An opinionated guide to Haskell in 2018

2018-02-10 ◉ haskell

For me, this month marks the end of an era in my life: as of February 2018, I am no longer employed writing Haskell. It's been a fascinating two years, and while I am excitedly looking forward to what I'll be doing next, it's likely I will continue to write Haskell in my spare time. I'll probably even write it again professionally in the future.

In the meantime, in the interest of both sharing with others the small amount of wisdom I've gained and preserving it for my future self, I've decided to write a long, rather dry overview of a few select parts of the Haskell workflow I developed and the ecosystem I settled into. This guide is, as the title notes, *opinionated*—it is what I used in my day-to-day work, nothing more—and I don't claim that anything here is the only way to write Haskell, nor even the best way. It is merely what I found helpful and productive. Take from it as much or as little as you'd like.

Build tools and how to use them

When it comes to building Haskell, you have options. And frankly, most of them are pretty good. There was a time when `cabal-install` had a (warranted) reputation for being nearly impossible to use and regularly creating dependency hell, but I don't think that's the case anymore (though you *do* need to be a little careful about how you use it). Sandboxed builds work alright, and `cabal new-build` and the other `cabal new-*` commands are even better. That said, the UX of `cabal-install` is still less-than-stellar, and it has sharp edges, especially for someone coming from an ecosystem without a heavyweight compilation process like JavaScript, Ruby, or Python.

Nix is an alternative way to manage Haskell dependencies, and it seems pretty cool. It has a reputation for being large and complicated, and that reputation does not seem especially unfair, but you get lots of benefits if you're willing to pay the cost. Unfortunately, I have never used it (though I've read a lot about it), so I can't comment much on it here. Perhaps I'll try to go all-in with Nix when I purchase my next computer, but for now, my workflow works well enough that I don't feel compelled to switch.

Personally, I use `stack` as my Haskell build tool. It's easy to use, it works out of the box, and while it doesn't enjoy the same amount of caching as `cabal new-build` or Nix, it caches most packages, and it also makes things like git-hosted sources incredibly easy, which (as far as I can tell) can't be done with `cabal-install` alone.

This section is going to be a guide on how *I* use `stack`. If you use `cabal-install` with or without Nix, great! Those tools seem good, too. This is not an endorsement of `stack` over the other build tools, just a description of how I use it, the issues I ran into, and my solutions to them.

Understanding `stack`'s model and avoiding its biggest gotcha

Before using `stack`, there are a few things every programmer should know:

- `stack` is not a package manager, it is a build tool. It does not manage a set of "installed" packages; it simply builds targets and their dependencies.
- The command to build a target is `stack build <target>`. Just using `stack build` on its own will build the current project's targets.
- **You almost certainly do not want to use `stack install`.**

This is the biggest point of confusion I see among new users of `stack`. After all, when you want to install a package with `npm`, you type `npm install <package>`. So a new Haskeller decides to install `lens`, types `stack install lens`, and then later tries `stack uninstall lens`, only to discover that no such command exists. What happened?

`stack install` is not like `npm install`. `stack install` is like `make install`. It is nothing more than an alias for `stack build --copy-bins`, and *all* it does is build the target and copy all of its executables into some relatively global location like `~/.local/bin`. This is usually not what you want.

This design decision is not unique to `stack`; `cabal-install` suffers from it as well. One can argue that it isn't unintuitive because it really is just following what `make install` conventionally does, and the fact that it happens to conflict with things like `npm install` or even `apt-get install` is just a naming clash. I think that argument is a poor one, however, and I think the decision to even include a `stack install` command was a bad idea.

So, remember: don't use `stack install`! `stack` works best when everything lives inside the current project's *local* sandbox, and `stack install` copies executables into a *global* location by design. While it might sometimes appear to work, it's almost always wrong. The *only* situation in which `stack install` is the right answer is when you want to install an executable for a use unrelated to Haskell development (that is, something like `pandoc`) that just so happens to be provided by a Haskell package. **This means no running `stack install ghc-mod` or `stack install intero` either, no matter what READMEs might tell you!** Don't worry: I'll cover the proper way to install those things later.

