Memorial
University of Newfoundland

Technical Report #2000–02

A COMPARISON OF THE HASKELL AND MIRANDA FUNCTIONAL PROGRAMMING LANGUAGES

by

D.C. Craig

Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5

December 2000

Technical Report #2000–02

A COMPARISON OF THE HASKELL AND MIRANDA FUNCTIONAL PROGRAMMING LANGUAGES

by

D.C. Craig

Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5

December 2000

Abstract

The primary purpose of this paper is to discuss the similarities and differences between Haskell and Miranda. The topics addressed by this report, include both technical issues (such as syntactic and structural variations) and nontechnical issues (for example, differences in the respective distribution licenses). This report assumes that the reader is already familiar with functional programming in general and Miranda in particular.

This paper is not intended to be an exhaustive study of *all* the minutiae of Haskell and how they compare to Miranda. However, an effort will be made to compare and contrast the more commonly used language constructs in Haskell and Miranda. A reasonably comprehensive overview of the language features of Haskell that would typically be taught in an introductory course to functional programming will be addressed. In addition, a summary of some of the advantages and disadvantages of Haskell versus Miranda will be described.

This document should serve well as a transition paper for those who are familiar with Miranda and wish to explore an alternative functional programming language. Note that this technical report makes references to Technical Report 2000–03. Therefore having access to both reports will be helpful.

1 Introduction

Haskell, like Miranda, "is a modern, standard, non-strict, purely-functional programming language." [1] Like other functional programming languages, it supports polymorphic typing, lazy evaluation and higher-order functions.

Generally speaking, the language is very similar to Miranda. Therefore, users who are familiar with Miranda should be able to pick up Haskell relatively easily. That said, however, there are still several subtle (and not so subtle) differences between the two languages that can confuse the novice Haskell programmer. This paper will attempt to identify and shed light on most of these issues.

In cases where a given feature in Miranda is practically identical to a corresponding feature in Haskell, very little explanation is given of the feature itself. The purpose of this paper is not to provide an introduction to functional programming or describe all the intricate low-level details of Miranda or Haskell. Rather, this paper should be seen as giving a reasonable comprehensive background of the feature set of Haskell as it compares to Miranda.

Before embarking on a detailed comparison of the two languages, a short overview of some of the implementations of Haskell interpreters and compilers is provided below.

1.1 Haskell Implementations

There are several implementations of Haskell compilers and interpreters on the Internet. Each implementation comes with its relative strengths and weaknesses. This section provides a very brief overview of the more common Haskell implementations currently available. Most of the information in this section is found in [2].

Due to time constraints, only the first implementation described below (Hugs) was actually downloaded and evaluated.

1.1.1 Hugs

Hugs (which stands for Haskell Users Gofer System) is a successor to Gofer and was written initially by Mark P. Jones, currently from the Oregon Graduate Institute (OGI) of Science and Technology, with contributions from many others around the world. Hugs was initially derived from Gofer version 2.30b.

The examples presented in this paper have all be tested with the `hugs` Haskell interpreter (released in February 2000) running on Digital Unix (`garfield`). It should be noted, however, that `hugs` should successfully compile and run on several other Unix-like operating systems on a variety of architectures. A precompiled binary even exists for Microsoft Windows. Instructions for starting and using the interpreter on Unix machines are presented in Technical Report 2000–03.

Like Miranda, `hugs` comes with a command-line environment which provides a means for users to edit and compile scripts interactively. Unfortunately, because it is an interpreter, scripts may execute slower than other Haskell implementations. However, as a teaching tool, Hugs comes highly recommended. The most recent version of Hugs is almost 100% compliant with the Haskell 98 specification and has reasonable resource requirements.

1.1.2 GHC, the Glasgow Haskell Compiler

This is a compiler which, itself, is written in Haskell. It is primarily intended as a research tool and generates fast code. GHC has a profiling feature which allows developers to evaluate the space and time performance of their programs. Its source is freely available and is portable across Unix and Windows. It also claims to support parallel and concurrent programming as well. There is an extension to this compiler called GPH (Glasgow Parallel Haskell) which allows for even more expressive representation of parallel program through the introduction of a pair of composition primitives to the Haskell language.

1.1.3 nhc98

This Haskell 98 compiler project, initiated by Niklas Röjemo, requires much less memory resources than the other compilers and interpreters described above. It provides heap profiles and debugging capabilities not found in other Haskell implementations. nhc98, like the others, is also quite portable.

2 Comparison of the Type Systems

Like Miranda, Haskell supports several data types representing boolean, characters, numbers; simple data structures such as tuples and lists; and, of course, functions. There are, however, several differences between the type systems of Haskell and Miranda. Some of these differences are small while others are quite substantial.

2.1 Type Variable Representation

One aesthetic difference between Haskell and Miranda is that in Haskell type variables are represented with different letters of the alphabet rather than with a varying number of asterisks. This significantly improves the readability of function types, as shown below. Type variables appearing in manuscripts such as books and papers are commonly assigned letters starting with the Greek alphabet, starting with α, β, γ, and so on. In a Haskell environment (for example, an interpreter running on a textual terminal) different type variables are represented with letters of the English alphabet (`a`, `b`, `c`, etc.).

For example, in Miranda, the type of the composition operator (.) is typically displayed as:

```
Miranda (.)::
(*->**)->(***->*)->***->**
```

Whereas in Haskell, the type of the composition operator would be formally represented as:

$$(.) :: (\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$$

In the `hugs` Haskell interpreter, the type of the composition operator would be displayed as:

```
Prelude> :type (.)
(.) :: (a -> b) -> (c -> a) -> c -> b
```

The Haskell type representation looks much easier to understand than the equivalent Miranda representation.

As an aside, note that in Miranda, one determines the type of an expression by typing two colons after the expression (*expr*`::`). To do the same thing in `hugs` one would use the `:type` command at the prompt (`:type` *expr*). There are several other commands in the interpreter for performing operations such as loading and editing modules and quiting the interpreter. These are discussed in Technical Report 2000–03.

Also note that `Prelude>` is the default `hugs` prompt. `Prelude`, in this case, refers to the startup script (or *module*) that is loaded by a Haskell interpreter, such as `hugs`, by default. This file is just like any other Haskell script file, except that it contains the standard type and function definitions as specified in the Haskell language specification. More information will be presented on the prelude file in Section 6.

2.2 Native Haskell Types

The names of the native data types supported by Haskell are `Bool`, `Char`, `Int`, `Integer`, `Float`, `Double`. There are other, slightly more esoteric types such as `Ordering`, `Maybe`, `IO` and the unit data type: `()`, (the so-called *nullary tuple*).

Note that this demonstrates another fairly minor difference between Haskell types and Miranda types — Haskell types have their first character capitalized, (e.g. `Bool`) whereas in Miranda, the type name is all lower case (`bool`).

Like Miranda, Haskell defines the two boolean values `True` and `False`, which may be used in boolean expressions. `Int` and `Integer` are builtin data types for fixed size and arbitrary sized integers, respectively. `Float` and `Double` are builtin data type for single precision and double precision floating point numbers, respectively. Note that this demonstrates another difference between Haskell and Miranda — Miranda does not distinguish between difference types of numbers (numbers are of the type `num`) while Haskell does differentiate amongst different numeric types. In other words, Haskell treats numeric representations less coarsely than Miranda.

