Haskell — A Wild Ride

Sven Moritz Hallberg <pesco@gmx.de>

21st. Chaos Communication Congress
Berlin, December 27th–29th 2004

Abstract

The Haskell programming language is a comparatively young purely functional member of the Miranda family. Its main features include its strong type system, declarative style, concise syntax, and clean structure. A to my knowledge unique trait of Haskell, at least among programming languages "in practical use", is the actual absence of nonfunctional side effects in the core language. That is, in Haskell the statement of referential transparency is *true*.

Contrary to the frequently found notion of Haskell being a language of mostly "academic interest", it is in my opinion actually very suitable for writing practical programs. And, mind you, without sacrificing its academic assets at all...

This introduction will briefly sketch the basic concepts of Haskell and then open into a tour of some language highlights, in order to provide a motivating outlook. Also given is a set of practical exercises on the basic concepts with some tutorial instructions. They serve as the basis for a hands-on workshop to be held after the lecture at 21C3.

1 Safety Precautions

First of all, it must be stated clearly that all but the tutorial part of this introduction is *not* to be taken as a programming course. I am deliberately only sketching concepts, leaving out details. The intent is to teach only what's necessary to "get to the point" instead of fiddling out all the background theory. Don't hesitate to "read over" parts not entirely clear. Maybe you'll understand anyway. This is supposed to be a fun ride.

However, to gain acceptable probability levels of maintaining the audience's mental integrity, a few basics are in order.

Compiled or Interpreted?

Haskell can be both compiled or interpreted, the language does not prescribe it. For example, GHC is a compiler that generates fast native machine code. Hugs is an interpreter, written in C, for quick tasks, running on pretty much any platform. NHC is another compiler, written in Haskell, which is much more portable than GHC because it generates byte-code linked with a tiny run-time interpreter.

In the following, I will use the term *Haskell system*, or just *system* to mean whatever is interpreting the code, be it compiler, interpreter, or Human.

Basic Syntax and Program Structure

In general, a Haskell program is nothing but a sequence of declarations. For example,

```
hello    = "Guten Tag!"
goodbye  = "Auf Wiedersehen!"
```

define the names `hello` and `goodbye` to refer to the given strings. Notice that I avoid to call them variables. Their values cannot be changed at runtime. Declarations in Haskell are absolute. That is one reason why Haskell is called *purely* functional.

Function Definitions

To perform any meaningful computations, functions are defined like this:

```
foo x = x + 2
```

The function `foo` takes a single argument and adds `2` to it. Notice two simple facts:

- The function argument is seperated from the function name simply by whitespace. Parentheses are only used to specify binding precedence in Haskell, as in `2 * (5+1)`, for example.
- Operators, like `+`, are written in infix notation.

The most important detail, however, is that I did not declare any kind of type signature for `foo` (or `hello`, or `goodbye`). Didn't I mention Haskell being strongly-typed? Yes! In fact, `foo` is not at all untyped. The Haskell type system can *infer* its type. Because `+ 2` is applied to the argument `x`, it can deduce from the types of `+` and `2` that `x` must be a

number, and that the result will also be a number.[1] That's all statically determined at compile time.

Now, to apply `foo` to some value, just write the argument after the function name, as in `foo 5`, which evaluates to `7`.

Now, let's define a two-argument function.

```
bar x y = foo y + x
```

This function applies `foo` to its second argument (`y`) and adds the result to the first argument, `x`. Notice that `foo` is applied only to `y`, not `y + x`. Function application in Haskell binds tighter than any operator![2]

Procedures

With all functions being pure, how can a Haskell program accomplish anything involving, say, network or file I/O? For instance, how do you write a web-server in Haskell? Or a text editor? The answer is that it *is* possible to write imperative procedures in Haskell, but they are strictly seperated from the functional parts by the type system. Let me demonstrate: To produce an executable Haskell program, you define the special procedure `main`, like so:

```
main = do putStrLn hello
          input <- getLine
          putStrLn (show input ++ "?")
          putStrLn goodbye
```

Don't mind the details, but notice the keyword `do` in the above. It is very important. In effect, it tells the type system that "the following is a sequence of imperative commands". Now the type system knows, that main is not a function, but just what I said, a "sequence of imperative commands". The `main` procedure is executed exactly once, when the program starts, and there is only *one* way to execute other such sequences: calling them from `main`.

