Although allowed by the Haskell report, recursive imports are not supported by Hugs, (i.e. a module may not directly, or indirectly, import itself). Also Hugs requires modules to be put in a file with the same name as the module name, i.e. the Stack module must appear in the file Stack.hs.

[2] In Haskell, `newtype` can be used so that the physical allocation of this constructor at run-time can be avoided. See the Haskell Report for details.

5 Enumerations

Any type which is an instance of the `Enum` class has functions which can 'step' from one value to another. This allows Haskell programmers to use *dot dot* notation in list generators. For example: `[1,3..]` is the list of odd integers; the list `['a','c'..'z']` is the list of alternate characters, `"acegikmoqsuwy"`. Programmers can use this notation for their own types by making the appropriate `instance` declarations, see the example in appendix B.

6 Map

In Miranda, `map` is a function over lists. In Haskell, type classes have been used to generalise this to allow `map` to work over any *collection* type, e.g. sets, trees, queues, etc.

Of course, `map` also works on lists, so `map toUpper ['a'..'z']` gives the list of capital letters.

A type class called `Functor` with the operation `map` is pre-defined in the Haskell prelude:

```
class  Functor f  where
    map              :: (a -> b) -> f a -> f b
```

So, we can make `map` work on the binary search trees defined earlier by declaring them to be an instance of Functor:

```
instance TreeType of Functor where
   map _ Null = Null
   map f (Tree l v r) = Tree (map f l) (f v) (map f r)
```

Now the programmer can double all elements of a tree using map:

```
doubleTree :: Num a => TreeType a -> TreeType a
doubleTree in_tree = map (*2) in_tree
```

7 Monads and Lists

`Monad` is a Haskell built-in class, (the term *monad* comes from category theory). Values of a monadic type can be thought of as *computations*. The operators in the monad class allow these computations to be sequenced.

```
class  Monad m  where
    (>>=)            :: m a -> (a -> m b) -> m b
    (>>)             :: m a -> m b -> m b
    return           :: a -> m a

    m >> k           =  m >>= \_ -> k
```

The bind operator, `>>=`, executes a computation, takes its result and feeds it into the next computation. Monads are a very useful advanced programming technique and can significantly simplify programs:

- They allow parameters such as symbol tables to be passed between computations implicitly.
- Since the programmer implements the sequencing operator it can deal with failing computations or exceptions.
- They allow a pure functional programming language to support features that are normally thought of as fundamentally imperative, such as IO (see next section), update-in-place data structures.

In Haskell 1.4, Lists are instances of Monads. (This can be useful because it allows list comprehensions and general list functions (such as fold and filter) to be used on any monadic structure).

Unfortunately, this means that if programmers make errors whilst developing list processing functions they are likely to receive error messages which refer to monadic types, rather than the lists they thought they were using. In addition, if they ask Hugs for the type of a list processing function they may well be confused, for example the type of filter is given as:

```
filter :: MonadZero b => (a -> Bool) -> b a -> b a
```

(`MonadZero` is a sub-class of `Monad` with a `zero` operation). This must be borne in mind when debugging list processing functions.

8 I/O in Haskell

Supporting I/O poses two main difficulties for pure functional languages such as Haskell.

The first difficulty is making functions which perform I/O referentially transparent. Referential transparency requires that for any given argument (or set of arguments) a function must always return the same result (regardless of the context in which it is called). This transparency simplifies reasoning about programs, simplifies re-use of functional code and allows a compiler a lot of freedom to optimize code.

The C function `fgetc()` is an example of a function that is *not* referentially transparent. Recall that `fgetc()` has one argument which represents the stream from which it should read data, and it returns the next character (as an `int`) from that stream or `EOF` indicating that the end–of–file has been encountered. In Haskell terms, the type of `fgetc()` would be `fgetc :: FileHandle -> Int`. The problem from a purely functional point of view is that multiple calls to `fgetc()` with the same argument may return different results. The type that we have assigned to `fgetc()` fails to capture the fact that the function is interacting with and changing the state of the operating system (each subsequent call to `fgetc()` causes (amongst other things) a pointer into an input buffer to be incremented). We call this hidden operation of `fgetc()` a *side–effect*. A mechanism is required which represents the fact that performing I/O introduces a side–effect into the computation, and that the value of a function which performs I/O encapsulates the execution of the side–effect.