Actually building your project with `stack`

Okay, so now that you know to never use `stack install`, what *do* you use? Well, `stack build` is probably all you need. Let's cover some variations of `stack build` that I use most frequently.

Once you have a `stack` project, you can build it by simply running `stack build` within the project directory. However, for local development, this is usually unnecessarily slow because it runs the GHC optimizer. For faster development build times, pass the `--fast` flag to disable optimizations:

```
$ stack build --fast
```

By default, `stack` builds dependencies with coarse-grained, package-level parallelism, but you can enable more fine-grained, module-level parallel builds by adding `--ghc-options=-j`. Unfortunately, there are conflicting accounts on whether or not this actually makes things faster or slower in practice, and I haven't extensively tested to see whether or not this is the case, so I mostly leave it off.

Usually, you also want to build and run the tests along with your code, which you can enable with the `--test` flag. Additionally, `stack test` is an alias for `stack build --test`, so these two commands are equivalent:

```
$ stack build --fast --test
$ stack test --fast
```

Also, it is useful to build documentation as well as code! You can do this by passing the `--haddock` flag, but unfortunately, I find Haddock sometimes takes an unreasonably long time to run. Therefore, since I usually only care about running Haddock on my dependencies, I usually pass the `--haddock-deps` flag instead, which prevents having to re-run Haddock every time you build:

```
$ stack test --fast --haddock-deps
```

Finally, I usually want to build and test my project in the background whenever my code changes. Fortunately, this can be done easily by using the `--file-watch` flag, making it easy to incrementally change project code and immediately see results:

```
$ stack test --fast --haddock-deps --file-watch
```

This is the command I usually use to develop my Haskell projects.

Accessing local documentation

While Haskell does not always excel on the documentation front, a small amount of documentation is almost always better than no documentation at all, and I find my dependencies' documentation to be an invaluable resource while developing. I find many people just look at docs on Hackage or use the hosted instance of Hoogle, but this sometimes leads people astray: they might end up looking at the wrong version of the documentation! Fortunately, there's an easy solution to this problem, which is to browse the documentation `stack` installs locally, which is guaranteed to match the version you are using in your current project.

The easiest way to open local documentation for a particular package is to use the `stack haddock --open` command. For example, to open the documentation for `lens`, you could use the following command:

```
$ stack haddock --open lens
```

This will open the local documentation in your web browser, and you can browse it at your leisure. If you have already built the documentation using the `--haddock-deps` option I recommended in the previous section, this command should complete almost instantly, but if you haven't built the documentation yet, you'll have to wait as `stack` builds it for you on-demand.

While this is a good start, it isn't perfect. Ideally, I want to have *searchable* documentation, and fortunately, this is possible to do by running Hoogle locally. This is easy enough with modern versions of `stack`, which have built-in Hoogle integration, but it still requires a little bit of per-project setup, since you need to build the Hoogle search index with the following command:

```
$ stack hoogle -- generate --local
```

This will install Hoogle into the current project if it isn't already installed, and it will index your dependencies' documentation and generate a new Hoogle database. Once you've done that, you can start a web server that serves a local Hoogle search page with the following command:

```
$ stack hoogle -- server --local --port=8080
```

Navigate to `http://localhost:8080` in your web browser, and you'll have a fully-searchable index of all your Haskell packages' documentation. Isn't that neat?

Unfortunately, you *will* have to manually regenerate the Hoogle database when you install new packages and their documentation, which you can do by re-running `stack hoogle -- generate --local`. Fortunately, regenerating the database doesn't take very long, as long as you've been properly rebuilding the documentation with `--haddock-deps`.

Configuring your project

Every project built with `stack` is configured with two separate files:

- The `stack.yaml` file, which controls which packages are built and what versions to pin your dependencies to.

- The `<project>.cabal` file *or* `package.yaml` file, which specifies build targets, their dependencies, and which GHC options to apply, among other things.

The `.cabal` file is, ultimately, what is used to build your project, but modern versions of `stack` generate projects that use hpack, which uses an alternate configuration file, the `package.yaml` file, to generate the `.cabal` file. This can get a little bit confusing, since it means you have *three* configuration files in your project, one of which is generated from the other one.