Also note the special syntax `input <- getLine` above. That is an imperative assignment. `getLine` is executed once, after the call to `putStrLn hello`, and in all the following commands, `input` refers to its result.

[1] By the way, I say "number" instead of `Int` because this function will also work for `Floats`, `Rationals`, or complex numbers, or any other number type for which addition is defined!

[2] Function application also associates to the left, so the expression `foo foo 1` does *not* mean `(foo (foo 1))`, but `((foo foo) 1)` — which makes no sense of course.

Types

As already mentioned, the type system is one of the most important parts of Haskell.[3] Every expression in a Haskell program is statically (i.e. at compile-time) assigned a fixed type. For example, the literal constants `'a'` and `"Hello"` have the types `Char` and `String`, respectively. Complex types are built by combining simpler types and all-new custom types can be defined by the user.

Type Declarations

In the examples above, no types where mentioned. So what are they good for? The type signatures are *optional*. In addition to defining the value of `hello` as above, I can also declare it's supposed type.

```
hello :: String
```

In this example, the type signature does not add much, because the type of `hello` is immediately clear from its value. In the case of (complicated) functions, however, the type annotation serves

1. as documentation for the programmer and
2. as a hint to the compiler, enabling it to spot type errors earlier, resulting in better error messages.

Lists

The most important complex data type in Haskell, much like in LISP, is that of the list. It is written as `[a]`, where `a` may be any type (you can have lists of lists of ...). For instance, the type "list of `Int`s" is written as `[Int]`. A list of lists of `Int`s has the type `[[Int]]`.

Literal lists are written like `[1,2,3]` or `['H','i']`. `[]` is the empty list.

Function Types

Obviously, being a functional language, Haskell is going to allow functions to be passed around as values. So, according to what I said above, they must be assigned types as well. A function type is of the form `a -> b` where `a` is the argument type and `b` is the result type. Multiple-argument functions have types of the form `a -> b -> c -> r` where `r` is the result type and the others are argument types. So, I can declare the following type signatures for the functions defined earlier:[4]

[3] For the curious: It's a so-called Hindly-Milner type system, with some extensions.

[4] The alert reader may notice that I claimed these functions would work for any number type. Indeed, their infered type is more general, and by declaring the given type sig-

```
foo :: Int -> Int
bar :: Int -> Int -> Int
```

List Constructors

Like in many other programming languages, values in Haskell are created by special functions called constructors. Conceptually, a constructor is just a function that takes some arguments (data fields) and returns the corresponding value of the respective type.

As the most important example, consider the list type `[a]`. It has two constructors: `[]` and `(:)`. `[]` takes no arguments and is the empty list. `(:)` is the cons operator (an infix constructor). It takes an element and a list, so `(x:xs)` is taken to mean the list with `x` as the first element, followed by the elements in `xs`.

NB: The syntax for list literals is only syntactic sugar for repeated application of `(:)`:

```
['H','i','!'] == 'H':('i':('!':[]))
```

2 Regular Attractions

Now you should be ready for the ride. This section will first show some of the more "mundane" features of Haskell, that make it, well, attractive. Then, in the next section, the *real*[5] fun begins...

Local Definitions

A very convenient feature when defining functions are local definitions, introduced by a *where clause*:

```
readability :: String -> Float
readability text =
        if n==0 then 1
                else 1 / fromIntegral n
    where
    n = length text
```

The `fromIntegral` function converts from `Int` to `Float`. Of course it would have been correct to substitute `length text` for `n` in both places, but that would decrease the code's readability![6]

Constructor Pattern Matching

So far I have mentioned how to construct lists, but not how to inspect them, let alone custom data types. Recall that any value in Haskell is created by application of a special constructor function to some arguments specific to that constructor. The constructor used and its arguments are remembered with the value. In fact, they *are* the value.

