The Monad.Reader/Issue3/Notes on Learning Haskell

From HaskellWiki

< The Monad.Reader | Issue3

1 Learning Haskell Notes

By *Graham Klyne*

This page contains some personal notes accumulated over several months of learning to program in Haskell, developing a moderately-sized application.

Contents

- 1 Learning Haskell Notes
  - 1.1 Learning Materials
  - 1.2 Error messages from compilers
  - 1.3 Multi-parameter type classes
  - 1.4 Confusing introduction to monads
  - 1.5 Programming idioms
    - 1.5.1 Separate structure traversal logic from data specifics
  - 1.6 Functors
    - 1.6.1 Functors and Monads
  - 1.7 Using and composing monads
  - 1.8 Type classes, class constraints and data declarations
  - 1.9 Common errors in do sequences
  - 1.10 Type classes are not as useful as they first seem
  - 1.11 Existential types
  - 1.12 More on type class subtleties; class dependencies
  - 1.13 Using ShowS
    - 1.13.1 to turn a string into a `ShowS`, use `showString`
    - 1.13.2 to turn a `ShowS` into a string: simply apply it to an empty string
    - 1.13.3 to concatenate `ShowS` values, use function composition
  - 1.14 Use standard types as far as possible
  - 1.15 The power of sequence
  - 1.16 ((->) e) is a Monadic type
  - 1.17 Scanning a list
  - 1.18 Acknowledgments

1.1 Learning Materials

I found there was plenty of good tutorial material for introducing a new programmer to Haskell, and the language report was a very precise and concise reference for language specialists, but there seems to be little in between for a 'jobbing programmer'. The report could sometimes be unhelpful when trying to learn details of some subtle new feature of the language not covered by the tutorials.

A book in the style of, say, the *The Annotated C++ Reference Manual* (Ellis and Stroustrup) would be useful: when I was learning and using C++, this book was invaluable for me, as it contained just the right mix of formal definition and in-depth pedagogical material.

I find the Haskell report tends to concentrate largely on definition, and sometimes skimps on explanation. This would be fine, but the other, more tutorial, sources I consulted don't seem to have complete coverage of the language.

1.2 Error messages from compilers

Error messages can sometimes be very obscure, misleading even. Especially in the realm of types and classes.

The type system is very subtle. Maybe this is unavoidable. Better diagnostics would help; GHC and GHCi seem to be better in this respect than Hugs, in part because they give more information about the difference between what was expected and what was found.

I think that because so many concepts are separated, it's easy to focus on the wrong aspects of one's type declarations.

A common error is to mis-judge operator precedences and end up with a sub-expression being inferred to have a very different type from what was intended; usually a function type. (Hint: function application binds more tightly than any operator; function composition has very low precedence.) Here is a simple example:

```
f1 :: (Char->Float) -> (Float->Int) -> Char -> Int
f1 g h c = h . g c
```

results in the HUGS error message

```
... - Type error in application
*** Expression : h . g c
*** Term : g c
*** Type : Float
*** Does not match : a -> b
```

A valid form would be:

```
f1 :: (Char->Float) -> (Float->Int) -> Char -> Int
f1 g h c = (h . g) c
```

In this case, it's fairly easy to see the mistake, but in more complex expressions the type errors can be rather tricky to spot, especially when inferred intermediate types are being used. In the following example, the same error is less easy to determine from the offered diagnostic message `Term `f5`, type `Char -> a -> b` does not match `Char -> Int``:

```
f2 g h c = h . g c
f3 'a' = 1.0
f4 1.0 = 1
f5 :: Char -> Int
f5 c = f2 f3 f4 'a'
```

It can sometimes be difficult to distinguish between simple typos and deep abuse of the type system. In this example:

```
class (Eq k, Show k) => LookupEntryClass a k v where
 entryKey :: a -> k
 entryVal :: a -> v
 newEntry :: (k,v) -> a

 data (LookupEntryClass a k v) => LookupMap a k v = LookupMap [a k v]
```

the error message reported (by HUGS) was of an invalid type constructor in the 'data' declaration, when the real problem was missing type parameters in the class method declarations; i.e. should have been:

```
class (Eq k, Show k) => LookupEntryClass a k v where
 entryKey :: a k v -> k
 entryVal :: a k v -> v
 newEntry :: (k,v) -> a k v

 data (LookupEntryClass a k v) => LookupMap a k v = LookupMap [a k v]
```

1.3 Multi-parameter type classes

Although not part of the official Haskell 98 language, multi-parameter classes seem to be pretty much essential for many sizeable tasks or re-targetable software components. I think clearer signposting of this widely-implemented feature would help a new programmer. (But see also point [#type-class-misuse below] about using type classes,) I note others, too, have observed the importance of multi-parameter classes.