I happen to use and like hpack, so I use a `package.yaml` file and allow hpack to generate the `.cabal` file. I have no real love for YAML, and in fact I think custom configuration formats are completely fine, but the primary advantage of hpack is the ability to specify things like GHC options and default language extensions for all targets at once, instead of needing to duplicate them per-target.

You can think of the `.cabal` or `package.yaml` file as a specification for *how* your project is built and *what packages* it depends on, but the `stack.yaml` file is a specification of precisely *which version* of each package should be used and where it should be fetched from. Also, each `.cabal` file corresponds to precisely *one* Haskell package (though it may have any number of executable targets), but a `stack.yaml` file can specify multiple different packages to build, useful for multi-project builds that share a common library. The details here can be a little confusing, more than I am likely going to be able to explain in this blog post, but for the most part, you can get away with the defaults unless you’re doing something fancy.

Setting up editor integration

Currently, I use Atom to write Haskell. Atom is not a perfect editor by any means, and it leaves a lot to be desired, but it's easy to set up, and the Haskell editor integration is decent.

Atom's editor integration is powered by `ghc-mod` , a program that uses the GHC API to provide tools to inspect Haskell programs. Installing `ghc-mod` must be done manually so that Atom's `haskell-ghc-mod` package can find it, and this is where a lot of people get tripped up. They run `stack install ghc-mod` , it installs `ghc-mod` into `~/.local/bin` , they put that in their `PATH` , and things work! …except when a new version of GHC is released a few months later, everything stops working.

As mentioned above, **`stack install` is not what you want**. Tools like `ghc-mod` , `hlint` , `hoogle` , `weeder` , and `intero` work best when installed as part of the sandbox, *not* globally, since that ensures they will match the current GHC version your project is using. This can be done per-project using the ordinary `stack build` command, so the easiest way to properly install `ghc-mod` into a `stack` project is with the following command:

```
$ stack build ghc-mod
```

Unfortunately, this means you will need to run that command inside every single `stack` project individually in order to properly set it up so that `stack exec -- ghc-mod` will find the correct executable. One way to circumvent this is by using a recently-added `stack` flag designed for this explicit purpose, `--copy-compiler-tool` . This is like `--copy-bins` , but it copies the executables into a *compiler-specific location*, so a tool built for GHC 8.0.2 will be stored separately from the same tool built for GHC 8.2.2. `stack exec` arranges for the executables for the current compiler version to end up in the `PATH`, so you only need to build and install your tools once per compiler version.

Does this kind of suck? Yes, a little bit, but it sucks a whole lot less than all your editor integration breaking every time you switch to a project that uses a different version of GHC. I use the following command in a fresh sandbox when a Stackage LTS comes out for a new version of GHC:

```
$ stack build --copy-compiler-tool ghc-mod hoogle weeder
```

This way, I only have to build those tools once, and I don't worry about rebuilding them again until a the next release of GHC. To verify that things are working properly, you should be able to create a fresh `stack` project, run a command like this one, and get a similar result:

```
$ stack exec -- which ghc-mod
/Users/alexis/.stack/compiler-tools/x86_64-osx/ghc-8.2.2/bin/ghc-mod
```

Note that this path is scoped to my operating system and my compiler version, but nothing else—no LTS or anything like that.

Warning flags for a safe build

Haskell is a relatively strict language as programming languages go, but in my experience, it isn't quite strict enough. Many things are not errors that probably ought to be, like orphan instances and inexhaustive pattern matches.

Fortunately, GHC provides *warnings* that catch these problems statically, which fill in the gaps. I recommend using the following flags on all projects to ensure everything is caught:

- `-Wall`
- `-Wcompat`
- `-Wincomplete-record-updates`
- `-Wincomplete-uni-patterns`
- `-Wredundant-constraints`

The `-Wall` option turns on *most* warnings, but (ironically) not all of them. The `-Weverything` flag truly turns on *all* warnings, but some of the warnings left disabled by `-Wall` really are quite silly, like warning when type signatures on polymorphic local bindings are omitted. Some of them, however, are legitimately useful, so I recommend turning them on explicitly.