The second difficulty is ensuring a sequential ordering on the execution of I/O actions. In an imperative language such as C it is easy to ensure the correct ordering of I/O actions since the order of evaluation is explicit in the syntax of the language. In a declarative language such as Haskell, a programmer does not (in general) specify the order in which the program should evaluate. This is also complicated by the fact that Haskell evaluates a program lazily (or on demand) and so the actual evaluation order of functions is difficult to predict in advance (without expert knowledge of the implementation). For the most part of programming in Haskell we don't care about the order in which the program is evaluated, except when we are performing computations that have side-effects.

Fortunately Haskell has a powerful facility for supporting I/O which preserves referential transparency and allows a sequential ordering to be defined for the execution of I/O actions. This facility is called the I/O Monad.

The ability to perform I/O in a purely functional manner does not come for free, and the use of the I/O Monad does tend to complicate the definition of a program. The designers of the Haskell language have noticed that for many tasks which involve I/O it is easier for the programmer to reason about the execution of I/O in an imperative manner. From this observation came Haskell's *do–notation* which enables the programmer to write code that uses the I/O Monad in a way which resembles the use of statements in imperative languages. The use of do–notation so closely follows an imperative style that it leads many people to believe that it is not functional. Bear in mind that do–notation is only *syntactic sugar* for more complex underlying functional code.

We illustrate the two styles of sequencing monadic computations with the following example (from A Gentle Introduction to Haskell) which copies the contents of one file to another.

Using the raw Monadic operators:

```
import IO

main
  =
      getAndOpenFile "Copy From: " ReadMode >>= \fromHandle ->
      getAndOpenFile "Copy To: " WriteMode  >>= \toHandle ->
      hGetContents fromHandle               >>= \contents ->
      hPutStr toHandle contents             >>
      hClose toHandle                       >>
      putStr "Done.\n"

getAndOpenFile :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode
                 =
                    putStr prompt      >>
                    getLine            >>= \name ->
                    -- catch will trap any exceptions caused by opening the file.
                    -- if we get an exception then ask the user to try again
                    openFile name mode `catch`
                           (\_ -> putStr ("Cannot open " ++ name ++ "\n") >>
                                  getAndOpenFile prompt mode)
```

Using do notation:

```
import IO

main
  = do
      fromHandle <- getAndOpenFile "Copy From: " ReadMode
      toHandle <- getAndOpenFile "Copy To: " WriteMode
      contents <- hGetContents fromHandle
      hPutStr toHandle contents
      hClose toHandle
      putStr "Done.\n"

getAndOpenFile :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode
                 = do
                    putStr prompt
                    name <- getLine
                    -- catch will trap any exceptions caused by opening the file.
                    -- if we get an exception then ask the user to try again
                    openFile name mode `catch`
                           (\_ -> do putStr ("Cannot open " ++ name ++ "\n")
                                     getAndOpenFile prompt mode)
```

Note that, due to Haskell's lazy evaluation, the list of characters returned by `hGetContents` will only be read in as required by `hPutStr`. Thus the whole input file will not need to be read into memory before it is written out.

We have provided a sample Haskell program which makes use of the I/O Monad in Appendix C. This program allows the user to interactively explore nodes in a binary tree, and illustrates the use of do–notation quite extensively.

More information about using the I/O Monad (and monads in general) can be found in *A Gentle Introduction to Haskell*.

9 Monomorphism Restriction

Haskell has a rule that at the top level of a module only function definitions can be overloaded. This is to avoid 'unintuitive' recomputations of constants.

For example the following definition of `lessThan` is overloaded (it can be applied to any arguments whose type is an instance of `Ord`):

```
lessThan = (<)
```

```
ERROR "dmr.hs" (line 7): Unresolved top-level overloading
*** Binding             : lessThan
*** Outstanding context : Ord b
```

There are two simple solutions to this restriction. Always provide an explicit type signature:

```
lessThan :: Ord a => a -> a -> Bool
lessThan = (<)
```

Or provide the arguments, so lessThan is now a function definition:

```
lessThan x y = (<) x y
```

Ideally, do both.

10 Sources

Information in this document has been taken from:

- Report on the Programming Language Haskell, version 1.4, John Peterson and Kevin Hammond (editors).
- Standard Libraries for the Programming Language Haskell, version 1.4, John Peterson and Kevin Hammond (editors).
- Differences between Miranda and Haskell (Notes), Bernard Pope.
- Haskell for Miranda Programmers (Slides), Harald Søndergaard.