Also, more accessible documentation in this area would help. I think it could be made clearer that multiparameter classes are general relational constraints. I got locked into an incorrect reading that 'C a s' meant that 'a s' was an instance of 'C', rather than C is a relation on types, of which (a,s) is a member. It's not clear why I made this mistake -- I think the seeds of my confusion may have been laid by the form of subclass declarations:

```
(C a) => (D a)
```

which means `D` is a subclass of `C`, or `a` stands for an instance of `C` as well as `D`. In OO programming, classes are types and instances are values -- Haskell separates these ideas more carefully.

I don't know if this was just a personal confusion, or more widely experienced. I recognize that it's all explained, but it's difficult to take on all the new material at once without being lead by previous experience.

1.4 Confusing introduction to monads

Monads can be a source of confusion. The basic ideas are easy enough to get hold of, and using existing monad-based libraries is quite simple, but the deeper issues of when and when not to use monads are less well covered. I found little help given on how to design new monads. Maybe this is covered in the common language specification and/or tutorial materials, but if so it's not clear where to find the appropriate descriptions.

I found the paper *State in Haskell* by Simon Peyton-Jones and John Launchbury to be one of the more helpful sources in this respect. (The introductory approach of introducing IO first, then generalizing to state -- as in, say, *The Craft of Functional Programming* -- I found to be less helpful, as it didn't help me to understand how monadic values are created and activated. This is covered in later material, but by then some incorrect/incomplete views about how monads work had become entrenched.

(I had some ideas for writing a short Programmers' View Of Monads - if I can remember them! If I recall correctly, my idea was to focus on concrete monads, describing the monadic behavior of Maybe, and then moving up to lists, state, and eventually IO.)

1.5 Programming idioms

I suspect that even though I am now able to write Haskell programs of moderate complexity, I am not really using the functional idioms as effectively as I might. E.g. I have read claims that functional programming allows for better modularization of code, but I've found my own attempts to modularize seem to run into difficulties. I'm probably trying to apply conventional techniques inappropriately.

I could use a good source of information about effective programming style and idioms in Haskell.

Later, I read John Hughes' paper *Why Functional Programming Matters*, and *Scrap Your Boilerplate* by Ralf Laemmer and Simon Peyton-Jones, which have helped me to some insights about how one might strive to use functional programming more effectively.

Later still, it seems to me that a key to effective use of Haskell is designing things so that each function is concerned with just one aspect of a computation: data structure traversal, simple data value manipulations, computational strategy, etc., with all other aspects of a computation dealt with by separate function parameters. Thus, a complex computation can be expressed as a combination of primitive single-purpose functions,

each of which defines some single part, and can be re-used in a different context. One way of seeing this might be to consider Haskell as a pattern language: I think many, if not most, design patterns commonly found in design pattern catalogs can be expressed in some way has higher order functions.

(All this is easier said than practised, and I find most of my own code falls well short of this ideal.)

1.5.1 Separate structure traversal logic from data specifics

A particular example of the preceding exhortation comes in separating structure traversal from the specifics of processing the structure members.

Haskell's polymorphic type system allows very generic structure handling routines to be written. The list functions in the Haskell prelude and library are examples of this. The handling of structure-related issues can be kept separate from data-specific issues by using function parameters to handle data-specific functions. The Haskell list libraries have

```
processBy
```

functions that abstract element matching tests in this way.

I have found that failing to separate traversal logic in this way tends to lead to over-complicated code. Separating traversal logic means that it can be tested on simpler data structures, and of course means that it can be re-used with different data. But even when the traversal function is so arbitrary as to defy re-use, the code simplification is a big win.

(This advice is carried a good deal further by the functional strategic programming project (Strafunski); see also the 'Scrap Your Boilerplate' paper . See [1] (http://www.ninebynine.org/Links.html#Programming-Haskell) for links).

1.6 Functors

In my early days of Haskell programming, I kept on bumping into this idea of a 'functor' without any real motivation or sense of what it was used for. Then suddenly it seemed to click...