The `Char` type represents 16-bit Unicode characters, hence the `Char` type allows international character sets to be represented. The `String` type is simply defined as a list of character values, as is the case in Miranda. Like Miranda, Haskell allows for the convenience of enclosing a sequence of characters in side double quotes to represent a list of characters (i.e. a string). For example, `"abc"` is the same as `['a', 'b', 'c']`. Incidentally, Haskell implementations typically represent the `String` type as a type synonym, as follows:

```
type String = [Char]
```

2.2.1 Type Synonyms

Note that apart from the syntactic differences with respect to defining type synonyms, Miranda's and Haskell's type synonyms have the same semantics. That is, they don't introduce a new bona fide type — rather they just introduce a synonym for an existing type.[1] Like Miranda, type synonyms in Haskell may also be parameterized. For example:

```
type Pair a b = (a,b)

swapper :: Pair Int Char -> Pair Char Int
swapper (i,c) = (c,i)
```

2.2.2 Function Types

Functions are defined in a manner very identical to Miranda:

```
interleave :: [a] -> [a] -> [a]
interleave [] [] = []
interleave (x:xs) (y:ys) = x:y:interleave xs ys
```

The above example shows the type of the function followed by its actual definition. However, like Miranda, the `hugs` interpreter supports automatic type inference so that it isn't necessary to give the type of the function when defining it. If one chooses to define the type of the function, `hugs` will generate type error messages if any inconsistencies between the given function type and the inferred function type are found. Unfortunately, the type error messages, while a bit more verbose than the Miranda type error messages, are still somewhat cryptic, especially to the novice user.

Later sections will discuss other aspects of function definition including, for example, layout, guards and `where` clauses.

2.2.3 List and Tuple Types

The list types of Haskell are identical to the list types of Miranda. Syntactically, a list consists of a series of values of homogeneous types enclosed in square brackets and separated by commas (for example, `[1,3,5]` and `[True,False,True]`). Similarly, there are no appreciable differences between tuple types of the two languages — in both languages, tuples are a sequence of potentially heterogeneous types enclosed in parenthesis and separated by commas (for example, `('a', True, [1,2,3])`).

2.3 Type Classes

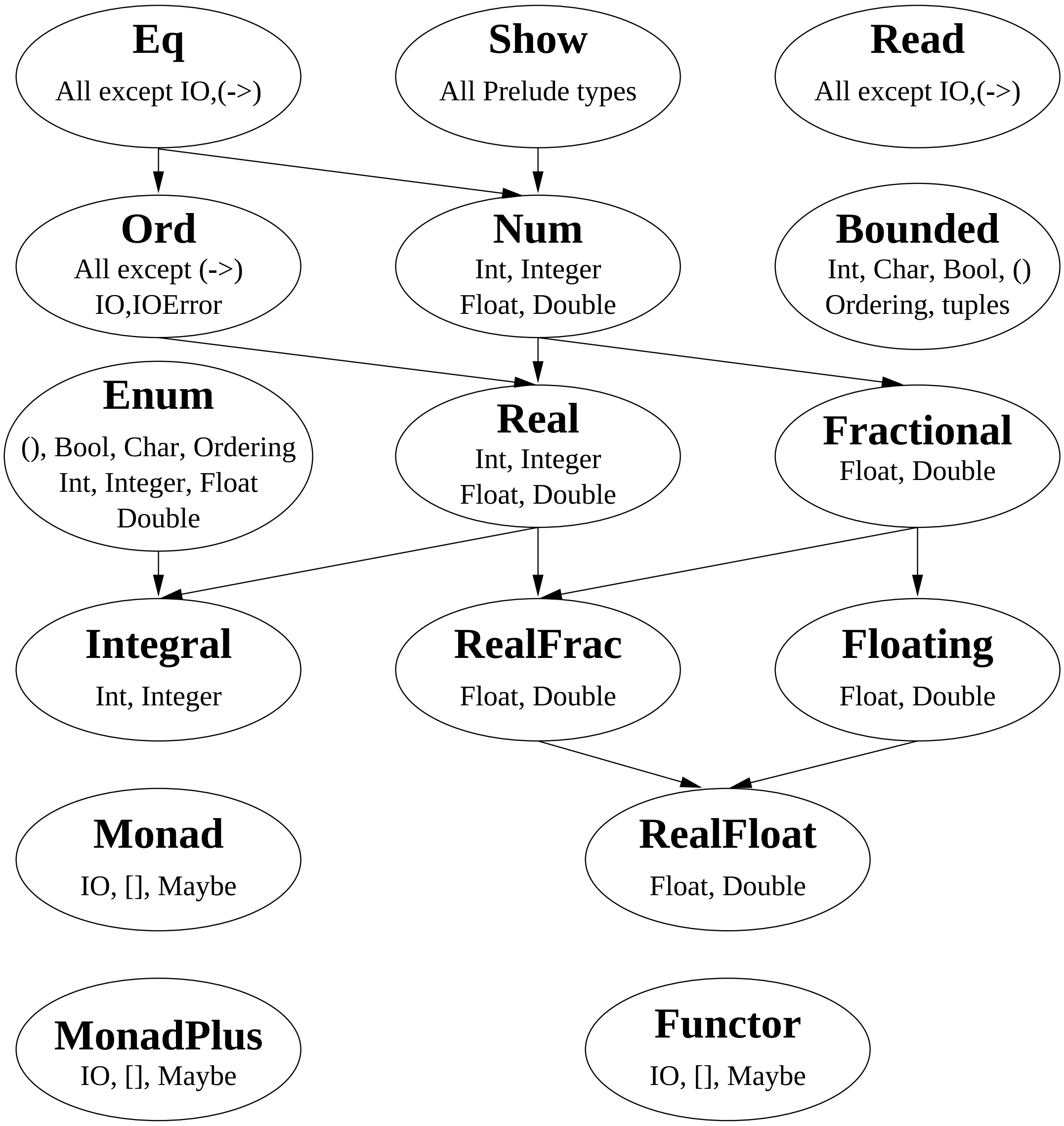
The most substantial (and potentially the most confusing) difference between the Miranda and Haskell type systems is that Haskell supports a concept called *type classes* while Miranda does not. Type classes provide a means of grouping types which have similar properties (or methods) together. In addition, behaviour from one type class may be inherited by another type class so as to increase the specialization of the derived type class. These type classes become part of the official type of a function, when applicable.

Figure 1, which was taken from the Haskell specification, describes all the type classes usually found in a Haskell implementation as well as the type instances which are contained in each type class. Note that a type instance can belong to several different type classes.

To see the role that type classes play in the Haskell type system consider the following Haskell session which determines the types of some trivial expressions:

[1] Haskell actually has a means to create a new type via the (appropriately named) `newtype` declaration.

Figure 1: Haskell Type Classes




```
Prelude> :type 1
1 :: Num a => a

Prelude> :type 2.3
2.3 :: Fractional a => a

Prelude> :type elem
elem :: Eq a => a -> [a] -> Bool
```

Under Miranda, the types of both the value `1` and `2.3` would simply be `num`. However, Haskell attempts to distinguish between numbers of different types. The notation `Num a => a` can basically be read as "the given expression is of type `a` where `a` is an instance of the type class `Num`." Hence, looking at the `Num` class of Figure 1, we can see that the number `1` can be an instance of type `Int`, `Integer`, `Float` or `Double`.[2]

In the second example, the type of the value `2.3` is `Fractional a => a`. Note that the type `Num a => a` was not used because the type class `Num` encompasses all types of numbers including integers, floating point numbers and even complex numbers. Clearly `2.3` is not an integral number and therefore cannot be classified in the `Num` class. Consequently, Haskell indicates that this value is either a single or double precision floating point number (either `Float` or `Double`) by claiming its type to be `Fractional a => a`

The third example gives the type of the function `elem` which is a boolean function that determines if a given value is an element of a given list. The part of the type after the `=>` (that is, `a->[a]->Bool`) is fairly straightforward — it indicates that the function takes a value of a specific type followed by a list containing the same types of values and produces a boolean result. The `Eq a` which occurs before the `=>` serves to restrict the type of `a` to include only those types which are instances of the `Eq` class; that is, types for which the equality operator is defined. According to Figure 1, we see that most the builtin types are members of the `Eq` class, so this restriction on the types of values which can be processed by the `elem` function is not too draconian.