11 Further Reading

Most web-based Haskell information can be found by following links from Haskell's home page:

`http:/www.haskell.org/`

In particular the following on-line sources are recommended:

- Report on the Programming Language Haskell, version 1.4, John Peterson and Kevin Hammond (editors).

  `http://www.cs.mu.oz.au/~bjpop/fpu/haskell-report-1.4-html/index.html`

- Standard Libraries for the Programming Language Haskell, version 1.4, John Peterson and Kevin Hammond (editors).

  `http://www.cs.mu.oz.au/~bjpop/fpu/haskell-library-1.4-html/index.html`

- A Gentle Introduction to Haskell, version 1.4, Paul Hudak, John Peterson, Joseph Fasel.

  `http://www.cs.mu.oz.au/~bjpop/fpu/haskell-tutorial-1.4-html/index.html`

The Haskell 98 language definition and library report is often clearer than the 1.4 reports, although care must be taken that the information is still valid in version 1.4:

- Haskell 98: A Non-strict, Purely Functional Language, Simon L. Peyton Jones, John Hughes (editors).

  `http:/www.haskell.org/onlinereport/`

- Standard Libraries for Haskell 98, Simon L. Peyton Jones, John Hughes (editors).

  `http:/www.haskell.org/onlinelibrary/`

Many papers on aspects of Haskell can be found from `http://www.haskell.org/bookshelf/`
The following books on Haskell are currently available:

- Haskell: The Craft of Functional Programming, Simon Thompson, Addison-Wesley, 1996. ISBN 0-201-40357-9.
- Introduction to Functional Programming using Haskell, 2nd edition, Richard Bird, Prentice Hall Press, 1998, ISBN: 0-13-484346-0.

A Common Operators and Built-in Functions

The following tables give the Haskell equivalents (where they exist) of all the Miranda operators and functions defined in standard environment.

A.1 Miranda operators

Miranda	Haskell	Comment
=	==	Class Eq
~=	/=	Class Eq
>=	>=	Class Ord
>	>	Class Ord
<=	<=	Class Ord
<	<	Class Ord
+	+	Class Num
-	-	Class Num
*	*	Class Num
/	/	Class Fractional
div	`div`	Class Integral
mod	`mod`	Class Integral
^	^	exponent
~	not	
&	&&	conjunction
\/	||	disjunction
:	:	list cons
++	++	list append
#	length	
!	!!	list indexing
--	\\	list difference (Module List)
.	.	function composition

A.2 Miranda Functions

Miranda	Haskell	Comment
`abs`	`abs`	Class Num
`and`	`and`	
`arctan`	`atan`	Class Floating
`cjustify`		*No simple equivalent*
`code`	`ord`	module Char
`concat`	`concat`	
`converse`	`flip`	
`cos`	`cos`	Class Floating
`decode`	`chr`	module Char
`digit`	`isDigit`	module Char
`drop`	`drop`	
`dropwhile`	`dropWhile`	Note the capital 'W'
`e`	`exp 1`	
`entier`	`floor`	class RealFrac
`error`	`error`	
`exp`	`exp`	class Floating
`filter`	`filter`	
`foldl`	`foldl`	
`foldl1`	`foldl1`	
`foldr`	`foldr`	
`foldr1`	`foldr1`	
`fst`	`fst`	
`hd`	`head`	
`id`	`id`	
`index`		*No simple equivalent*
`init`	`init`	
`integer`	`integer x = x == fromIntegral (truncate x)`	
`iterate`	`iterate`	
`last`	`last`	
`lay`	`unlines`	
`layn`		*No simple equivalent*
`letter`	`isAlpha`	module Char
`limit`		*No simple equivalent*
`lines`	`lines`	
`ljustify`		*No simple equivalent*
`log`	`log`	class Floating
`log10`	`log10 n = logBase 10 n`	class Floating
`map`	`map`	class Functor
`map2`	`zipWith`	class Functor
`max`	`maximum`	
`max2`	`max`	
`member`	`elem`	Args different order
`merge`		*No simple equivalent*
`min`	`minimum`	
`min2`	`min`	
`mkset`	`nub`	module List