A 'functor' is a structured collection (or container) type with a method (

```
fmap
```

) that accepts

a method and applies that method to the members of the collection yielding an isomorphic collection of values of a (possibly) new type. This separates data structure traversal from operations on members of the structure, which is a simple example of the separation of concerns mentioned in the section [#programming-idioms above] on

programming idioms.

It has been pointed out to me that this characterization of Functor doesn't obviously embrace many useful Functors which are rather not collections.

For example, every monad (like IO) can be a Functor if you take

```
fmap = Monad.liftM
```

.

(For common monadic functors like `[]` and `Maybe` this would give the same operation as their normal definitions.) For myself, if I squint, I can see most monads as a kind of container or carrier or arrangement of other data values, so while I accept the comment it doesn't completely invalidate my early intuition about functors.

If the true nature of a functor has been intuitively and clearly explained, it doesn't seem to be anywhere that I've come across yet.

1.6.1 Functors and Monads

(This sub-section deals with some matters arising in discussion of the original *Haskell Learning Notes*, and probably does not contain any useful information for new students of Haskell.)

Let's look more closely at the comment that every Monad can be a Functor:

```
liftM :: Monad m => (a->b) -> m a -> m b
fmap :: Functor f => (a->b) -> f a -> f b
```

The function signatures are clearly comparable. Consider `Maybe`:

```
liftM f a = do { a' <- a ; return f a' }
 = a >>= \a' -> return f a'
 = case a of
 Nothing -> Nothing
 Just a' -> Just (f a')
fmap f Nothing = Nothing
fmap f (Just a') = Just (f a')
```

so it's easy to see these are equivalent for `Maybe`.

Looking at the pattern here, it seems that a Monad is a extension of a Functor, which also deals with the 'no value' case (Nothing, error, empty list, or whatever is returned by the monad's fail function).

It has been claimed that every monad is a functor, but not the other way around; a monad is a functor PLUS functions >>= and return satisfying some laws. Similarly, every functor is a type constructor but not vice versa; a functor is a type constructor PLUS a function fmap satisfying some laws. (*So what are the Functor laws?* I can think of maybe `fmap id = id` when applied to a Functor)

The claim that every monad is a functor is not without controversy.

I think this is because there are other ways of defining `fmap` than

```
fmap = Monad.liftM
```

which result in equally valid functors. (It has been said: "The fact Haskell does not make Functor a superclass of Monad is an infelicity of the Standard. In Haskell 1.4 (I think), it was in fact a superclass.")

Strictly speaking monads need not support fail functions. The Id monad has no 'no value' case, for example.

Rather than using `liftM`, `fmap` can be defined via the monad operations:

```
fmap f m = do { x <- m; return (f x) }
```

or, more succinctly:

```
fmap f = (>>= return . f)
```

But you cannot define `>>=` and `return` via `fmap` alone. So, in this sense, a monad is a special kind of functor. (Note that this definition of `fmap` is the same as the standard monadic definition of `liftM`.)

The term 'monad' is used above rather loosely. 'monad' is a term from category theory (whose precise definition is quite abstrusely mathematical), from which the Haskell Monad type class is derived. In Haskell, a Monad is a *type constructor*: it accepts a type value and returns a new type. That is, a Monad has *kind* `* -> *`. If `m` is a Monad and `a` is a type, then `m a` is a *monadic type*. An instance of a monadic type is a *monadic value*.

1.7 Using and composing monads

Reading the available material, I found this really confusing. Not because it's difficult, but because the patterns used are so different from what I have been used to in conventional programming. The particular area that I was having problems getting my head around was the understanding how a monadic value (i.e. an instance of a monadic type) is created, and how final values are extracted and used. Many of the examples I found appeared to gloss over these issues, especially if some result was to be used in a non-monadic expression. I think a cookbook approach to use of composed monads, with emphasis on actually how to actually execute the monads and extract a final result, might be helpful.

Eventually, I found Mark Carroll's example at [2] (http://web.archive.org/web/20070706123846/http://www.haskell.org/hawiki/MonadState) to be really useful. I spent a little while adding comments to this example, and everything became much clearer.

1.8 Type classes, class constraints and data declarations

Try to avoid putting class constraints on data declarations.

I found myself doing this when declaring a datatype intended to be an instance of a particular class, and then finding the type system would not allow me to make it a member of some other class, such as Functor, because the constraint was not justifiable by the method signature for that class.

I think the desire to do this is an artefact caused by OO-style thinking, along the lines of 'an instance of this class must be subject to the class constraints'.