Inspection of values and access to their data fields (i.e. constructor arguments) is accomplished by *pattern matching* on the constructors. Most importantly, in a function definition, the arguments on the left side can be constructor patterns. For example, here is a function to test whether a given list is empty or not:

```
null []       = True
null (x:xs)   = False
```

In the second case, notice how the constructor's arguments are given names, `x` and `xs`. These names are bound as variables on the right side (not used here), providing access to the data fields. This double use of the pattern matching (testing and binding) has its share in making Haskell programs very concise.

User-Defined Data Types

One of the most prominent activities in programming is the definition of custom data types suitable for the problem at hand. In Haskell especially, custom data types not only help structure the program and make it more readable, they also leverage the type system to assist the programmer in writing correct programs.

To give an example, imagine you had a reader/writer for magnetic stripe-cards. Were you to control this device from a Haskell program, the following data type definitions might be convenient.

```
data Cmd =  Read   Track
         |  Write  Track

data Track = Track1 | Track2 | Track3
```

The definitions above introduce two new data types. `Cmd` has two constructors, `Read` and `Write`. Both taking one argument of type `Track`. The type `Track` in turn has three constructors which take no arguments.

Notice that types and constructors are written in upper-case. That is mandatory in Haskell for reasons soon to become clear.

Now suppose the control protocol for the card reader specified that in order to read a track, you send the character sequence `"aaa"` to the device

---

natures I am *restricting* them to the types specified. It is sometimes useful to do that, in order to catch type errors early.

[5] or *really evil*

[6] And `length text` would probably be evaluated twice in the usual case. To avoid duplicate evaluation of equal subexpressions the system would need to search expressions for them. Although possible, I don't think they do that.

for track 1, `"aab"` for track 2, and `"aac"` for track 3. The same holds for writing, only that the first character is `'p'` then. Using pattern matching, you can easily define the function mapping a `Cmd` to the corresponding control string.

```
ctlstr :: Cmd -> String
ctlstr (Read   Track1) = "aaa"
ctlstr (Read   Track2) = "aab"
ctlstr (Read   Track3) = "aac"
ctlstr (Write  Track1) = "paa"
ctlstr (Write  Track2) = "pab"
ctlstr (Write  Track2) = "pac"
```

Then, given an I/O routine `sendstr` which transmits a given string to the device, `sendcmd` is easily defined as

```
sendcmd cmd = sendstr (ctlstr cmd)
```

Given these definitions, you can interactively controll the device from your Haskell interpreter.

```
Main> sendcmd (Read Track1)
...stuff happens...
```

Next, one would extend `sendcmd` to a routine that also retrieves the result of the command, and so on. But the important thing here is that the newly-defined data type `Cmd` enables you to express the device commands in a structured and readable way. As long as this interface is used, it is also impossible to send erroneous command strings to the device. Last but not least, `Cmd`s can be processed like any other type of data, i.e. stored in lists or other structures, saved to files, and displayed in the interpreter.

Pattern Matching Again: Wildcard Matches

The constructor patterns above specified the values for all fields of the data. Of course, as with any decent pattern matching mechanism, the patterns can contain wildcards. Write `_` (underscore) for any data fields, and the pattern matches any value in that field.

```
firstcmdchar :: Cmd -> Char
firstcmdchar (Read _)   = 'a'
firstcmdchar (Write _)  = 'p'
```

To bind the field's contents to a variable, just put a lower-case identifier in its place, as already done in `null` earlier:

```
cmdtrack :: Cmd -> Track
cmdtrack (Read t)   = t
cmdtrack (Write t)  = t
```

By the way, such variable bindings in patterns are the reason why constructor names in Haskell are always in upper-case: to distinguish them from the variables.[7]

List Comprehensions

Because lists and list operations are so common in Haskell, there is a special syntax for generating lists by combining certain elements from other lists. Such expressions are called *list comprehensions* and they are similar to the set comprehension notation used in mathematics, e.g. $\{x^2 | x \in \mathbb{N}, x > 5\}$.

