Bat Snacks <https://nyeogmi.com/>

It's dark! Stay close, food.

Assessing Haskell

art by **KrAzOn89 < https://twitter.com/KrAzOn89>**

Most college-educated programmers know a few different languages before they enter the workforce. At my school we learned Java and C, and we were expected to pick up some Python on the side because it made things easier. Meanwhile, I see a lot of self-taught people learning Lua, thanks to Roblox.

I think people who only know a few programming languages tend to look at a new language like the key to a lock. They start to learn it because they need to know it in order to work on whatever project they have to do.

On the other hand, people who use a lot of programming languages usually start to look at programming languages as software products — which is what they are, especially from the point of view of their developers. They're able to compare them on their merits, rather than defaulting to whichever language solves the particular problem they're stuck on now, and they can evaluate those merits without relying on the claims of an outside authority.

Those of you who have hung out in novice communities will notice that a lot of beginners are preoccupied with the idea of "graduating" to C++. It's rare for them to know C++ especially well — their reasons for liking it seem to be based on claims they're not qualified to evaluate (often about its performance compared to other languages, or its level of use in industry) or implicit emotional appeals made by others, usually also novices.

From what I can tell, the C++ community doesn't see this as a big problem. (At least, I've never seen a C++ programmer complain about this tendency, and I've seen a lot of C++ programmers spread fear and uncertainty about close competitors, especially Java.)

I think this kind of community exists for several other technologies — Java programmers do this and so do PHP programmers.

A lot of people pitch Haskell to beginners by saying "it eliminates whole classes of errors at compile-time."

Well, what's that mean? Some people bristle because it sounds like an obvious lie — or at least a case of salesmanship. Other people believe it. It turns out the statement is partially true.

Haskell can panic just the same as Java does, but it doesn't panic on a NullPointerException — only on nonexhaustive pattern matches, which in practice are a generalization of NullPointerExceptions.

Based on my pretty complete Haskell knowledge, I think it's safe to say that Haskell's feature is less likely to generate errors in actual use. I also think the claim is not literally true — Haskell has pattern match errors at runtime — and that even if it were literally true, beginners are completely unqualified to evaluate it.

To clarify, I'm not saying that beginners are stupid. My opinion is that when your main impetus to learn a language is that it allows you to use a technology you want — the "key" model that applies to a lot of novices — then you're likely to know one language for each category of development you do, and you're not likely to understand it that deeply because you don't have a lot of things to compare it to.

So, I think it's implausible that the claim is being read literally. I also think the claim that Haskell eliminates whole classes of error contains an implicit emotional appeal. You're meant to hear "whole classes of error" and think "that sounds cool!" even if you don't know what it means.

When you talk to a novice who's heard of Haskell, they're likely to say that it's more difficult than other languages, it's weirder, it's more complicated. They might be saying it negatively or they might be saying it positively.

Generally, these are all conclusions that you'd normally draw by looking at some underlying facts about the language. Some are statements of opinion — meaning that you *can't* legitimately arrive at them without underlying facts to make a decision about. Others are conclusions it would be better to arrive at by assessing multiple Haskell features at once, meaning that, again, you'd need those underlying facts.

There's usually little evidence that those novices understand those facts.

There's a common feature between all the technologies where I've accused their communities of making emotional appeals to novices.

If I'm charitable, then the common feature is that all of them — Java, C++, PHP — encourage a pretty narrow usage model. Most make some strong assumptions out the gate and tell you that if you stick to them, it's going to make your software a lot better in some dimension.

That's certainly true of Haskell, too!

Those technologies I listed are usually half-right about their usage model having advantages — PHP's statelessness is a good model for software that needs to scale up; C++'s memory management discipline can be used to write especially fast software; Java-style OOP is a pretty good model for a lot of tasks.

They're half-wrong, because in each case there are significant disadvantages, often located outside the things the pitch is about, which the language doesn't adequately mitigate — and that's where the headache is.

If I were less charitable about those languages, I'd probably say something else, which is that all three of them are legacy. Most of them are not a

particularly successful take on the model they claim to be an example of — most of them have been superseded by later technologies that took the good parts and accentuated them.

If I were even less charitable, I'd say this: I think all three technologies are pretty bad. Most people I see praising Java, C++, or PHP only do so with reservation — or they do so because their chosen technology fits like a key into a lock for a technology they do want to work with.