Note that the function type (`->`) is *not* a member of the `Eq` class. Therefore, we cannot, for example, use `elem` to determine if a given function is present in a list of functions. Essentially, by imposing a constraint (or requirement) on the types of values that can be used by a function, we can intelligently limit the polymorphism of a function to a subset of reasonable types.

2.4 Algebraic Types

Miranda supports the concept of algebraic (or recursive) types for constructing abstract data structures such as trees. Haskell also supports a similar concept using the `data` declaration. Like Miranda, algebraic types may also be parameterized. For example, the following two algebraic types create a parameterized binary tree representation in Miranda and Haskell, respectively:

```
bintree * ::= Atom * | Node (bintree *) (bintree *)

data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

In the above Haskell example, `Tree` is called a *type constructor* whereas `Leaf` and `Branch` are known as *data constructors*. Note that there is nothing preventing us from using the same data constructor names (`Atom`/`Node`) in the Haskell example as used in the Miranda example. The only restriction is that the data constructors start with an upper case letter, like Miranda. Unlike Miranda, however, the type constructor in Haskell, (`Tree`, in this example) must also start with an upper case letter. Building and processing nodes in the above tree structure is very similar between Haskell and Miranda. For example, in Haskell, we could write:

```
atom_list (Leaf x) = [x]
atom_list (Branch t1 t2) = (atom_list t1) ++ (atom_list t2)

test = atom_list (Branch
                        (Branch (Leaf 2)
                                (Branch
                                        (Leaf 3)
```

[2]Haskell appears to want to use the most generic type possible when assigning a type class to a value. Hence the reason why `1` is described as a belonging to the generic `Num` class rather than to the overly specific `Integral` type class.

```
                                    (Leaf 1))
                        )
                        (Leaf 4)
              )
```

As expected, the above function creates a list of all the leaf elements in the fringe of the supplied tree, and the function `test` will display `[2,3,1,4]`.

3 Haskell Operators

Like Miranda, operators in Haskell can be thought of as functions that are applied to two arguments in an infix (as opposed to prefix) manner. A later subsection will describe how infix operators can be used in prefix notation (and visa versa).

The Standard Haskell language specification lists the operators specified in Table 1. These operators should be available in any compliant Haskell implementation. The operators, their precedence and their associativity are typically defined in one or more *prelude* files.

Monads, alluded to by the monadic operators in the above table, essentially allow imperative semantics to be introduced into a purely functional programming language, such as Haskell. For example, Haskell's underlying I/O system is built using a monad. The monad essentially acts as a bridge between the values associated with functional programming with actions that would normally be associated with an imperative language. Monads are a relatively advanced topic in Haskell, and are not discussed in further detail in this paper.

As mentioned earlier, Haskell is a non-strict language. However, should the need arise to employ strict semantics when defining a function the `$!` operator can be used. (The `$!` operator is actually defined in terms of the (lower-level) `seq` operator.) The `$` operator, which represents lazy evaluation, represents the default, non-strict behaviour.

As can be seen from the table, Haskell supports many of the operators featured in Miranda (as well as a some additional operators); however, the syntax for the operators varies significantly in many cases. Table 2 gives a list of common operators in Miranda that are syntactically different from Haskell.

Note that the syntax of the equality and the logical operators in Haskell are the same as the corresponding operators in C. This is helpful to those coming to Haskell from a C background. Of particular interest, one should note that the comment delimiter in Miranda, `||`, is the logical OR operator in Haskell. The comment delimiter in Haskell is two consecutive hyphens and the comment continues until the end of the line. Haskell also supports multiline comments by using an opening {- sequence followed by any number of comments lines followed by a closing -} sequence.

The other relational and mathematical operators (e.g. $<$, $>$, $<=$, $>=$ and `+`, `-`, `/`, `*`, `` `div` ``, `` `mod` ``) are similar in the two languages. The reason for the enclosing backquotes (`) on the last two operators is explained below. Like Miranda, comparisons between list types and comparisons between tuple types are possible and exhibit similar behaviour. For example:

```
Prelude> [1, 2, 3] <= [1, 3, 5]
True

Prelude> "abc" < "xyz"
True

Prelude> (10, 20, 30) < (1, 2, 3)
```

Table 1: Haskell Operators (listed in decreasing precedence)

Operator	Precedence	Associativity	Meaning
`!!`	9	left-to-right	List index
`.`	9	right-to-left	Function composition

`^`
`^^`
`**`
	8	right-to-left	Exponentiation (for different types of bases and exponents)

`*`
`/`
`` `div` ``
`` `mod` ``
	7	left-to-right	Multiplication
			Float/Double division
			Integer division
			Remainder

`+` `-`	6	left-to-right	Addition Subtraction
`:` `++`	5	right-to-left	Cons List Concatenation

`==`
`/=`
`<`
`<=`
`>=`
`>`
`` `elem` ``
`` `notElem` ``
	4	–	equal to
			not equal to
			less than
			less than or equal to
			greater than or equal to
			greater than
			is an element of
			is not an element of

`&&`	3	right-to-left	logical AND
`||`	2	right-to-left	logical OR
`>>` `>>=`	1	left-to-right	Monadic Bindings
`=<<`	1	right-to-left	Monadic Binding

`$`
`$!`
`` `seq` ``
	0	right-to-left	lazy application
			strict application
			force evaluation

```
False
```

The list concatenation operator (`++`) and cons operator (`:`) are supported in Haskell and have similar semantics as in Miranda. Therefore, the following operations are possible:

```
Prelude> [1,2,3] ++ [4,5,6,7]
[1,2,3,4,5,6,7]

Prelude> "abc" ++ "xyz"
"abcxyz"

Prelude> [(1,True), (2,False)] ++ [(3,True),(4,True)]
[(1,True),(2,False),(3,True),(4,True)]
```

Table 2: Differences between common Haskell and Miranda operators

Miranda	Haskell	Meaning
`=`	`==`	is equal
`~=`	`/=`	is not equal
`~`	`not`	logical NOT
`\/`	`||`	logical OR
`&`	`&&`	logical AND

```
Prelude> 1:[]
[1]

Prelude> 1:(2:(3:[]))
[1,2,3]

Prelude> 'x':"yz"
"xyz"
```

Note that the list subtraction operator in Miranda (`--`) is the comment delimiter in Haskell. Strangely, list subtraction does not appear to be part of the standard Haskell library; however, it is relatively easy to define a custom list subtraction function as follows:

```
lsub x [] = x
lsub x (b:y) = lsub (remove b x) y
```

The `remove` function, as expected, removes element `b` from list `x` and will be defined in the next section. Also note that the length operator in Miranda (`#`) is not available in Haskell; however, Haskell defines a `length` function as part of its standard library of functions.

As expected, function composition operator `(.)` is the same between the two languages and is defined as:

```
(.)            :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x       = f (g x)
```

Haskell also has definitions of all the standard mathematical functions one would expect to find in a modern programming language. The names of the functions are similar to their C counterparts, for example `exp`, `log` and `sqrt`. The trigonometric functions `sin`, `cos`, `tan`, `asin`, `acos` and `atan` are supported as well.

3.1 Operator Sections

Haskell, like Miranda, also supports sections on operators. For example, `(*2)` represents a partial application of the multiplication operator which doubles a given number and `(1/)` represents the reciprocal section. Sections may be used in much the same way as they are used in Miranda. For example:

```
Prelude> (/2) 5
2.5
```

```
Prelude> (1:) [3,5,7]
[1,3,5,7]
```

A section that subtracts x from y number must be written as `(subtract x) y`. This is because `(-x)` is a number, not a partial application of the minus sign. The `-` in this case is treated as a unary minus as can be seen by studying at the types of `(-1)` and `(subtract 1)`.