The data type used as an instance of a class need not always be subject to the class constraints (and, it seems, usually should not). The data type may be used additionally for non-class purposes, and this is assumed when it is also used as an instance of some other class, like Functor. It's only when the specific class methods are used that the constraints come into play, and need to be declared.

Thus, rather than adding class constraints to a data type, they should be part of the type signature of any function or class that depends on the class membership, e.g. by using functions associated with the class signature.

1.9 Common errors in do sequences

An error I frequently make, which leads to really obscure error messages, is:

```
return f a
```

in a do-sequence, instead of:

```
return $ f a
```

or:

```
return (f a)
```

I feel that compilers might helpfully notice this and offer a hint. I can't offhand think of any situation when the former case is actually sensible (though I'm told there are a few).

I've also noticed that, because a String is a list, and a list is a Monad, it is sometimes possible to completely omit the 'return' statement when returning a string value without this being an immediately invalid do sequence. Instead, the return type of the do sequence is an unexpected value (a list monad containing a Char, or something like that), which in turn leads to very confusing secondary type error messages.

1.10 Type classes are not as useful as they first seem

Or, maybe more correctly, there are many better ways to achieve the same things that an OO programmer would do using classes (or, in Java, interfaces).

I have encountered situations in which I declare a class, and instances of that class, and other functions and structures that use the class, and later I find it really doesn't support what I want to do because one cannot mix different instances of the class in situations where a single common datatype is needed. E.g., a list of class instances must have values of the same instance type in every position. In this kind of situation, storing instances of a class is often the wrong thing to do, unless it is genuinely desired that every value has the same constrained but unspecified data type.

Commonly, a more effective way is to use polymorphic function types: if the putative class has just one interface method, consider declaring a function type instead, and using that in place of a class-constrained type. If there are several methods, declare an algebraic data type whose members are polymorphic functions. (This latter amounts to almost exactly how late-binding class methods are implemented for many OO programming languages.)

The time to use a class seems to be when there is a genuine desire to instantiate a particular interface over different and specific underlying types of data. A class doesn't make sense as a structuring or organizing mechanism. For example, it is common to want to format values of very different types, and to write functions that use this generic capability. But it doesn't really make sense to organize data types into those that can be formatted and those that cannot -- they may have no other attributes in common.

Some tutorial material on using algebraic data types for 'OO-style' programming tasks might be helpful. (There's a useful message from John Meacham here: [3] (http://www.haskell.org//pipermail/haskell/2004-June/014164.html) .)

A weakness of over-use of the type class system has been pointed out by Frank Atanassow in the context

of formatting (the

```
Show
```

class):

> "I do my best to avoid type classes, but I'm in the minority. In the long run, I think it's better to use existentials, as they let you define multiple instances for the same type. This usually makes more sense. For example, 'Show': why support only one, global way of printing things? I often need multiple ones."

I too have noticed that -- I often seem to want both a simple conversion to text, and a variation which formats over several lines with indentation. Still, classes like Show that provide common methods over a range of types are often useful.

In general, I think beginners overuse type classes, and use them incorrectly. I think a good approach is to write a first version without any fancy type hacking, just using the usual ADT approach, and later come back to it and see how to generalize it with classes, if and when the need arises. People tend to forget that the major difference between ADTs and OO-style classes is really only that with a class you can have many type instances in the same program simultaneously, whereas with an ADT you can have only one; but the ADT implementation is still interchangeable. Usually, that is all you need. Also, the single instance of an ADT can still be polymorphic, as long as the variable types are opaque to operations on the ADT itself.

1.11 Existential types

It seems that

```
forall x
```

introduces x as existential when it appears immediately preceding a

datatype constructor declaration. Elsewhere, it appears to signal a universal quantifier.

Existential types are a kind of information hiding mechanism: some datatype is used in an implementation, but its details are hidden from view, and the particular type used may vary from instance-to-instance in the same surrounding type context (e.g. between members of a list). Having the existential type being an instance of a class may provide means for some functions to do useful things with the type.

See also: discussion of issues on Haskell-cafe mailing list, about October 2003, starting with [4] (http://www.haskell.org//pipermail/haskell-cafe/2003-October/005231.html) .

The reason is that, in:

```
data AppT a = forall x. App (x -> a, x)
```

by considering the constructor's role as a function, we see that `App` must be an instance of type:

```
(forall x. (x -> a, x)) -> AppT a
```

(note the `forall` is under the second arrow) which is *implied by*:

```
exist x. ((x -> a, x) -> AppT a
```

where the `exist` is at top-level. With rank-N polymorphism you can construct such things without data declarations. (A forall induces an existential when it occurs in an `odd' argument position.)