I could be wrong about this, but I think I'm right. It seems to me like a little bit of manipulation might be one good way to keep a legacy technology alive — and if you've seen someone evangelizing C++ in a space for new game devs, then you know what I'm talking about.

Unfortunately, I think that it's possible the Haskell community is engaging in similar behavior. I frankly think not enough people are asking the question of why, for instance, monads as a pattern for controlling access to resources are a good idea, and I think a lot of beginners are expected to assume there's wisdom behind the decision when that's not evident.

I'm going to make an effort to assess Haskell fairly but without presupposing that any of its decisions are correct in isolation. Some of the things I criticize might be valid reactions to existing problems and misfeatures — but I'm assessing them as they present themselves now, with indifference to their history.

I considered trying to assess Haskell without mentioning my own opinions at all. I don't think that's fair or possible, nor do I think that would lead people to an accurate assessment of Haskell. I basically think that novices who aren't used to working in the product framing can still work inside it if they're given pretty comprehensive and relevant information.

I think it's hard to be objective about this issue in the sense that what you say is something no one can object to. I don't think you can show, to everyone's satisfaction, that a product is good or bad.

I do think you can demonstrate the sort of facts that lead people to conclude a language is good or bad, then explain your conclusions based on those facts,

then let other people draw their conclusions now that you've outlined the things people might object to.

I also think most people who evangelize Haskell are already committed to the idea that programming language fans have a roughly shared set of standards. When you sell your language mostly based on comparisons, you're making appeals to people who, you assume, have some standards in common with you.

You're doing this even if the comparisons aren't meant to be literally understood by your audience — like the "whole classes of errors" claim — because if there wasn't at least an imaginary audience that your comparisons made sense for, you wouldn't be making them.

Based on what I think is the set of assumptions I share with beginners and Haskell evangelists, I'm going to try to assess how Haskell measures up in a variety of domains that most people see as relevant when they assess a programming language. I've picked these based on what people seem to assess when they try a new language, and also based on what Haskell sees as a selling point.

My hope is that if you read this article, you'll come to the conclusion that several of Haskell's decisions seem to have major drawbacks without obvious advantages.

Importantly, there are still a lot of things about Haskell that I like and I think you'd be entitled to promote it even if you agreed with all the claims I'm about to make. But I'm hoping you'll come out of it with a strong inclination to treat encumbrances as encumbrances, if not outright flaws!

One other note: when I refer to a Haskell feature, I'm going to try to explain what it is before I criticize it. Beginners might still benefit from reading a few chapters of **Learn You A Haskell < http://learnyouahaskell.com/>** so they can follow my examples. (It's not the most popular Haskell guide any more, but I'm recommending it because it's short.)

Safety

Haskell users tend to praise the language for allowing them to write reliable code, and for allowing them to precisely specify what input they expect and what output they produce.

Some Haskell programmers call the language “safe” — by this they mean that it is hard to write code that has unexpected consequences.

Garbage collection

Haskell is garbage-collected — you don’t need to manually free memory, or track which object owns a value.

(I personally think garbage collection is a basic usability feature that most languages should seek to include unless they’re trying to target a hard-realtime use case.)

Error handling

Haskell has mandatory error handling in some cases.

It accomplishes this using its type system. Suppose `a` is the type of a value. In that case, `Maybe a` is the type of a value that could be `Nothing`. (Haskell’s version of `null`) `Either String a` is the type of a value that could be an error message instead. (written `Left "error"` or `Right result` respectively)

Haskell enforces that you handle errors by converting `Maybe a` values back to `a`s before doing `a`-specific operations on them. It also supports null-coalescing — taking a `Maybe a` and doing an operation resulting in a `Maybe b` will result in a `Maybe (Maybe b)`, which can be converted back to a `Maybe b`.

Testing

Haskell code is usually easy to unit test because Haskell programs have no access to their environment except through their function arguments.

However, Haskell code that hasn’t been written to support testing can (for instance) refuse to be run without a network connection, meaning that once access to its environment is granted, it needs its whole environment to be configured in a similar way to the conditions it needs under production.

This is similar to the problem that other languages solve with mock subclasses — Haskell usually solves it with type parameters, where the type parameter for a service client is replaced with the type of a mock for that service client. This pattern can mean adding an extra type parameter for every mockable component, which will often bubble up to calling code.