```
Prelude> :type (-1)
-1 :: Num a => a

Prelude> :type (subtract 1)
subtract 1 :: Num a => a -> a
```

3.2 Infix and Prefix operators

In Haskell, operators (and 2-argument functions in general) can be used in both prefix or infix notation. To see how this can be done, consider the following examples:

```
Prelude> 1/2
0.5

Prelude> (/) 1 2
0.5

Prelude> div 7 3
2

Prelude> 7 `div` 3
2

Prelude> elem 'x' "this list has an x"
True

Prelude> 'x' `elem` "this string has an x"
True
```

Note that before using a function in infix notation, the function name must be enclosed in backquotes (‘). This feature adds flexibility to the language, but should not be abused. Indeed, using general (non-mathematical) function names in infix notation can make function application more difficult to recognize for a human reader. In the last call of the `elem` function above, for example, confusion between the function name and its parameters can result, especially if one is accustomed to seeing function application done using prefix notation.

It should be noted that Miranda has a similar ability to use functions in infix notation as well by preceding the function name with `$`. For example, `1 $elem [1, 2, 3]` is `True` in Miranda.

4 Functions

The cornerstone, by definition, in any functional programming language is the function. The section presents a reasonably comprehensive overview of functions in Haskell and how their definition and type differ from Miranda. This section will explain the use of guards, pattern matching, `where` clauses and the roles they play in function definition and application.

Functions in Haskell, like Miranda, have non-strict semantics. In other words, if a function doesn't have to completely evaluate all of its arguments in order to deduce a return value, then it won't. [3] This allows function calls such as the following, for example:

```
Prelude> take 5 [1..]
[1,2,3,4,5]
```

As alluded to in the first section, functions in Haskell (as in Miranda) can be polymorphic. For example, the `length` function can take as a parameter a list containing `Integer`s, `Char`s, tuples or even other lists and the function will operate as expected. Haskell functions are also strongly typed. Before functions are evaluated, the function parameters are checked to ensure that they are compatible with the operations being performed on them. Definitions of function types are checked with their actual definition to make sure they are both consistent. As mentioned in Section 2.2.2, `hugs` will infer function types when they were not explicitly specified. Any type mismatches or conflicts are caught and reported by the interpreter.

As with Miranda, the general form of a function definition in Haskell is:

$$funcname\ [opt\_args] = RHS$$

where *functname* is the name of the function, *opt_args* is an optional list of formal arguments and/or patterns (separated by white-spaces) and the *RHS* is the actual definition of the function. For example,

```
-- Square the supplied number
square x = x * x

-- Returns the first element of non-empty list.
head (x:xs) = x
```

4.1 Guards

In Miranda, multiple guards (or conditional expressions) in a function take the general form:

$$\begin{aligned} LHS &= RHS_1\ ,\ guard_1 \\ &= RHS_2\ ,\ guard_2 \\ &\ldots \\ &= RHS_n\ ,\ guard_n \end{aligned}$$

where $guard_i$ is a boolean expression. Haskell also supports multiple guards but the syntax is quite different. In the general form, the guards are actually placed *before* the `=` sign and are prefixed by a vertical bar (`|`). So, the general form of a Haskell function with guards is:

$$\begin{aligned} LHS\ &|\ guard_1 = RHS_1 \\ &|\ guard_2 = RHS_2 \\ &\ldots \\ &|\ guard_n = RHS_n \end{aligned}$$

[3] However, we briefly saw in the previous section how this default behaviour can be overridden, if necessary.

For example, consider the simple definition of the `fact` function specified in [6] which determines the factorial of the number supplied to it. The following script can be typed into a file and loaded it into the Haskell interpreter for processing.

```
fact :: Integer -> Integer
fact n | n == 0 = 1
       | n < 0  = error "negative argument to fact"
       | n > 0  = n * fact (n - 1)
```

Note that the boolean `n > 0` on the last line could have been replaced with `otherwise`. The `otherwise` guard behaves as the general "catch-all" case, as it does in Miranda. Haskell actually defines `otherwise` to be synonymous with `True`. The above `fact` function behaves as expected:

```
Main> fact 12
479001600

Main> fact 0
1

Main> fact (-12)
Program error: negative argument to fact
```

Note that the prompt for `hugs` changes from `Prelude>` to `Main>` when you load in a script (or module) that does not specify an internal module name. `Main` can be thought of as the default module if the script does not explicitly specify a name. Modules essentially provide a means in Haskell for namespace management. A module name can be specified using the `module` keyword. For example, if at the beginning of the above script, the text `module Fact where` was included, then the module name would become `Fact` and the prompt would reflect this.

In the last test above, we must parenthesize the `-12`. If we didn't, Haskell (like Miranda) would interpret the expression as `fact - 12`, which would generate a very interesting type error.

Haskell allows for a more conventional conditional syntax in the form of a familiar `if` ... `then` ... `else` expression. For example, the `remove` function used to define list subtraction in the previous section can be defined as:

```
remove b [] = []
remove b (a:x) = if a == b then x else a:remove b x
```

In this case, the guard is placed *after* the `=` sign. Note that it is very easy to syntactically confuse the two guard styles, especially when it comes to the placement of the `=` sign. If desired, `if` ... `then` ... `else` expressions may nest, so we can express, for example, a `signum` function (which returns -1, 0 or 1 if the number supplied is negative, zero or positive, respectively) on a single line.

```
signum x = if x > 0 then 1 else if x == 0 then 0 else -1
```

However, in cases where there are more than two mutually exclusive conditions, using the first form presented above (with the vertical bars) is usually easier to understand than nested `if` ... `then` ... `else` expressions.

If you forget a guard for a specific case, the `hugs` interpreter generates a rather poor error message. For example, consider the following (incorrect and naïve) definition of `isin`, which tries to determine if an element is a member of a list.