My favourite example is the following Haskell implementation of Quicksort:

```
qs []     = []
qs (x:xs) = qs [y | y<-xs, y<x] ++ [x] ++
            qs [y | y<-xs, y>=x]
```

Here, `++` is the list concatenation operator. The algorithm takes the first element, `x`, of the given list and filters the rest of the list for those elements less than `x` and those greater or equal to `x`. These two filters are expressed quite naturally as list comprehensions.

In general, an expression of the form

```
[f x y | x <- xs, y <- ys, p x y]
```

evaluates to a list whose elements are generated by applying `f` to all possible pairs of elements `x` and `y`, taken from the lists `xs` and `ys` respectively, for which `p x y` evaluates to true. This works for any number of lists.

3 Wipeout!

OK, you've gotten a glimpse of some cool features that let you at least implement Quicksort. Good!

But enough of the boring standard stuff we all know from Standard ML...[8]

2...1... Go!

The Hackers Must Have Slack!

Haskell is a lazy language. It only evaluates expressions when their value is needed (to be printed to the screen, for example). This makes it possible to define infinite data structures, infinite lists especially. Consider:

```
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

[7] The same argument applies to type names, to distinguish them from *type variables* in polymorphic types, like `a` in `[a]`.

[8] I've never used Standard ML.

That's an infinite list containing the entire sequence of Fibonacci numbers. As you can see, the definition is recursive, it mentions itself, `fibs`, on the right side. The function `zipWith` is a very handy higher-order function[9] that takes two lists and "zips" them into a new list by combining the elements at same indices with a given function. For instance:

```
zipWith (+) [1,2,3] [4,5,6]
== [1+4,2+5,3+6]
```

Now the trick with `fibs` above is that the first two elements of the list are given as a starting point and the rest is defined as `zipWith (+) fibs (tail fibs)`. Because of the way `zipWith` works, to generate its first result element, it only needs the first elements of its argument lists. These in the above are `fibs` and `tail fibs`. The first argument of `fibs` is fixed as `0`, and the first argument of `tail fibs` happens to be `1`, also given. So `zipWith` can generate the third element before looking further into its arguments. Then, because the arguments refer back to `fibs` itself in the definition, when `zipWith` asks for more elements, it gets back the values it has generated itself.

Who Uses RC4 Anyway?

I don't, but I still implemented it in Haskell once. I'll spare you the details, but if I recall correctly, it works roughly like this: From the given key, a big array of numbers, a so-called S-Box, is calculated. The algorithm then iterates through a loop that reads a certain element out of the S-Box, performs a bunch of swapping and other operations to transform the S-Box, and starts over. The resulting stream of numbers is xor'ed with the plain-text.

So imagine implementing this in C. You'd have a data structure for the S-Box, call an initialization routine on it with the key, and then generate the byte stream in little chunks. In Haskell, first of all, there is no reason for chunkyness. Given the S-Box, it's easiest to define the resulting byte stream as an infinite list. But also, there is no need for an initialization routine!

```
type Key    = String
type Byte   = Word8
data SBox   = ...

mksbox      :: Key -> SBox
keystream   :: SBox -> [Byte]

rc4  :: Key -> [Byte] -> [Byte]
rc4 k xs = zipWith xor xs
           (keystream (mksbox k))
```

What's the point, you say? There is none, until you know about partial application. Given the above, suppose you want to encrypt a bunch of files with the same key. You have the files' contens in `[Byte]` lists, as required for `rc4` above, and write

```
key = "deadbeef"

file1_encrypted = rc4 key file1
file2_encrypted = rc4 key file2
```

Instead of writing `rc4 key` every time, you can apply only the key argument to `rc4` once, which yields a new function you call `enc`. When applied to the byte stream, `enc` returns `rc4 key bytes`.