Haskell programmers are big fans of property-based testing through the library QuickCheck, which I think is just a great idea.

Danger zones

Haskell's compile-time safety features are (in my opinion) fairly well-designed, but I think it has some eye-popping behavior at runtime.

Note that any code in Haskell can crash the program by throwing an exception. This kind of fault is not checked-for at compile time. You can also loop forever, which will not crash the program but will certainly break it.

(Technically, throwing an exception just unwinds the stack, but if you don't handle it, the thread or the program will crash.)

Note that `bracket` (Haskell's equivalent of `try/finally`) does not necessarily run your `finally` block for non-main threads on program exit. Haskell unceremoniously terminates all threads except the main thread without raising an exception.

Haskell contains several functions that do IO operations at an unspecified time — for instance, this program is not guaranteed to read the file before writing it:

```
main = do
  writeFile "test.txt" "Hello, bats!"
  bats <- readFile "test.txt"
  writeFile "test.txt" "Hello, Nyeogmi!"
  putStrLn bats -- might say "Hello, Nyeogmi!"
```

(There is an alternative library of IO functions that do not exhibit this behavior, but it's bad that these are the default ones.)

Terseness

Haskell has a reputation for being terse.

It has a few features (mostly syntax-level) that enable this:

- You can define functions in the middle of a line: saying `f x = 1 + x` is the same as saying `f = \x -> 1 + x`.
- Functions are values. For instance, you can say `f x = 1 + x`, then say `g = f`, then use `g` as if it were `f`. You can put functions into a list: `[\x -> x + 1, \x -> x + 2]` and then manipulate the list.
- It has type inference: if you write `x = "abc"`, you typically don't have to specify `x :: Text`.
- Shorthand syntax exists for certain ways of turning functions into values. Instead of writing `\x -> x * 2` ("x times 2"), you can write `(*2)`. This is called "operator sections."
- It has builtin functions `read` and `show` that turn values into text or back from text, which allows you to save temporary results.
- It has implicit "currying" — if you write `f x y = x + y`, then `f 1` is equivalent to `\y -> 1 + y`. This has similar advantages to operator sections
- It has an operator for composing functions. For instance, `putStrLn . show` is equivalent to `\x -> putStrLn (show x)`

It makes some choices that are cumbersome. For one thing, updating a field of a record is verbose: `updateName newName x = x { name = newName }`

I also think its notation for code that does input and output is fairly verbose, but that's a topic I plan to introduce later.

Naming and the standard library

A lot of the functions built into Haskell have short, non-descriptive names, such as `ap`, `pure`, `nub`, and `return`. This seems to have an advantage for productivity, but may have disadvantages for readability.

Earlier I said Haskell has an operator for composing functions. Actually, Haskell has multiple operators to compose functions which are not compatible — for instance, `putStrLn . show` is valid, but `putStrLn . getLine` is not (it must be written `putStrLn =<< getLine`) — likewise, `return . return` and `return <=< return` have very different meaning.

Note on user-written code

What I've conveyed above is that the standard library of Haskell provides a lot of functions that operate on functions. What I'd like to add to that is that most user-written Haskell code actually *defines* a lot of functions that operate on other functions.

This style can be very concise, because it allows taking a pattern that appears in existing code, wherever it is, and replacing it with a function that implements this pattern.

Performance

Haskell tends to achieve C-adjacent performance in Benchmarks Game-style activities.

However, its optimizer achieves a lot of performance gains through application of **rewrite rules < https://downloads.haskell.org/~ghc/7.0.3/docs/html/users_guide/rewrite-rules.html>** . Across module boundaries, Haskell will sometimes struggle to rewrite temporary data structures out of existence. In general, **whole-program optimization is a weak point < https://github.com/ghc-proposals/ghc-proposals/pull/313#issuecomment-590143835>** .

Haskell's garbage collector suffers from the same periodic pauses that are typical of most garbage-collected languages. These pauses may cause frame drops in realtime game development but are unlikely to be a big deal for other program categories.

Out of all that — it appears to perform much better than the average scripting language, but I would not expect it to perform at close to the same speed as Java or C++ without profiling.

Data structures

Haskell's primary data structure is the linked list.