```
isin a xs | xs == []            = False
          | (head xs) == a      = True
```

When tested with a case in which the element being sought is not at the head of the list, this function generates the following error message:

```
Main> isin 3 [1, 3]

Program error: {isin instEq_v30 3 [1,Num_fromInt instNum_v35 3]}
```

Obviously, this is not a very user-friendly error message. In the same situation, Miranda generates the more comprehensible "`missing case in definition`" error message. To fix the problem and eliminate the program error message above, we simply need to add the recursive case to the `isin` function definition[4]:

```
isin a xs | xs == []            = False
          | (head xs) == a      = True
          | otherwise           = isin a (tail xs)
```

Better still, of course, we can define the function as:

```
isin a (x:xs) | a == x    = True
              | xs == []  = False
              | otherwise = isin a xs
```

Section 6 demonstrates a much more succinct (and clever) way of expressing membership in a list using function composition.

4.2 Higher-Order Functions

Haskell supports higher-order functions (HOFs) in much the same way as Miranda. That is, functions can take other functions as inputs and produce functions as outputs. For example, consider the function application `f a b`. Function application binds left to right (as it does in Miranda), so this implies that `f a b` is equivalent to `(f a) b`. Therefore, `f` takes a single argument, `a`, and produces a function `(f a)` which is then applied to `b` to generate the final result.

The two popular higher order functions `map` and `filter` are part of the standard library for Haskell, and they are discussed in more detail in Section 6

4.3 Lexical Scoping

Like Miranda, Haskell supports lexical scoping through the use of `where` clauses, which lets the user create variables which are local to a function definition. However, in addition Haskell, unlike Miranda, also supports so-called `let` clauses as well, which allows a programmer create variables local to a single expression.

For example, consider the following function to compute the distance between two cartesian coordinates:

[4] Note that in Haskell, the function that returns the first element of a list is `head` while in Miranda, the function is `hd`. Likewise, the `tail` function that returns the remainder of the list in Haskell, while Miranda abbreviates this function name to `tl`

```
distance x1 y1 x2 y2 =
 let xd = x2 - x1
     yd = y2 - y1
     xsq = xd * xd
     ysq = yd * yd
 in sqrt (xsq + ysq)
```

Note that the variables in the `let` clause can only be used in the following expression and not globally throughout the function. This gives Haskell an added level of lexical scoping which is not available in Miranda.

As expected, the variables in the `where` clause could be used anywhere in the function definition in which it appears. `where` clauses may also be nested, if necessary, just as long as the offside rules are obeyed. For example:

```
contrived_func a = b + b
 where b = c + 1
        where c = a + 2
```

4.3.1 A Note on Layout

Through the use of layout, Haskell programmers can avoid having to use semicolons and other such terminators to resolve ambiguity. The layout rules employed by `hugs` are similar to those used by Miranda. There are two main points to remember about layout in Haskell.

First, when using a `where` or `let` clause, the first (non-whitespace) character following these keywords determines where the column in which all subsequent clauses must begin on the following lines. For example, the following function to compute the quadratic formula is correct:

```
quadform a b c
 | a == 0    = error "not quadratic"
 | disc < 0  = error "complex roots"
 | disc == 0 = [-b / a2]
 | otherwise = [(-b + radix) / a2, (-b - radix) / a2]
 where  disc  = b * b - 4 * a * c
        radix = sqrt disc
        a2    = 2 * a
```

However, if the `radix = sqrt disc` line was outdented one or more spaces to the left, then it would have been offside and deemed unparsable by the interpreter.

Second, when continuing a clause (for example, a function definition) on a second line, ensure that the second lines begins to the right of the first line of the clause. For example, in the above definition, note that the guards and the `where` keyword for the `quadform` starts at least one column to the right of where the first line (`quadform a b c`) started.

Semicolons and braces may be used for enforce a particular layout representation. While useful at times, this method can make a script more difficult to follow.

4.4 Pattern Matching

Haskell uses a pattern matching strategy that is very similar to that used by Miranda. As seen earlier the same function may have different left hand sides and Haskell will use pattern matching to determine which one to use. Pattern matching may occur over a variety of types and the idioms that apply in Miranda, generally apply to Haskell, too.

For example the pattern `n - 1` will match any type in the type `Integral` type class that is greater than one. The familiar pattern `x:xs` will match any list containing at least one element and the pattern `(a, b, [])` will match any tuple in which the third element is the empty list. As seen in Section 2, patterns are also used for matching abstract data types.

Haskell does allow some "syntactic sugar" when specifying function parameters. While these features do not add to the functional capabilities of the language, they do offer some convenience to the programmer.

4.4.1 As-patterns

As-expressions allow patterns that appears on the left hand side of an equation to be cached, so that it can be used on the right hand side of an equation too. For example, consider the following (parallel-friendly) function which determines the last element of a non-empty list:

```
lastel [x] = x
lastel (x1:x2:xs) = last (x2:xs)
```

We can use an as-expression for the last equation so as to avoid having to retype the pattern `x2:xs` on both the left and right hand sides:

```
lastel [x] = x
lastel (x1:s@(x2:xs)) = last s
```

Obviously, in short examples, such as the one above, the benefit of as-expressions is small. However, if the pattern that we want duplicated is large and complicated, then using as-expressions can make function definitions that have duplicated patterns on the LHS and RHS easier to change since only the expression on the LHS will have to be modified. They can also aid in allowing readers to understand the function because similarities between patterns on the LHS and RHS are made explicit by the as-expression.

4.4.2 Wild-card Parameters

Haskell allows the use of the underscore character as a *wild-card* character in the parameter list of the LHS of an equation to represent a value which we do not care to use on the RHS of the equation. This feature can act as a documentation aid by indicating to the reader that the value supplied to the function is not used in the actual definition on the RHS. Underscores can even be used twice in the same function definition.

For example, consider the definition of the `null` function which is a standard function that determines if the list supplied to it has no elements. One way to define it is as follows:

```
null            :: [a] -> Bool
null []          = True
null (_:_)       = False
```

Note that we could also have defined the `null` function as follows:

```
null2 xs = if xs == [] then True else False
```

However, this function has the type `null2 :: Eq a => [a] -> Bool` while the original function has the type `null :: [a] -> Bool`. This means, for example, that the first `null` function can be used on a list of functions, while the second one may not, because function types are not members of the `Eq` type class.

5 Iterative expressions

This section describes some of the mechanisms available in Haskell for list generation. As with Miranda, there are two ways to automatically generate lists of elements in Haskell — using list comprehensions and dot-dot notation. Again, the similarities between Haskell and Miranda in their respective support for these mechanisms are obvious.

5.1 List Comprehensions

Like Miranda, Haskell supports list comprehensions (or ZF-expressions), albeit with a minor syntactic and semantic difference. The basic format of the list comprehension in Haskell is:

$$[\mathit{expression} \mid \mathit{qualifier}_1, \mathit{qualifier}_2, \ldots, \mathit{qualifier}_n]$$

where $\mathit{qualifier}_i$ is either a generator or a filter. This is almost identical to Miranda, except that a comma rather than a semi-colon is used to separate generators and filters. For example, to generate the list of all products of numbers between 1 and 10 that are divisible by 2 or 3, one could write the following in Haskell:

```
[ x * y | x <- [1..10], y <- [1..10], x `mod` 2 == 0, y `mod` 3 == 0]
```

Whereas in Miranda, (taking into account other syntactic differences as well), one would write:

```
[ x * y | x <- [1..10]; y <- [1..10]; x mod 2 = 0; y mod 3 = 0]
```

We can use the list subtraction, `lsub`, and the `remove` functions shown earlier to define a list permutation function using a list comprehension as follows:

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

With the exception of the comma instead of the semicolon separating the qualifiers, and the use of the `lsub` function instead of the `--` operator in Miranda, the above Haskell definition of the permutation function is identical to the Miranda version.

5.1.1 Recurrence Relations

Haskell, unfortunately, does not support generators of the form:

```
pattern <- exp, exp' ..
```

or more specifically,

```
x <- a, f x ..
```

which causes x to sequentially take on the values `a`, `f a`, `f (f a)`, and so on. This generator is very useful in expressing recurrence relations, where previous values of a sequence are used in order to generate subsequent values.

An example of its use is in the definition of the Fibonacci series. For example:

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

Haskell can support recurrence relations, however it is neither as succinct nor as immediately intuitive as the Miranda equivalent:

```
fibs = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

In the Haskell equivalent of the Fibonacci function, the `fibs` function is "primed" with the first couple of Fibonacci numbers and the function then generates subsequent numbers via the list comprehension, by, in effect, chasing its tail.

