Haskell Sucks!

Paul Bone

You asked for it

Lyndon:	So how do you feel about Haskell?
Paul:	I have feelings.
Lyndon:	Good or bad?
Paul:	It depends.
Lyndon:	Do you want to give a talk about why Haskell sucks?
Paul:	Ummm, okay.

It's not my fault if you don't like what I have to say. One of your own asked for this.

But..

there are some things preventing me from making the claim that **Haskell Sucks!**

- "*X* sucks" is not a useful criticism
- "90% of everything sucks" - Theodore Sturgeon (writer).
- My criticism of Haskell is specific, rather than general
- I'm on good terms with the Simons.
- Haskell really is **pretty damn good**.

Haskell is pretty damn good but has some critical problems

Paul Bone

Who am I to *dis* Haskell anyway?

I am *not* a Haskell expert.

I have been working on and in pure declarative languages for 8 years. My Ph.D. was awarded for my work on auto-parallelisation of Mercury programs.

I have worked on:

- The Mercury implementation (in Mercury and C), A pure logic/functional language.
- ThreadScope, a visualisation tool for GHC parallel execution (in Haskell).
- I tutored declarative programming classes (Mostly Haskell).

I like Haskell

Haskell is a **pure functional** programming language. I **love** **pure functional** programming! It has:

- Decent performance
- Good library support
- Support for parallelism and concurrency
- Books and other resources
- A strong community

So what's my problem with Haskell?

I have two main issues with Haskell:

- **Monads**
- **Lazy evaluation**

I will also discuss some more general problems that I believe affect Monads, Laziness and Haskell generally.

Remember that everything **sucks** but different things just suck **differently**.

What are Monads?

> *A monad is just a monoid in the category of endofunctors, what's the problem?*
>
> — Phil Wadler

What are Monads?

> ~~*A monad is just a monoid in the category of endofunctors, what's the problem?*~~
>
> ~~— Phil Wadler~~

According to "The Internet" this is fictionally and satirically attributed by Wadler by James Iry (http://tinyurl.com/so-monoid-is).

Maybe it is based on:

> *All told, a monad in X is just a monoid in the category of endofunctors of X, with product × replaced by composition of endofunctors and unit set by the identity endofunctor.*
>
> — Saunders Mac Lane from "Categories for the Working Mathematician"

Anyway...

it doesn't matter who said it, it matters that it's funny. And it's funny because there is truth to the joke:

Monads are an unfamiliar abstract concept that many people struggle with.

People **struggle**, and yet we are overwhelmingly bad at providing good explanations and learning material.

So, back to that question. **What is a monad?**.

Monads are...

There are many monad tutorials, including many of the form:

Monads are [like]...

Space Suits, **Toxic Waste**, **Burritos**.

This is a **symptom** of the problem with Monads: they're a very abstract thing, and difficult to learn.

> **Brent Yorgey**: Abstraction, intuition, and the “monad tutorial fallacy” http://tinyurl.com/monad-tut-fallacy

Monads are...

M is a monad such that:

> bind :: $\forall\, a,b \cdot M\ a \to (a \to M\ b) \to M\ b$
> return :: $\forall\, a \cdot a \to M\ a$

Simple as that!

This only makes sense if you *already* know what a monad is.

Think back to when you finally **got** monads and understood this, how did it **feel**?

It felt **good** you probably even felt **proud** of yourself. You feel good because you solved a **difficult problem**.

Thoughts: Maybe this is no worse than learning **recursion** or **pointers**.

Using monads

Monads are **extremely useful**, they're used for many things in Haskell:

- Controlling side-effects,
- Handling errors,
- Managing state,
- Reading and writing streams,
- Nondeterministic search,
- Anything else!

Monads are so useful, that I have a favorite: `List` can be used with lazy evaluation to perform non-deterministic search!

Do you have a favorite?

Using monads

The `IO` monad makes it convenient to sequence IO operations.

```
putStr "Hello " >>= (\_ -> putStrLn "world!")
```

Of course it looks better with **do notation**

```
do putStr "Hello "
   putStrLn "world!"
```

Using monads

How about error handling

```
foo :: Either String a
foo = case computeSomething of
        Left x  -> case computeSomethingElse x of
            Left y  -> computeAnotherThing x y
            Right e -> Right e
        Right e -> Right e
```

The `Either` monad provides simple error handling.

```
foo :: Either String a
foo = do x <- computeSomething
         y <- computeSomethingElse x
         computeAnotherThing x y
```

Using monads

The `Data.Binary.Get` assists with reading binary data.

```
readEvent :: Get Event
readEvent = do time_stamp <- getWord64be
               id <- getWord32be
               payload <- readEventData
               return $ Event time_stamp id payload
```

Using monads

We can use the `Writer` monad to create a logging facility

```
logMsg :: String -> Writer [Entry] ()
logMsg m = tell [m]

something :: Writer [Entry] Something
something = do a <- ...
               logMsg "Computing something"
               b <- ...
               return $ makeResult a b
```

Example from `wiki.haskell.org`.

Combining monads

Monads are not naturally **composable** because one type constructor must contain all the returned information.

The only way to combine monads is with **monad transformers**, which are themselves monads. We can combine `Get` and `Writer` to handle errors while reading binary data.

```
readEvDat :: WriterT [String] Get (Maybe EventData)
readEvDat = do type_ <- lift getWord16be
               case type_ of
                 0 -> lift readStartEventData
                 1 -> lift readStopEventData
                 _ -> do hoist generalize
                           (logMsg ("Unknown ev...
                         return $ Nothing
```

Combining monads

```
readEvDat :: WriterT [String] Get (Maybe EventData)
readEvDat = do type_ <- lift getWord16be
               case type_ of
                 0 -> lift readStartEventData
                 1 -> lift readStopEventData
                 _ -> do hoist generalize
                           (logMsg ("Unknown ev...
                         return $ Nothing
```

Operations on the `Get` monad must be **`lifted`** into that monad. And a **monad morphism** is used to convert from the `Writer` monad to the `WriterT` monad.