By default, Haskell strings are represented as linked lists of characters, but this causes performance problems so most users switch to using an array-

based representation instead. (such as `Text`) Code that is compatible with one representation is not automatically compatible with other representations.

Haskell has some built-in data structures to do state management with — most are not very efficient. `WriterT`, which exists for logging, can have terrible performance in some cases, and so can `StateT`.

Most Haskell data structures are "persistent," meaning that modifying the data structure results in a new data structure with no changes to the old one. There are some fast implementations of ephemeral data structures (the kind you are probably used to) but they usually require `IO` to use — that is, the type signatures of their methods are unusually complicated.

Laziness

Haskell is "lazy" — that means that it typically only evaluates your code when it needs your return value.

This means that Haskell code may appear to produce a result more quickly than the result is actually calculated, and that your client code can cause a function you call to produce results much more slowly than it's supposed to.

For instance, if a function produces a binary tree, and you explore one path to the bottom, then you might see logistic time complexity — if you explore the tree from left to right, you might see linear time.

When Haskell does not evaluate a value, it produces an unevaluated value called a thunk. (which is a pointer to the code that would generate the result)

Laziness is a good way to skip work, but if the value's going to be created anyway, it can have a performance cost. Most Haskell programs, to be fast, will opt out of laziness in at least some cases.

Many builtin Haskell data types come in strict and lazy varieties — usually with the same names. These varieties are usually incompatible from the point of view of library code that has to operate on them, so most libraries silently pick the data structure that they think will guarantee the best performance to outside users.

Libraries

Like every other language in the world, Haskell comes with a standard library which covers input and output — it also comes with a large user-created package repository called Hackage, and two popular distributions of compatible packages called Stackage and Haskell Platform.

The reason that multiple package distributions are common for Haskell is that Haskell packages have a reputation for getting into complicated versioning situations.

(I personally don't understand the reasons for this reputation and think the problem is overstated, but it's such a consistent complaint I feel the need to acknowledge it. I've experienced this problem with dependencies that provide development tools, but nothing else.)

Idiom

The Haskell community likes math a lot.

Many types and libraries in Haskell are named after mathematical objects, even if the connection is somewhat tenuous. (For instance, Haskell's `Functor` typeclass corresponds to something more specific than the mathematical notion of a functor — a covariant functor.)

Beginners are usually expected to learn to use the names of mathematical objects when explaining their code.

Haskell programmers tend to write very golfy code with a lot of symbolic names. Many people use point-free style, where functions' arguments are not named or written out, and instead code is written as a composition of shorter functions.

There are a variety of utility functions designed for use with Haskell's IO and error-handling types which are technically allowed to operate on functions, with surprising results:

```
f = join (+)
-- equivalent to
f x = x + x
```

Mastering these tricks is a mark of pride in the community.

Module system

In Haskell, record fields are in the module namespace, which means two types in the same module usually can't have the same field name. This means that field names in Haskell are usually two words:

```
data Person = Person { personName :: Text }
data Dog = Dog { dogName :: Text }
```

Your code is written in files called modules. If module A uses module B, module B usually cannot use module A.

Functions in Haskell cannot be associated with a type. (they are in the module namespace) In the standard library, some functions are prefixed or suffixed with an extra character (ex `mapM` and `map`) specifically to avoid problems caused by needing to provide the same function for two types.

It's pretty common for users to have to come up with unusual names for functions belonging to their own types, since the obvious names tend to exist in library code.

Tooling

Building

Haskell has two build systems: Cabal and Stack.

Stack depends on Cabal.

Stack has an enormous number of GitHub issues, most of which do not appear to be responded to. Over four attempts in the past three years, I have never been able to get Stack to work on Windows.

From what I can tell, Stack is the more popular build system, and I don't know why.

Cabal produces boring, statically-linked binaries which will run anywhere. (that is, anywhere compatible with boring, statically-linked binaries for the platform

in question)

Debugging

Tracing is awkward in Haskell. Its builtin logging tool, WriterT, can have a significant negative performance impact because instead of logging right away, it dumps the logs to a data structure which, ideally, would be optimized out but which, in practice, will not be.

There's a second tracing tool in Haskell called `Debug.Trace`, but it traces in the order that work is done — which is to say, basically random order, because Haskell does not make guarantees about evaluation order except the guarantee that it will evaluate your code before you need its result.