It should also be noted that Haskell does not support diagonalizing ZF-expressions, which are supported in Miranda using the form `[ exp // qualifiers ]`.

5.2 Dot-dot expressions

The dot-dot operator (`..`) notation for representing lists of finite and infinite elements is supported in Haskell and has a similar syntax as in Miranda. This includes the syntax for producing infinite lists and for creating lists with constant intervals between consecutive elements. For example, the following operations are legal in Haskell and Miranda and produce similar results:

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]

Prelude> [0..]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17 {Interrupted!}

Prelude> [1,3..11]
[1,3,5,7,9,11]

Prelude> [0,(-1/8)..(-1)]
[0.0,-0.125,-0.25,-0.375,-0.5,-0.625,-0.75,-0.875,-1.0]
```

As can be seen from the above examples, the syntactic and semantic behaviour of `..` expressions are identical in Haskell and Miranda.

Unlike Miranda, however, Haskell's dot-dot notation also works with `Char` types to produce strings. As shown by the following examples, the dot-dot idioms regarding the intervals between the characters of the generated strings as well as the ordinal direction are preserved for character lists too:

```
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"

Prelude> ['z','x'..'a']
"zxvtrpnljhfdb"

Prelude> ['\0'..]
"\NUL\SOH\STX\ETX\EOT\ENQ\ACK\a\b\t\n\v\f\r\SO\SI\DLE
 \DC1\DC2\DC3\DC4\NAK\SYN\ETB\CAN\EM\SUB\ESC\FS\GS\RS
 \US !\"#$%&'()*+,-./0123456789:;<=>?@
 ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`
 abcdefghijklmnopqrstuvwxyz{|}~\DEL
 \128\129\130\131\132\133\134\135
 \136\137\138\139\140\141\142\143
 \144\145\146\147\148\149\150\151
 \152\153\154\155\156\157\158\159
```

```
\160\161\162\163\164\165\166\167
\168\169\170\171\172\173\174\175
\176\177\178\179\180\181\182\183
\184\185\186\187\188\189\190\191
\192\193\194\195\196\197\198\199
\200\201\202\203\204\205\206\207
\208\209\210\211\212\213\214\215
\216\217\218\219\220\221\222\223
\224\225\226\227\228\229\230\231
\232\233\234\235\236\237\238\239
\240\241\242\243\244\245\246\247
\248\249\250\251\252\253\254\255"
```

Because the character set used by `hugs` is finite, you cannot produce an infinite string, as the third example above shows. Note that `hugs` produces either helpful mnemonics or escape-codes for unprintable characters.[5]

6 Haskell Library

As described earlier, many of the standard type, operator and function definitions used in the `hugs` interpreter are defined in file called `Prelude.hs` which is loaded and parsed when the `hugs` interpreter is started. The definitions included in this file then become available for the user. This section provides an overview of some of the definitions that are found in this file. With respect to the `hugs` implementation, a considerable portion of the standard Haskell functionality is actually defined in the `Prelude.hs` file itself. As will be seen below, `Prelude.hs` even defines many of the operators (except '`:`') and their respective precedences. This means that the `hugs` interpreter itself is only responsible primarily for parsing scripts, assessing type consistency and evaluating scripts and not for providing the actual definitions for the types and functions that end-users would use.

Many of the code segments presented in this section are taken directly from the `Prelude.hs` file of the `hugs` Haskell implementation.

6.1 Type Classes and Type Instances

Section 2 discussed the concept of type classes and type instances in Haskell. This section provides details for how the type classes and the type instances are actually defined in Haskell by the `Prelude.hs` file.

The `Prelude.hs` defines many of the type classes and instances which are commonly used in Haskell. For example, recall the `Eq` type class. Type instances belong to the `Eq` class if the concept of equality makes sense to the type instance. The concept of equality applies to numeric type instances and the character type instance, but not to function types. Therefore, the numeric and character types will be instances of the `Eq` class, but the function type will not.

The `Eq` type class is declared as follows:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
```

[5]The last example actually produced a continuous string of 256 characters. Newlines were introduced into this string so as to allow the entire output to be displayed.

```
    -- Minimal complete definition: (==) or (/=)
    x == y      = not (x/=y)
    x /= y      = not (x==y)
```

In the declaration, the `(==)` and `(/=)` are operations on the class and default definitions for each of these operations are provided. As expected, they are each defined as boolean opposites of one another. The above declaration basically says that type `a` is an instance of the `Eq` class if the operations `(==)` and `(/=)` are supported by type `a`.

The `Prelude.hs` file defines the `Integer` class to be a member of the `Eq` class:

```
instance Eq  Integer where
    (==)    = primEqInteger
```

Here, the `(==)` operation is defined for `Integer`s by using the `primEqInteger` method. This method is defined (in C) internally by the interpreter. Note that the `(/=)` method is not explicitly defined. This is because the existing definition of `(/=)` in the `Eq` class, which defines inequality in terms of equality, is sufficient. Similar instance declarations exist for the other numeric types (`Int`, `Float` and `Double`) and for the `Char` type too.

Next, the `Eq` type class can be used to define the `Ord` type class which is used to express the idea of relational order within a type. The `Ord` type class is derived from the `Eq` type class as follows:

```
class (Eq a) => Ord a where
    compare                :: a -> a -> Ordering
    (<), (<=), (>=), (>)   :: a -> a -> Bool
    max, min               :: a -> a -> a

    -- Minimal complete definition: (<=) or compare
    -- using compare can be more efficient for complex types
    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT

    max x y   | x >= y      = x
              | otherwise   = y
    min x y   | x <= y      = x
              | otherwise   = y
```

The notation at the beginning of the declaration:

```
class (Eq a) => Ord a where
```

basically states that the `Ord` type class inherits all the operations from the `Eq` class and also defines its own operations as well (after the `where` keyword). This is similar, in principle to type-derivation commonly found in object-oriented systems. In this context, `Eq` is a superclass of `Ord`, while `Ord` is called a subclass `Eq`. The `Integer` type instance is made a member of the `Ord` type class by defining the `compare` operation as follows:

```
instance Ord Integer where
    compare = primCmpInteger
```

Note that once the `Integer` instance defines the `compare` option, all the other operations "fall into place." Again, `primCmpInteger` is a low-level C routine defined by the interpreter to compare integer values. It returns a value of the `Ordering` enumerated type (that is, `GT`, `EQ` or `LT` depending upon whether one entity is greater than, equal to or less than the other entity, respectively). Similar instance declarations exist for the other numeric type instances and for the `Char` instance, too.

Another instance of the `Eq` type class is defined for lists as follows:

```
instance Eq a => Eq [a] where
    []     == []     =  True
    (x:xs) == (y:ys) =  x==y && xs==ys
    _      == _      =  False
```

The notation `Eq a => Eq [a]` states that before we can establish the equality of two lists containing values of type `a`, the type `a` itself must be a member of the `Eq` type class. Note that because strings are simply lists of characters, the above instance declaration is used to define equality of strings too (in this case, the tye of `a` would be `Char`).

The above type instance declarations for the `Integer` class made it a member of the `Eq` type class and the `Ord` type class. However, as can be seen from the figure showing all the type classes (Figure 1), we note that `Integer` is also an instance of the `Num`, `Real`, `Integral`, `Enum`, `Read` and `Show` type classes too. The `Prelude.hs` file contains appropriate type instance declarations which makes the `Integer` type instance a member of these classes, too.