(The above explanation was a response to my original notes, which I now find I don't entirely understand. Hopefully, a more comprehensible explanation will arise in due course.)

1.12 More on type class subtleties; class dependencies

I think I finally figured out how to get my LookupMap abstraction to work the way I wanted. The idea was that I would be able to use LookupMap to provide a lookup over any datatype by creating an appropriate LookupEntryClass instance declaration. Originally, I had:

```
class (Eq k, Show k) => LookupEntryClass a k v
 where
 newEntry :: (k,v) -> a k v
 keyVal :: a k v -> (k,v)</pre>
```

and

```
instance (Eq k, Show k) => LookupEntryClass (,) k v where
 newEntry = id
 keyVal = id
```

and

```
data LookupMap a k v = LookupMap [a k v]
```

(A list is a poor implementation structure for this purpose. The idea of this class is that I can vary the implementation while maintaining a consistent interface for my applications.)

The trouble with this is that I couldn't use an existing type as a lookup entry unless it had a type constructor that was of the prescribed form; i.e. (typename keytype valuetype) in this case. I finally realized, after 6 months or so of using Haskell, that I could declare the lookup entry class thus:

```
class (Eq k, Show k) => LookupEntryClass a k v | a -> k, a -> v
 where
 newEntry :: (k,v) -> a
 keyVal :: a -> (k,v)
```

and

```
instance (Eq k, Show k) => LookupEntryClass (k,v) k v where
 newEntry = id
 keyVal = id
```

and

```
data LookupMap a = LookupMap [a]
```

That is, the `LookupEntryClass` constructor takes three parameters of kind `*`, rather than the first being of kind `* -> * -> *`. In this way, the relationship between the component types and the lookup entry type is hidden. The downside of this is that the system can no longer infer that if, say, `a k v` is a `LookupEntryClass` then so is `a v k`. There is also some other type information that was inferred automatically, but which now must be stated explicitly, though, in practice, not so much.

When needed, the key and value types are made available through a context declaration, thus:

```
keyOrder :: (LookupEntryClass a k v, Ord k) => a -> a -> Ordering
keyOrder e1 e2 = compare k1 k2
 where
 (k1,_) = keyVal e1
 (k2,_) = keyVal e2
```

I don't think this overall approach would be possible without using the type class dependency extension to Haskell.

1.13 Using ShowS

`ShowS` is a type that is used to improve the efficiency of creating long strings (character sequences) that are for output, or are otherwise used in left-to-right order.

The problem addressed by `ShowS` is that string concatenation is O(n) in the length of its first string argument: this means that when building a long string by successive concatenation, the time required to append each character is proportional to the length of the string, making the overall time required to build the string proportional to the square of its length.

Type `ShowS` is defined as:

```
type ShowS = String -> String
```

At first glance, `ShowS` does not appear to rectify this, since for a string value it is defined simply as `(++)`:

```
showString :: String -> ShowS
showString = (++)
```

`ShowS` uses lazy evaluation to defer evaluation of the `++` operation; if the string is accessed strictly sequentially, the `++` never needs to be explicitly evaluated.

Consider the expression:

```
a ++ b
```

In order to access the first character of the resulting string, it is necessary to evaluate 'a'. If 'a' itself is a concatenation of strings:

```
(a1 ++ a2) ++ b
```

then that concatenation too must be evaluated.

By representing an unterminated string as a function, the evaluation of concatenation can never be required until the string is closed by applying the `ShowS` function to en empty string (or any string).

The above example then would be:

```
showString a1 (showString a2 (showString b ""))
```

Thus, the introduction of a function allows the left-associated construction of concatenations to be evaluated in a right-associated fashion. To get at the start of a sequence, it if not necessary to evaluate anything that comes after it.