The built-in Haskell debugger works step by step in basically-arbitrary order, same as the builtin tracer.

A minor note — Haskell doesn't have a New Relic client, which would have been useful to me at work, although it does have a third party Datadog client.

Customizable control flow (Monad)

Haskell has a syntax feature called do notation, along with a corresponding library-level feature called Monad, which together allow Haskell to provide customizable control flow and imperative-looking programs.

Here's a sample program in do notation:

```
main = do
  beerBottles 10

beerBottles 0 =
  putStrLn "I don't know a lot about beer bottles or songs
actually"

beerBottles x = do
  putStrLn "I don't know a lot about beer."
  print x
  putStrLn "I'm really sad."
  putStrLn "Sorry, Beavis."
  beerBottles (x - 1)
```

There are ten million Monad explanations on the internet, so I'm not writing one here.

There's a little bit of snark in this section, which might bother some readers — I think that this is a case where Haskell's design is pretty difficult to present without judgment, because it has so many immediately apparent problems.

Usage

Monad allows people to write programs that, line-by-line, have similar structure to programs in ordinary scripting languages. Unlike in normal Haskell (where evaluation order doesn't matter), Monad can force evaluation to take place in a specific order.

These programs do not have access to C-style `for` or `while` loops. They have access to `foreach`-style loops through the `forM` family of functions, but those loops don't allow you to set variables for future iterations, nor is it possible to `break` early or `continue` from the next iteration.

Even in do notation, Haskell does not provide mutable variables. However, its built-in `IO` type provides `IORef`s, which act like objects that gate access to a single mutable field. Haskell also provides `STRef`s, which do exactly the same thing but are completely incompatible in every way.

Monad allows users to handle errors using the previously-mentioned `Maybe` and `Either` types. To do this, users need to use the `<-` statement (in do notation) or the equivalent `>>=` function. (provided by the Monad typeclass) This is also needed whenever users do `IO`. If users want to do IO and handle errors at the same time, they need to use the `ExceptT String IO` type instead, for some reason.

That type is provided in a separate library that is not installed by default. Because IO operations are defined for the `IO` type and not for the `ExceptT String IO` type, using `ExceptT` requires users to prefix every IO operation with the function `liftIO`.

There are a variety of libraries designed to avoid this inconvenience, called "effect systems." For instance, you can avoid this by installing `polysemy` or `mtl`, which will figure out how to convert the type you were given into an `IO` value that you can operate on directly. (Note that polysemy requires ten language extensions and two compiler flags, and it is not recommended to use it without its GHC plugin and compile-time code generation.)

Monad also allows Haskell users to express `LL(*)` parsers tersely. There are several libraries that do that.

Danger zones

Because Haskell implements Monad for `(->) a`, you will get a cryptic error message if you mistype the number of arguments for a function. (in plain english; an `IO` value will support the Monad interface, but so will a function that *would* produce an `IO` value if called)

Haskell's `MonadFail` typeclass has an extra `fail` method that will swallow some failures normally caused by pattern matching, which may be surprising to some people who expect pattern matching failures to result in a runtime error. It's even implemented for `Maybe`, where the behavior is to suppress the error and return `Nothing`. (aka null)

You can use `do` notation with types that do not instantiate `Monad`, without getting an error message. (The error you get will be on an invalid invocation of `>>` or `>>=`, two of the functions from the Monad interface, if an error happens at all.)

You'll receive a compile-time type error if you use the wrong function to operate on a monadic value — if you have an `IO Text`, printing it will require you to call `>>=` to get the Text out, or use the `(<-)` statement, and failing to do so will result in a fairly clear error message.

Conclusions

I think Haskell suffers from a variety of major problems.

Ignoring its purpose as a research tool and describing it only as a programming language, its apparent niche is this: it's a pretty fast garbage-collected language with a lot of safety features and very terse syntax. In my opinion, it lacks serious competition in this category.

Some languages that target a similar niche to Haskell are C#, Ruby, Rust, and Scala, but each of those languages has misfeatures that disqualify it as a clear competitor:

- C#: Microsoft; lacks sum types; until recently, limited null safety
- Ruby: has no static type system; is very slow
- Rust: has a relatively complicated type system; no garbage collector; verbose; pretty crappy support for iterators
- Scala: has a relatively complicated type system; has a preponderance of fascists; sbt sucks

(I'm not including Kotlin, because while it competes with Scala, it lacks true pattern matching and doesn't have traits, implicits, or typeclasses.)