6.2 Operator Definitions

The `Prelude.hs` defines the operators, their precedence and their associativities as the following script segment shows:

```
infixr 9  .
infixl 9  !!
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`, :%, %
infixl 6  +, -
--infixr 5  :    -- this fixity declaration is hard-wired into Hugs
infixr 5  ++
infix  4  ==, /=, <, <=, >=, >, `elem`, `notElem`
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

The first component of each line gives the associativity of the operator. `infixr` indicates that it is right associative; `infixl` means that the operator is left associative; `infix` implies that associativity is immaterial. The next component on each line is the precedence of the operator with `9` being the highest and `0` being the lowest. Finally, the operators themselves are listed, separated by commas. Note that the infix form of the operators are given. Therefore, we use `` `div` `` rather than just `div` for the integer division operator.

Many of the definitions of the lower-level numeric operators are actually defined in C code in the interpreter. Therefore, for example, for the type instance `Integer`, we have the following instance declaration defining `Integer` to be a member of the `Num` type class:

```
instance Num Integer where
    (+)           = primPlusInteger
    (-)           = primMinusInteger
    negate        = primNegInteger
    (*)           = primMulInteger
    abs           = absReal
    signum        = signumReal
```

6.3 Common Function Definitions

Many of the familiar higher order functions present in Miranda are also present in Haskell and are defined in the `Prelude.hs` file. For example, the following definitions are supplied in the prelude file for `hugs`:

```
map              :: (a -> b) -> [a] -> [b]
map f xs          = [ f x | x <- xs ]

filter           :: (a -> Bool) -> [a] -> [a]
filter p xs       = [ x | x <- xs, p x ]

concat           :: [[a]] -> [a]
concat            = foldr (++) []

foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      = z
foldl f z (x:xs)  = foldl f (f z x) xs

foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

Note that `map` and `filter` are defined in terms of list comprehensions rather than as recursive functions. However, the ultimate semantics are the same.

The definitions of the `elem` and `notElem` functions which determine whether an element is or is not a member of a list, respectively, are defined using the functions `any` and `all` composed with the equality operators. `any` and `all` are defined by composing the `and` and `or` functions with the map operator. The `and` and `or` functions, in turn, are defined using the `foldr` higher order function and the logical-AND (`&&`) and logical-OR (`||`) operators. All these definitions are presented below, as they appear in `Prelude.hs`.

```
and, or   :: [Bool] -> Bool
and        = foldr (&&) True
or         = foldr (||) False

any, all  :: (a -> Bool) -> [a] -> Bool
any p      = or  . map p
all p      = and . map p

elem, notElem    :: Eq a => a -> [a] -> Bool
```

```
elem            = any . (==)
notElem         = all . (/=)
```

7 Advantages and Disadvantages of Haskell

Upto now, this paper has described only differences between Haskell and Miranda without attempting to weigh the advantages of one functional programming language with the other. This section will provide some benefits and potential problems of the `hugs` Haskell interpreter. Any decision to introduce `hugs` to a teaching environment should consider the following advantages and disadvantages. Because the languages are so similar in syntax and semantics, most of the issues discussed in this section are non-technical in nature.

7.1 Good Points

The license under which `hugs` is distributed is very liberal in relation to `mira`'s license. `hugs` is essentially free software and can be redistributed with very minor restrictions. (see Technical Report 2000–03 for the actual license under which `hugs` is distributed). In particular, note that, unlike `mira`, `hugs` can be installed on as many machines as desired. Students can even take it home to install on their own machines (but see Technical Report 2000–03 for more information). There are no known public domain or free versions of Miranda available.

The compilation and installation of `hugs` is very simple and straightforward. A `configure` script is supplied with the distribution which allows the interpreter to be built on a wide variety of Unix-like operating systems and hardware architectures. As shown in Technical Report 2000–03, using the interpreter is very similar to using Miranda's environment and is easy to learn. Because the source code is freely available, any problems encountered with the interpreter can be corrected by anyone familiar with the code. Free source code availability also encourages the programming community at large to correct and redistribute fixes independent of a single organization.

A final point is that the Haskell language resembles Miranda enough to make learning Haskell relatively easy for someone who is already familiarity with functional programming in general and Miranda in particular. The constructs and semantics are sufficiently similar to allow anyone familiar with one language to very quickly pick up the other language with little difficulty.

7.2 Bad Points

Despite the last point mentioned above, there are still enough differences and so-called *gotchas* that can make learning Haskell tricky for person already familiar with Miranda. Perhaps the most notable difference is the concept of the type class and the type hierarchy that pervades Haskell's type structure. This can be difficult to adjust to and would naturally require some practice. Also, some of the error messages generated by `hugs` can be very confusing and outright hostile to new and experienced users alike.

Miranda's interactive help facility (obtained by using the `/man` command) is more comprehensive than that which comes with `hugs` — indeed, Miranda's entire reference manual is accessible through this command. However, to offset this, there does appear to be much more documentation on the web for Haskell than Miranda. In addition, an informal review indicated that most of the newer books that teach functional programming languages use Haskell rather Miranda as the functional programming language of choice.

The Haskell implementation studied (`hugs`) is interpreted, not compiled. This would make pragmatic use of the interpreter somewhat limited. However for course work and assignments done in an academic environment, where the emphasis is on teaching the foundations of functional programming, this should not pose a problem. The performance of `hugs` in relation to Miranda is worthy of note. As shown in Appendix A, `hugs` used much more resources (both time and space) than Miranda when evaluating the same functions. This may pose a problem if several people are using the same machine to run Haskell scripts. Should a decision be made to evaluate Haskell for an actual course, the system administrators may wish to install it on a non-critical and sufficiently powerful machine just in case the requirements are excessive.

8 Conclusions

As can be seen from this paper, Haskell supports a considerable number of features found in Miranda. In fact, it appears that the two languages are more similar than they are different.

However, this paper could not possibly enumerate *all* the differences between the two languages — other differences not documented here do exist. For example, Technical Report 2000–03 demonstrates some of the differences in the standard library functions. Other differences may also occur is some of the less commonly used (but not necessarily less important) features of the language. Hopefully, this paper was able to address most of the major differences between the two languages' features that would be covered in an introductory course on functional programming. More advanced features, such as monads and lambda abstractions that are supported by Haskell in general (and `hugs` in particular) were not discussed in this paper. The support for these features demonstrates further differences between Miranda and Haskell.

Naturally, before any decision is made to use Haskell as a language for functional programming, further practical research should be undertaken. The `hugs` interpreter should be evaluated perhaps by an undergraduate student as part of a summer work term. Of particular interest would be to determine if a student with no functional programming background at all could navigate the `hugs` Haskell implementation enough to write some elementary programs.

References

[1] `http://www.haskell.org/aboutHaskell.html`
Gives a very good overview and introduction of functional programming and Haskell. This link provides the strengths and weaknesses of the functional programming paradigm as it relates to imperative languages such as C and Java.

[2] `http://www.haskell.org/implementations.html`
Provides more information on various Haskell implementations as well as their relative advantages and disadvantages.

[3] `http://www.haskell.org/onlinereport/`
Gives the formal Haskell language specification. This document contains information useful to those who are building Haskell compilers and interpreters. It is a useful document for reference and clarification purposes.

[4] `http://www.haskell.org/tutorial/`
This page contains a document called *A Gentle Introduction to Haskell* which serves as a tutorial for people new to Haskell. It is a fairly easy read if you are already familiar with functional programming, but can be overwhelming to others.

[5] `http://www.numeric-quest.com/haskell/hcompanion/index.html`
The Haskell Companion. A comprehensive site which attempts "to bridge the gap between experts-developers and new users".