So how should one use `ShowS` values? There are three key operations:

1.13.1 to turn a string into a `ShowS`, use `showString`

```
a = showString "Hello"
```

or just a section:

```
a = ("Hello"++)
```

The Haskell prelude and library provide functions that turn other values into ShowS: the default mechanism is to turn them into a string, then apply showString as above. The default showsPrec is defined as:

```
showsPrec _ x s = show x ++ s
```

which might equally be written as:

```
showsPrec _ x = ((show x)++)
```

or:

```
showsPrec _ x = showString (show x)
```

- When implementing complex data structures as instances of Show, consider implementing showsPrec instead of show

1.13.2 to turn a `ShowS` into a string: simply apply it to an empty string

```
a = (\x->"Hello "++(" world"++x)) ""
 = "Hello"++(" world"++"")
 = "Hello"++(" world")
 = "Hello world"
```

1.13.3 to concatenate `ShowS` values, use function composition

```
(a . (b . c)) x = a ((b . c) x)
 = a (b (c x))
 ((a . b) . c) x = (a . b) (c x)
 = a (b (c x))
```

```
So if:
```

```
a = ("Hello,"++)
 b = (" cruel"++)
 c = (" world"++)
```

then:

```
(a . b . c) "" = a (b (c ""))
 = a (b " world")
 = a " cruel world"
 = "Hello, cruel world"
```

1.14 Use standard types as far as possible

There is extensive support for doing useful things with the standard types declared in the prelude.

For example, it may be tempting to define a new type to represent a particular value or absence of a value. I did this for my RDF language tags, defining:

```
data Language = Lang String | NoLanguage
```

It would have been far better to use:

```
type Language = Maybe String
```

Many of the built-in types are defined as Monads or Functors, either in the prelude or in supporting libraries. While not obvious to someone new to Haskell, these additional definitions provide a rich seam of added functionality, ready to be accessed as the techniques associated with higher order functions become appreciated. For example, consider `sequence`...

1.15 The power of sequence

The prelude function `sequence` is surprisingly versatile.

The basic type signature is:

```
sequence :: Monad m => [m a] -> m [a]
```

Its function varies with the particular Monad (m) with which it is used:

- list: sequence has the effect of computing a Cartesian product
- `Maybe`: sequence returns Nothing if any of the list members is Nothing, otherwise Just a list of bare values.
- Error monad: returns the first error in the list, or a list of values if there are no errors
- State monad: returns the effect of applying each transformation in turn.
- IO: (also function sequence_) returns a new IO value that is the result of executing each of the statements in turn. One particular use for this is to turn a list of IO values, each returning a value of some type, into a single IO value that returns a list of values of that type. (An IO value being an instance of some type constructed using the IO monad.)

Sometimes, sequence can be applied more than once to a value, with each application having quite different effects as inner monadic values are exposed.

There is a generalization of `sequence` that re-arranges a more arbitrary monadic structure, as in:

```
foo :: (Monad m1, Monad m2) => m1 m2 a -> m2 m1 a
```

There is much work on composing monads. A function like this is usually called 'swap'. These two papers address this topic:

- http://www.cse.ogi.edu/%7empj/pubs/springschool.html
- http://www.cse.ogi.edu/%7empj/pubs/composing.html

(I understand there is another that addresses this topic more directly, but the reference is not currently available.)

1.16 ((->) e) is a Monadic type

The type `((->) e)` can be defined as the simplest(?) instance of a Reader Monad, which can be used to evaluate several functions with the same 'environment' value. (Note that `(->)` is the function type constructor, having kind `(* -> * -> *)` -- see Haskell Report section 4.1.2.

1.17 Scanning a list

A common requirement seems to be to scan through a list of values, returning a computation based on the first member of the list that yields a value for that computation. In imperative programming, this might be achieved by looping to probe each member of the list, and breaking out of the loop when a satisfactory element is found. If the entire list is scanned without finding a satisfactory element, some different or default result is returned.

In Haskell, a pattern for achieving this relies upon lazy evaluation: suppose the required computation is `processItem`, returning `Just` the desired result, or `Nothing` if the item does not yield a desired result. Then, the following code might be used:

```
listToMaybe $ catMaybes $ map processItem list
```

This returns `Just` the required value, or `Nothing` if no member of `list` yields such a value. Lazy evaluation means that only elements of `list` up to that which returns the desired value are actually processed. Alternatively, to return an empty list if there is no matching value, otherwise a singleton, use:

```
take 1 $ catMaybes $ map processItem list
```

This idea can be applied to other traversable structures, such as trees, etc.

1.18 Acknowledgments

Many of the lessons recorded here are due to careful and patient explanations by many members of the Haskell developer community, and denizens of the mailing lists listed at http://www.haskell.org/mailinglist.html. My thanks are given to all, too many to enumerate, who have helped me along the way. Included are specific contributions from Frank Atanassow and Andrew Pimlott. Any errors or misunderstandings propagated above are, of course, solely my own.

-- Graham Klyne (DateTime: 2005-05-11T15:18:48Z)