Usually programmers will write wrapper functions to do this, nevertheless it's boilerplate code and work that could have been avoided.

Combining monads

```haskell
readEvDat :: WriterT [String] Get (Maybe EventData)
readEvDat = do type_ <- lift getWord16be
               case type_ of
                 0 -> lift readStartEventData
                 1 -> lift readStopEventData
                 _ -> do hoist generalize
                           (logMsg ("Unknown ev...
                         return $ Nothing
```

The monads form a *stack* and now precedence of the monads affects which version of >>= is called. This can alter flow control in surprising ways.

Monad transformers are a kludge for the lack of composability for monads.

Monads: summary

- Monads are an unfamiliar abstract concept.
- Monads are not explained well.
- Monads are a significant barrier to Haskell adoption.
- Monads do not compose.
- Monad stacks do not solve the composition problem.
- Monads and other abstractions can hide relevant details.

Thoughts: People have a tendency to over-complicate things, then they reward themselves for solving difficult problems rather than avoiding them.

- Don't **work around** a problem that can be **solved**.
- Don't **solve** a problem that can be **avoided**.

Monads: alternatives

- Make monads available, but not required
- Don't over-use monads (or any other feature)
- Maybe provide other means of managing state, such as uniqueness types.

Lazy evaluation

In some ways **lazy evaluation** is the elephant in the room, with monads getting all the attention.

We mostly don't notice it, until we do something like use a larger data set.

`foldl` or `foldr`?

`foldl` or `foldr`

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

In a *strict* language `foldr` is not tail recursive because its higher order value is strict, and must be evaluated after the recursive call has returned but before the current call can return. `foldr` is **always** less efficient.

In Haskell the higher order value is **sometimes** *lazy*. When it is lazy it returns immediately without evaluating either argument. **Sometimes** `foldr` is more efficient as it can even avoid evaluating the tail of the list.

`foldl` or `foldr`

```
foldl f z []     = z
foldl f z (x:xs) = let z' = f z x
                   in foldl f z' xs
```

In a *strict* language `foldl` is tail recursive because it's final call is the recursive call — that's all there is to it! `foldl` is **always** more efficient.

In Haskell the context of the call to `foldl` is **sometimes** *lazy*. When it is lazy it constructs a long chain of *thunks* in memory. **Sometimes** `foldl` is more efficient (in a strict context).

So what if the context of the call is lazy and `f` is strict?

`foldl'`

```
foldl' f z []     = z
foldl' f z (x:xs) = let z' = f z x
                    in seq z' (foldl' f z' xs)
```

`seq` introduces strictness to make `foldl` efficient.

But `foldl'` is the most efficient **only** when `f` is also strict.

When to use `foldl`, `foldr` or `foldl'` is difficult for a **seasoned** Haskell developer to know. It is **impossible** for a **novice**!

More generally. Lazy evaluation makes it difficult to reason about performance.

Parallelism

Parallelism is expressed with `par`. Which creates a *spark* to evaluate its first argument to **Weak Head Normal Form** (WHNF) and returns the second argument.

```
par :: a -> b -> b
```

If the first argument represents a much larger computation, such as a lazy list, only the first part of that may be evaluated. This will not create enough work to make parallelism worth while.

Parallelism

To improve the amount of parallel work available, strictness must be used. The `pseq` function will evaluate it's first argument to WHNF **before** returning the second argument.

```
pseq :: a -> b -> b
```

But a lot of code would need to be annotated in this way. Plus, any annotated code is now always strict, even when parallelism **is not** used.

Parallel strategies

Parallel strategies were introduced as a means to allow developers to control parallel and sequential evaluation.

```
parMap :: (a -> b) -> [a] -> [b]
parMap f xs = map f xs `using` parList rseq

parList :: Strategy a -> Strategy [a]
parList strat = evalList (rparWith strat)

evalList :: Strategy a -> Strategy [a]
evalList strat []     = return []
evalList strat (x:xs) = do
  x'  <- strat x
  xs' <- evalList strat xs
  return (x':xs')
```

`Control.DeepSeq`

`Control.DeepSeq` provides the `NFData` type class that may be used to fully evaluate values (to *Normal Form*).

```
deepseq :: NFData a => a -> b -> b
```

Developers must implement NFData for each of their own types that they'd like to fully evaluate using `deepseq`

Each of these things: `pseq`, strategies and `deepseq`, **work around** problems with lazy evaluation.

Lazy evaluation: summary

- How do you know if some operation is lazy or strict?
- Developers must choose the appropriate algorithm depending on their data and the call's context
- It is difficult to reason about performance with laziness
- Parallelism and laziness interact poorly
- Stack traces are not available (fixed yet?)
- Lazy IO can also cause *surprises*

What's the alternative?

- Strict by default
- Opt-in laziness.

Mercury

Mercury is a pure logic / functional language. It has strong types, modes and determinisms.

`http://mercurylang.org`

Mercury also **sucks**.

Plasma

Plasma is very new functional language project, it is mostly vapourware and therefore also **sucks**.

`http://plasmalang.org`

YesLogic & Prince

I work for YesLogic.

`http://yeslogic.com`

Our Prince product typesets PDFs from HTML+CSS.
These slides are created with Prince!

`http://princexml.com`