[6] Bird, Richard, *Introduction to Functional Programming using Haskell (2nd edition)*, Prentice Hall, 1998.

[7] Turner, David, *An Overview of Miranda*, MUN Printing Services, Winter 1993.

A Relative Performance and Resource Usage

This appendix presents an evaluation of the relative performance of Miranda (version 2.025 last revised 22 March 1997) and the `hugs` (February 2000 release) implementation of Haskell. In an effort to enable a fair comparison between the two, a version of hugs was compiled for `liz` — the machine running the Computer Science Department's licensed version of Miranda. This will allow the evaluation to be independent of the underlying hardware and operating system.

For the execution of each script, the following metrics were measured:

- The number of reductions required to solve the problem. A reduction is basically the replacement of the left-hand-side of and equation with the righ-hand-side for an equation (this is commonly known as a rewrite rule) or it can be the evaluation of an operator (such as the arithmetic operators). The number of reductions that occur in a Miranda script can be counted by giving the `/count` command in Miranda environment. In `hugs` the number of reductions can be displayed by giving the `:set +s` command.
- The average amount of time required to run the same script. For the purposes of these tests, each script was run five times and the average amount of time required to run each of the scripts was determined. The `/count` command in Miranda gives the amount of time for a function to finish evaluating. To obtain a similar metric for `hugs`, a special configuration option `--enable-timer` was specified during compilation of the interpreter. This causes the interpreter to display the amount of time taken when the interpreter is given a command or a function to evaluate.
- The amount of memory cells consumed during the script evaluation. Both `hugs` and Miranda have the concept of *cells* which are units of so-called *heap* memory which is used by both interpreters for storage purposes. The `/count` command in Miranda and the `:set +s` command in `hugs` will display the number of cells required for the evaluation of a particular function. In both environments, the heap memory was set to 250000 cells for the purposes of these tests.

Two scripts, each requiring moderate amounts of processing power, were implemented using both functional languages. The first script is an implementation of the quicksort algorithm. It takes a list as input and generates a list with all the elements in order as output. The second script is an implementation of the eight-queens problem. The input to the function is the number of columns (typically eight, which was used in the test case) and the output is a list of lists indicating the row position of each of the queens in each column. The code for both scripts are based implementations found in *An Overview of Miranda* [7].

At this point, it must be emphasized that the following empirical results may be somewhat crude. The results should only be used as a rough guide regarding the relative performance of the two languages. By no means do these results constitute a definitive measure of the overall performance of the two languages. Also, these tests are intended to compare the relative internal processing abilities of the languages and not how fast they can show lists on a display. In order to reduce the possibility of I/O latency perturbing the measurements, the scripts display the only length of the list resulting from the evaluation of the function and not the entire resultant list.

A.1 Sorting

The Miranda and Haskell versions of the quicksort algorithm are presented in Figure 2. Note that with the exception of the two semi-colons in the Miranda list comprehension being replaced with commas in the Haskell list comprehension, the two implementations are syntactically equivalent.

Figure 2: Quicksort Algorithm

Miranda:

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

Haskell:

```
qsort []    = []
qsort (a:x) = qsort [b | b <- x , b <= a ]
              ++ [a] ++
              qsort [b | b <- x , b > a ]
```

A.1.1 Results

In order to compare the performance of the quicksort algorithm, both `mira` and `hugs` were given the same list consisting of 10,000 integers to sort. The `hugs` interpreter took a considerable amount of time to parse the script (which contained the list holding the numbers to sort). Miranda seemed able to parse and process this list much faster and essentially required a negligible amount of time to compile the script.[6] Table 3 shows the average amount of time required to parse the script, the amount of time to evaluate the quicksort function on the list and the total time. The number of reductions, the reductions per second during evaluation and the number of cells required during processing are also included.

Table 3: Quicksort results

Language	Avg. Compile Time (sec)	Avg. Eval. Time (sec)	Avg. Total Time (sec)	Number of Reductions	Reductions per second	Cells Used
Miranda	–	2.82	2.82	1,294,200	458,936	1,451,192
Haskell	39.36	9.21	48.57	3,015,776	327,588	3,981,287

As can be seen from the table, `mira` performed much better than `hugs` in all metrics. Not taking into account the considerable amount of time required for `hugs` to parse the script, the evaluation of

[6]The `.x` intermediate file generated by Miranda was intentionally deleted each time `mira` was started up so as to force Miranda to recompile the script from start each time the sorting script was loaded.

the quicksort function still took more than three times longer in `hugs` than it did in Miranda. Taking into account compilation time, `hugs` took 17 times longer to perform the same task. The number of reductions performed by `mira` was less than half the number of reductions required by `hugs` to sort the same list. The number of memory cells required by `mira` was nearly one-third that required by the `hugs` implementation. In every case, `mira` handily outperformed `hugs` in this test in terms of time and resources.

Curiously, the `hugs` interpreter appears to cache the results of each evaluation. If the sort was attempted again in the same `hugs` session it required only 6 reductions and consumed only 8 cells of heap space. Miranda does not appear to cache results of earlier evaluations.

A.2 Eight-Queens Problem

The Miranda and Haskell solutions to the eight-queens problem are presented in Figure 3. Again, only minor syntactic difference between the two implementations exist, as indicated by the highlighted text. The algorithm in both cases is essentially the same.

Figure 3: Eight-Queens Problem

Miranda:

```
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
```

Haskell:

```
queens 0     = [[]]
queens (n+1) = [q:b | b <- queens n , q <- [0..7] , safe q b]
safe q b     = and [ not ( checks q b i ) | i <- [0.. length b - 1]]
checks q b i = q == b !! i || abs (q - b !! i) == i + 1
```

A.2.1 Results

The results from the evaluations of the eight-queens problem is presented in Table 3. The compilation times required for the script in Miranda and Haskell were negligible and are therefore not included in the table below.

As an aside, when the `queens.m` script was loaded and compiled (using the `/file` command in `mira`), attempts to run the script would repeatedly produce the error message `impossible tag (-9) in reduce`. Restarting `mira`, reloading the `queens.m` script (with the `queens.x` intermediate file generated from the last session) and rerunning it would then evaluate without problems. This appears to indicate a deficiency in the Miranda implementation.

Again, the `mira` implementation performed much better than the `hugs` implementation, requiring one-fifth the time and one-quarter as many memory cells to solve the eight queens problem. Less than one-third as many reductions were required by `mira` to perform what is essentially the same computation.

Table 4: Eight-Queens results

Language	Avg. Total Time (sec)	Number of Reductions	Reductions per second	Cells Used
Miranda	2.98	1,468,995	492,951	1,641,450
Haskell	14.862	5,090,939	342,547	6,674,108

A.3 A Final Note On Performance

The above numbers prompted a further study into simpler evaluations. For example, consider the following Haskell and Miranda sessions, respectively:

```
Prelude> 1+1
2
(17 reductions, 43 cells)

Miranda 1+1
2
||reductions = 2, cells claimed = 5, no of gc's = 0, cpu = 0.00
```

The `hugs` implementation appears to require nearly an order of magnitude more reductions and more cells to perform what should be a relatively simple operation. The following sessions further reinforce this observation:

```
Prelude> 1
1
(13 reductions, 39 cells)

Miranda 1
1
||reductions = 1, cells claimed = 4, no of gc's = 0, cpu = 0.00
```

The reason why `hugs` has such overhead is not immediately known (it may have something to do with the more powerful type system), but this may be worthy of further investigation.