I would argue that the reason Haskell lacks serious competition in this category is that its two goals are contradictory. Goal one is to make it as easy as possible to write code in shorthand; two is to increase maintainability. Even Haskell's standard library leans hard into short names and point-free style. It can genuinely be very hard to read other people's Haskell code.

I racked my brain for a while for serious contenders to Haskell's category and — not repeating Scala, which was on the previous list and might otherwise be your best bet — this is what I thought of:

- the OCaml family (OCaml, F#, and ReasonML)

- Purescript (which I had a much easier time getting to work on Windows, relative to Haskell)

I'm open to additional recommendations, especially because I can't recommend Haskell in good conscience based on its other problems. Please note — I'm only looking for things that are suitable for production use, so I don't intend to add hobby projects or Elm to the list.

Even though Haskell seems to be king of its niche, there are a few things I want to single out for additional scorn. I put this at the bottom so people who don't like rants can skip them — I think my points are substantiated OK, but I'm a lot angrier while I'm making them.

The tool situation

After a while, you just get tired of seeing WONTFIX.

I don't really understand how it happened, or particularly care — but every time Stack or GHC fell over for me and died, it was caused by a known bug.

From what I can tell, this is a consequence of Stack (and occasionally Haskell Platform) developers being aggressive about pinning to a particular version, while also not testing on Windows. However, there's a lot of bug reports on Linux that are very similar to what I experienced. I don't know how to explain this other than apathy.

There's never really been a time when the experience I had with IDE-like tools in Haskell was comparable to what I've had in Rust, even though Rust until recently had a smaller market share than Haskell. I don't know what it is that prevented the basic "invoke compiler, get error message" flow from working well.

I do know that the vast majority of Haskell development tools have, historically, been pinned to highly specific versions of other development tools. I frankly don't know if it has ever been possible to install them all at once. I'm sure somebody must have tried it.

Monad

In my opinion, Monad is perceived as mysterious to learners because at basically every turn, Haskell exposes it in a way that is useless to them:

- There are several major misfeatures that hurt readability and maintenance without a clear selling point. (The worst is `Monad ((->) a)`, which seems redundant given that `Reader` exists and is equivalent. Allowing `do` with non-`Monad` types also seems problematic, as well as rewriting into `>>=` and `>>` without first checking that each line has a matching type for the overall structure.)
- There are no effects systems that aren't significantly more bureaucratic than whatever scripting languages were already doing.
- Users are required to use extra utility functions to operate on values in `Monad` wrappers, compared to ordinary values.

In addition, Monad seems to have little advantage over other tools to relegate access. There are basically two subcategories of monads:

- Monads that affect the number of exit points code takes. (ContT, ExceptT)
- Monads that do not. (IO, Reader)

In case one, there are a few major patterns outside of FP-world which the bulk of imperative languages can now support: coroutines, exceptions, and backtracking capture the main cases, and the vast majority of languages can express them. (even if they're a little limp in the case of backtracking)

In case two, there's no apparent value to using monads to provide the sandboxing over, say, an object to manage access to each resource. In fact, Haskell's design seems worse, since IO provides access to all resources at once rather than one resource at a time.

This is the design choice made in the vast majority of OOP languages — not only is it simpler, but it seems to me that it's much safer however you slice the decision that Haskell made.

The one remaining benefit I can see for monads as a control flow tool — similar to Java with its checked exceptions, Haskell allows you to change the types of parts of your code based on their failure modes. However, converting between monad transformer stacks is typically prohibitively annoying, so most Haskell programmers build a standardized Monad transformer stack for their whole program, losing them even that advantage.

There might still be a case for monads as a tool to provide data structure operations — Haskell uses them for its list comprehensions — but I personally think Haskell's list comprehensions are arguably underfeatured compared to, say, C#'s, which provide "order by" functionality — and I think that's a consequence of supporting them through Monad.

I'll add something else: you really can't write most loops in Haskell that you can write in other languages. I think having to resort to forM and mapM is really bad, actually, and the lack of an ergonomic interface to mutable variables really sucks.