```
enc = rc4 key

file1_encrypted = enc file1
file2_encrypted = enc file2
```

This looks pretty and works, but there's one problem: If `rc4` is defined as given above, the S-Box and the key stream will be calculated seperately for every call to `enc`. The reason is that the Haskell system is not smart enough to see from the definition that the keystream only depends on the key. However, we can make that explicit.

```
rc4 k = \xs -> zipWith xor xs ks
    where
    ks = keystream (mksbox k)
```

Compare this definition to the original one. The expression `\xs -> ...` is a *lambda abstraction*[10]. The *value* of `\xs -> zipWith xor xs ks` is a *function* that, when applied to an argument `xs`, returns `zipWith xor xs ks`.

The new definition of `rc4` defines `rc4 k` to return that function. It is completely equivalent to `rc4 k xs = zipWith xor xs ks`. Only that this time, the keystream is defined in a where clause in the closure of `rc4 k`. Being attached to this "partial application" of `rc4`, calls to `enc = rc4 k` always refer back to it. In the first definition of `rc4`, the expression `(keystream (mksbox k))` lay within the closure of `rc4 k xs`, so its evaluation was only triggered once both arguments had been given.

[9] a function taking a function argument

[10] The backslash, \, symbolizes a λ.

Type Classes

Haskell supports compile-time polymorphism. For one thing, there are functions like `null` above. The type of `null` is `[a] -> Bool`, polymorphic in the list element type, `a` . Such type variables may be instantiated with absolutely any type. This is called *parametric polymorphism*. But there is more. Recall the functions `foo` and `bar`, which would actually work on any kind of number, instead of just `Int`s. `foo`'s type is of the form `a -> a` but here, `a` may not be any type, but must be restricted to numbers. That is accomplished with *type classes*. The most general type of `foo` as it would be infered by the system is:

```
(Num a) => a -> a
```

Read that as "`a -> a` under the constraint that `a` is a number".

`Num` is the class of types that support a certain set of operations specific to numbers. Its definition in the standard library looks (basically) like this:

```
class Num a where
    (+)     :: a -> a -> a
    (-)     :: a -> a -> a
    (*)     :: a -> a -> a
    negate  :: a -> a
```

There are some other operations but they are not important now.

So, type classes look much like, many say, interfaces in Java[11]. But there is an important difference. Java is only run-time-polymorphic. While calling class methods in Haskell also incurs a small run-time cost, the types are all resolved statically at compile-time. This eliminates the possibility of run-time type errors and alleviates the need for the corresponding run-time checks.

Now look at class `Num` again. If we assume associativity of `+`, `-`, and `*`, as well as commutativity of (+), `Num` is basically the algebraic class of *rings*. There is another type class in the standard, called `Fractional` that is a subclass of `Num` and adds division.

```
class (Num a) => Fractional a where
    (/)    :: a -> a -> a
    recip  :: a -> a
```

Under the assumption that `*` is also commutative, `Fractional` is basically the algebraic class of *fields*.

What's so cool about this? Nothing except for restricted parametric polymorphism in itself. Until you define the class of vector spaces.

```
class (Fractional a) => VS v a |v->a
    where
    -- vector add and subtract
    (^+^)  :: v -> v -> v
    (^-^)  :: v -> v -> v
    -- scalar multiplication
    (*^)   :: a -> v -> v
```

For example, one could declare the pairs of `Float`s to form a vector space over the scalar type `Float`[12], like so:

```
instance VS (Float,Float) Float where
    (x,y) ^+^ (a,b) = (x+a, y+b)
    (x,y) ^-^ (a,b) = (x-a, y-b)
    k     *^  (a,b) = (k*a, k*b)
```

OK, so I've not told you some things about the above. First of all, it's not Haskell 98[13]. But the required type system extensions are mild and well-supported by all major systems. In particular, `VS` is a *multi-parameter type class*. It is not simply a set of types but a binary relation on types. It establishes a relation between the vector types and their associated scalar fields.

Some of the vector space methods do not mention the scalar type. For instance, let `a` and `b` be two vectors of type `v`. Then, when writing `a^+^b`, there could be two class instances for `v`, differing only in the scalar type. The type system would not know which one to use. To resolve this, we can simply promise to the Haskell system not to do just that, i.e. to declare at most one vector space instance per vector type. That is precisely what the annotiation `|v->a` in the definition of `VS` means. It is called a *functional dependency* on the type relation `VS`.

Apart from those two bits, the definition of `VS` is covered by Haskell 98 and using it, you can write your linear algebra programs using a single set of vector operators and they will work for any specific implementation as long as you provide an appropriate `VS` instance.

4 Conclusion

As promised, I have only skimmed through some, as I think, interesting highlights of the vast topic that Haskell, as much as any sophisticated programming language, is. Maybe, however, it has shone through, that Haskell's sophistication is not bought with a bloat of language features. Extensions are accepted into the language only quite conservatively, so as to not roll it down the track of useless complexity.

[11] SHUDDER!

[12] Yes, to the mathematical purists, `Float` is not a field. But the only sensical option is to pretend it was.

[13] The current version of the standard, last revised in 2002

In the end, I hope this introduction was able faintly hint that Haskell programs can cleanly and naturally express things that might take much more sweat in the mainstream languages. Only to result in safer and more robust code on top of it.

For further exploration, pointers to all relevant tools and documentation for learning and using Haskell can be found at the central website:

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

5 Workout

Ex. 1: Implement a stand-alone Haskell program that prints the traditional mantra "Hello, World!" on standard output. You should be able to find all you need in the `main` example of section 1.

a) Compile the program with a compiler of your choice and run it as a stand-alone executable.

b) Load the program into an interpreter and run the main procedure from the prompt. Also try calling some basic I/O routines interactively at the prompt.

Hint: The customary extension for Haskell source files is `.hs` .

Ex. 2: Using pattern matching and recursion, implement a function that computes the sum of all numbers in a list.

a) What is the type of this function?

Hint: In Hugs or GHCi, type `:t exp` to find the type of any expression.

b) Load your implementation into an interpreter and interactively try it on some example inputs.

Hint: Type any expression at the interpreter prompt to evaluate it.

Ex. 3: Using pattern matching, recursion, and list construction, implement a function that increments all elements of a list of numbers by 1. That is, it should have the type `(Num a) => [a] -> [a]` .

Ex. 4: A function that takes a function as one of its arguments is called a *higher-order function*. Restricted to `Int`s, generalize the function from ex. 3 to apply any given function of type `Int -> Int` to all elements of a list of `Int`s.

a) Without asking the interpreter, what do you think the type of this function should be? Pay attention to correctly parenthesize the nested function type.

b) Can you generalize the function to work for any type of list, given a function of suitable type? What is the type then?

Hint: If you get stuck, ask the interpreter about the infered type of your original function.

Ex. 5: At the top of your program, put the lines

```
import Data.Bits
import Data.Word
```

to make the bit-manipulation modules available. Look up the `rotate` method of class `Bits` at:

```
http://www.haskell.org/ghc/docs/6.2.1/\
    html/libraries/base/Data.Bits.html
```

The type of an 8-bit byte is called `Word8` in Haskell. Implement a function to rotate every byte in a given list by 4 bits.

Hint: Use your solution to ex. 4b or `map` from the standard library, which is automatically imported.

Ex. 6: The I/O routine `putStr` prints a list of 8-bit latin-1 characters to stdout. You've already seen `getLine` in section 1, which reads one line of input from stdin. The procedure `getContents` reads the entire standard input byte-wise and returns it as a list of latin-1 characters.

The task is to implement a stand-alone program to apply the transformation from ex. 5 to its standard input. In particular:

- Read stdin,
- turn the characters back to their byte counterparts,
- apply the transformation from ex. 5,
- turn the bytes into characters again, and
- print the result to standard out.

To perform the character/byte conversions, import the module `Data.Char` and consider the following functions:

```
ord :: Char -> Int
chr :: Int -> Char
fromIntegral
    :: (Integral a, Num b) => a -> b
```

□