Miranda	Haskell	Comment
`neg`	`negate`	class Num
`numval`	`read`	
`or`	`or`	
`pi`	`pi`	class Floating
`postfix`	`postfix x xs = xs ++ [x]`	
`product`	`product`	
`rep`	`replicate`	
`repeat`	`repeat`	
`reverse`	`reverse`	
`rjustify`		*No simple equivalent*
`scan`	`scanl`	
`shownum`	`show`	
`sin`	`sin`	class Floating
`snd`	`snd`	
`sort`	`sort`	module List
`spaces`	`spaces n = repeat n ' '`	
`sqrt`	`sqrt`	class Floating
`sum`	`sum`	
`take`	`take`	
`takewhile`	`takeWhile`	Note the capital 'W'
`tl`	`tail`	
`undef`	`error ``undefined''`	
`until`	`until`	
`zip`	`miranda_zip (a,b) = zip a b`	
`zip2`	`zip`	
`zip3`	`zip3`	

B Type Classes by Example

In this this section we present the use of Haskell type classes by example. We show how the Haskell system can automatically *derive* some types to be instances of some classes, and we show how types can be manually made instances of a class. We use the standard type class `Enum` throughout this discussion because it is small enough for a full exposition, yet rich enough to be of practical use.

Programmers can define their own type classes, but this is beyond the scope of this tutorial.

B.1 The `Enum` type class

The `Enum` type class is useful for any type whose members can be enumerated. For example the character type (`Char`) is enumerable and is a member of `Enum`, however functions (of type `a -> b`, for example) cannot be enumerated, and therefore cannot be a member of `Enum`.

There are two basic benefits of an enumerable type being a member of `Enum`. Firstly, the class ensures a consistent manner for mapping integers to members of the type and vice versa. Secondly, the class enables the use of *dot dot* notation in list generators. For example, `[1..]` produces the infinite successive list of integers starting at the number `1`.

There are six functions which must be defined for a type if it is to be accepted as an instance of `Enum`. They are:

1. `toEnum :: Int -> a`, maps an integer to a unique member of the enumerable type.
2. `fromEnum :: a -> Int`, maps an element of the enumerable type to a unique integer.
3. `enumFrom :: a -> [a]`, enumerates all members of the enumerable type in successive order starting at a particular member (`[1..]`).
4. `enumFromThen :: a -> a -> [a]`, enumerates some members of the enumerable type, starting at a particular member and stepping by a particular distance between successive members (`[1,3..]`).
5. `enumFromTo :: a -> a -> [a]`, similar to `enumFrom`, except the enumeration is bounded by a particular member (`[1..10]`).
6. `enumFromThenTo :: a -> a -> a -> [a]`, similar to `enumFromThen`, except the enumeration is bounded by a particular member (`[1,3..10]`).

Note that the last two functions (`enumFromTo` and `enumFromThenTo`), are given default definitions in `Enum` as below, and so do not have to be specifically provided by the programmer (although, programmers may provide alternate definitions if they so desire):

```
enumFromTo x y = map toEnum [fromEnum x .. fromEnum y]
enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

Therefore, there are actually only four functions that *must* be introduced to make a type a member of `Enum`.

B.2 A basic enumerated type, and automatic inference

In many cases we may declare a new type whose members are all listed in the definition of the type. In other words, the type forms a finite countable set of values. For example, we may be interested in a type which names the days of the week:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

For such types we can ask the Haskell system to automatically make them an instance of `Enum`. Essentially, this requires the Haskell system to generate the four necessary functions mentioned above. For the type `Day`, this is a trivial matter of mapping each member of the type to an integer consecutively from zero to six.

To ask the Haskell system to derive `Day` to be a member of `Enum`, we simply write:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
           deriving Enum
```

We would also like to be able to display members of the type, and so `Day` must be made an instance of the `Show` type class. This too can be derived by Haskell (at least, for types such as `Day`), and so we extend our definition of `Day` a little further:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
           deriving (Enum, Show)
```

Having made `Day` an instance of `Enum` and `Show` we can do interesting things as in the following:

```
Hugs> fromEnum Wed
3
Hugs> (toEnum 6)::Day
Sat
Hugs> [Sun ..]
[Sun, Mon, Tue, Wed, Thu, Fri, Sat]
Hugs> [Mon, Wed .. ]
[Mon, Wed, Fri]
Hugs> [Mon .. Thu]
[Mon, Tue, Wed, Thu]
```

B.3 A more complex enumerable type

Suppose we would like to model the natural numbers using only the basis value `Zero` and the successor function `S`. From `Zero` and `S` we can recursively enumerate the set of natural numbers. For example the number four is represented by the application `S (S (S (S Zero)))`. We can implement this model in Haskell very succinctly using the type `Nat`, defined below:

```
data Nat = S Nat | Zero
           deriving Show
```

Addition and multiplication on this datatype (which reflect the normal meaning for addition and multiplication on natural numbers) can be defined in the following manner:

```
addNat :: Nat -> Nat -> Nat
addNat m Zero = m
addNat m (S n) = S (addNat m n)

multNat :: Nat -> Nat -> Nat
multNat _ Zero = Zero
multNat m (S n)
   = addNat m (multNat m n)
```

For example, we can perform the following operations:

```
Hugs> addNat (S Zero) (S (S Zero))
S (S (S Zero))
Hugs> multNat (S (S (S Zero))) (S (S Zero))
S (S (S (S (S (S Zero)))))
```

We know that the members of the `Nat` type are (recursively) enumerable and so we would like to make `Nat` an instance of `Enum`. We can't ask the Haskell system to automatically derive this instance because such derivation only works when all members of the type are listed in the definition of the type. Therefore, we have to manually make `Nat` an instance of `Enum` by defining the four required functions mentioned above (`toEnum`, `fromEnum`, `enumFrom`, `enumFromThen`).

First we must give an instance declaration, like so:

```
instance Enum Nat where
   toEnum = intToNat
   fromEnum = natToInt
   enumFrom = natFrom
   enumFromThen = natFromThen
```

Then we must implement the various required functions.

The function `toEnum` requires a mapping from the integers to members of the new type. We implement it in the function `intToNat` below. It is obvious that `0` maps to `Zero` and all positive integers map to a corresponding sequence of `S` applications. Since the natural numbers do not cover negative values, we map negative integers to the natural number that corresponds to the absolute value of the integer. An alternative option would have been to return an error if a negative integer was supplied as an argument, but that complicates the discussion somewhat.

```
intToNat :: Int -> Nat
intToNat n
   | n > 0 = S (intToNat (n - 1))
   | n == 0 = Zero
   | otherwise  = intToNat (-n)
```

The function `fromEnum` has a fairly obvious task: map each member of the enumerated type to a unique integer. We define this mapping using the function `natToInt` below:

```
natToInt :: Nat -> Int
natToInt Zero = 0
natToInt (S n) = 1 + (natToInt n)
```

The function `enumFrom` is implemented by a call to `natFrom`. Note that we do addition directly upon values of type `Nat` (using `addNat`), rather than converting them to integers first and then performing the addition. We do so for efficiency reasons. Due to the way `addNat` is defined the cost of adding two natural numbers is proportional to the magnitude of the second value in the addition. However, if we were to convert both natural numbers to integers first and then add them, the cost would be proportional to twice the sum of the magnitudes of both numbers (due to the cost of converting them to integers and to the cost of converting the result back to a natural number). If we are sure that the second number in the addition is a small natural number (i.e. `S Zero` in this case) the cost of the addition is negligible.

```
natFrom :: Nat -> [Nat]
natFrom n
   = n : (natFrom (addNat n (S Zero)))
```

The last required function, `enumFromThen`, is implemented by a call to `natFromThen`. There are three general cases to consider when generating an enumeration with a step value. Each case is handled separately by a guard in the definition of `natFromThen`. The first case occurs when the first value of the enumeration and the next value are the same. In such a case we simply generate an infinite list of the first value. The second case occurs when the second value is less than the first. It is possible to generate a finite list of naturals of decreasing magnitude, but an infinite list is not possible because the naturals do not cover negative values. To simplify the discussion we ban decreasing enumerations and report an error if an attempt is made to generate such a list. The third case occurs when the second value is strictly greater in magnitude than the first value, in such a case we call the function `natFromThen'` to generate the list.

```
natFromThen :: Nat -> Nat -> [Nat]
natFromThen first second
   | fromEnum first == fromEnum second
       = infiniteFirsts
   | fromEnum second < fromEnum first
       = error "the second value is less than the first"
   | otherwise = natFromThen' first stepSize
     where
     infiniteFirsts = first : infiniteFirsts
     stepSize = toEnum ((fromEnum second) - (fromEnum first))

natFromThen' :: Nat -> Nat -> [Nat]
natFromThen' next step
   = next : (natFromThen' (addNat next step) step)
```

After all the hard work, we can finally make use of the fact that `Nat` is now an instance of `Enum`:

```
Hugs> map fromEnum (take 10 [Zero ..])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Hugs> putStr (unlines (map show [intToNat 4, intToNat 8 .. intToNat 16]))
S (S (S (S Zero)))
S (S (S (S (S (S (S (S Zero)))))))
S (S (S (S (S (S (S (S (S (S (S Zero))))))))))
S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S Zero))))))))))))))))
```

C Monadic IO Example

```
{-------------------------------------------------------------------------------

        Author:         Bernard Pope
        Date:           22/1/98

        Notes:          Fun with Haskell Monadic IO. Contains code for
                        building a balanced BST from a list, and code for
                        interactively exploring a Binary Search Tree.
                        The nodes in the tree must be an instance of the
                        Show class.

        Usage:          For example:

                        exploreTree (bTree [1..10])

                        l/L for navigating left, r/R for navigating right.
                        u/U for going back up one level in the tree.
                        q/Q for quiting the navigation.
                        Any other key will be ignored and the choice of
                        direction will be asked for again.

-------------------------------------------------------------------------------}

module ExploreTree (bTree, Tree, exploreTree) where

{-
        Standard Binary Tree

-}

data Tree a = Node (Tree a) a (Tree a) | NilTree
              deriving (Show)


{-
        function:       bTree, bTree'

        description:    builds a balanced binary tree from a list of elements.

-}


bTree :: [a] -> Tree a

bTree list
   = bTree' list (length list)

bTree' :: [a] -> Int -> Tree a
```

```
bTree' [] _ = NilTree

bTree' list@(x:xs) listLength
   = (Node (bTree' left leftLength) mid (bTree' right rightLength))
     where
     (left, mid:right) = splitAt leftLength list
     leftLength = listLength `div` 2
     rightLength = listLength - (leftLength + 1)


{-
        enumerated type for the user's desired action.

-}

data Direction = GoLeft | GoRight | GoUp | Quit


{-
        function:       exploreTree

        description:    interactively traverse a binary tree using the IO
                        monad.

                        Calls the function exploreTree' which remembers the
                        ancestor nodes of the current node in the tree so
                        that it can traverse upwards.

-}

exploreTree :: Show a => Tree a -> IO ()

exploreTree t
   = exploreTree' [] t


{-
        function:       exploreTree'

        description:    Takes a list of ancestor nodes (as a stack)
                        and the current node.
                        Prints the current node then prompts the user for
                        the next direction to explore (or quit).
-}

exploreTree' :: Show a => [Tree a] -> Tree a -> IO ()

        -- hit a leaf node (Nil). Can only quit or go back up the tree.
exploreTree' parents NilTree
```

```
   = do
      putStr "NilTree\n"
      r <- response
      case r of
             GoUp     -> upTree parents NilTree
             Quit     -> return ()
             _        -> do
                          putStr "Can only go up\n"
                          exploreTree' parents NilTree

        -- an internal node. Can go left/right, up or quit.
exploreTree' parents t@(Node left node right)
   = do
      putStr (show node)
      putStr "\n"
      r <- response
      case r of
             GoLeft  -> exploreTree' (t:parents) left
             GoRight -> exploreTree' (t:parents) right
             GoUp    -> upTree parents t
             Quit    -> return ()


{-
        function:       upTree

        description:    Takes a list of ancestors (as a stack) and the
                        current node.
                        Traverses up the tree one level (if possible).

-}

upTree :: Show a => [Tree a] -> Tree a -> IO ()

        -- no ancestors, we must be at the top of the tree, re-call
        -- exploreTree' with the current node.
upTree [] node
   = do
      putStr "Can't go up any further\n"
      exploreTree' [] node

        -- at least one ancestor, call exploreTree with the parent and its
        -- ancestors.
upTree (a:as) node
   = exploreTree' as a


{-
        function:       response
```

```
        description:    get the response from the user as to which direction
                        they would like to explore.

-}

response :: IO Direction

response
   = do
      putStr "(L)eft, (R)ight, (U)p or (Q)uit?\n"
      c <- getLine
      case c of
             "l" -> return GoLeft
             "L" -> return GoLeft
             "r" -> return GoRight
             "R" -> return GoRight
             "q" -> return Quit
             "Q" -> return Quit
             "u" -> return GoUp
             "U" -> return GoUp
             _   -> response            -- unknown command, call response again
```