`-Wcompat` enables warnings that make your code more robust in the face of future backwards-incompatible changes. These warnings are trivial to fix and serve as free future-proofing, so I see no reason not to turn these warnings on.

`-Wincomplete-record-updates` and `-Wincomplete-uni-patterns` are things I think ought to be enabled by `-Wall` because they both catch what are essentially partial pattern-matches (and therefore runtime errors waiting to happen). The fact that `-Wincomplete-uni-patterns` *isn't* enabled by `-Wall` is so surprising that it can lead to bugs being overlooked, since the extremely similar `-Wincomplete-patterns` *is* enabled by `-Wall`.

`-Wredundant-constraints` is a useful warning that helps to eliminate unnecessary typeclass constraints on functions, which can sometimes occur if a constraint was previously necessary but ends up becoming redundant due to a change in the function's behavior.

I put all five of these flags in the `.cabal` file (or `package.yaml`), which enables them everywhere, but this alone is unlikely to enforce a warning-free codebase, since the build will still succeed even in the presence of warnings. Therefore, when building projects in CI, I pass the `-Werror` flag (using `--ghc-options=-Werror` for `stack`), which treats warnings as errors and halts the build if any warnings are found. This is useful, since it means warnings don't halt the whole build while developing, making it possible to write some code that has warnings and still run the test suite, but it still enforces that pushed code be warning-free.

Any flavor you like

Haskell is both a language and a spectrum of languages. It is both a standard and a specific implementation. Haskell 98 and Haskell 2010 are good, small languages, and there are a few different implementations, but when people talk about "Haskell", unqualified, they're almost always talking about GHC.

GHC Haskell, in stark contrast to standard Haskell, is neither small nor particularly specific, since GHC ships with *dozens* of knobs and switches that can be used to configure the language. In theory, this is a little terrifying. How could anyone ever hope to talk about Haskell and agree upon how to write it if there are so many *different* Haskells, each a little bit distinct? Having a cohesive ecosystem would be completely hopeless.

Fortunately, in practice, this is not nearly as bad as it seems. The majority of GHC extensions are simple switches: a feature is either on or it is off. Turning a feature on rarely affects code that does not use it, so most extensions can be turned on by default, and programmers may simply avoid the features they do not wish to use, just as any programmer in any programming language likely picks a subset of their language's features to use on a daily basis. Writing Haskell is not different in this regard, only in the sense that it does not allow all features to be used by default; everything from minor syntactic tweaks to entirely new facets of the type system are opt-in.

Frankly, I think the UX around this is terrible. I recognize the desire to implement a standard Haskell, and the old `-fglasgow-exts` was not an especially elegant solution for people wishing to use nonstandard Haskell, but having to insert `LANGUAGE` pragmas at the top of every module just to take advantage of the best features GHC has to offer is a burden, and it is unnecessarily intimidating. I think much of the Haskell community finds the use of `LANGUAGE` pragmas preferable to enabling extensions globally using the `default-extensions` list in the `.cabal` file, but I cut across the grain on that issue *hard*. The vast majority of language extensions I use are extensions I want enabled all the time; a list of them at the top of a module is just distracting noise, and it only serves to bury the extensions I really do want to enable on a module-by-module basis. It also makes it tricky to communicate with a team which extensions are acceptable (or even preferable) and which are discouraged.

My ***strong*** recommendation if you decide to write GHC Haskell on a team is to agree as a group to a list of extensions the team is happy with enabling everywhere and putting those extensions in the `default-extensions` list in the `.cabal` file. This eliminates clutter, busywork, and the conceptual overhead of remembering which extensions are in favor, and which are discouraged. This is a net win, and it isn't at all difficult to look in the `.cabal` file when you want to know which extensions are in use.

Now, with that small digression out of the way, the question becomes precisely which extensions should go into that `default-extensions` list. I happen to like using most of the features GHC makes available, so I enable a whopping **34** language extensions *by default*. As of GHC 8.2, here is my list:

- `ApplicativeDo`
- `BangPatterns`
- `ConstraintKinds`
- `DataKinds`
- `DefaultSignatures`
- `DeriveFoldable`
- `DeriveFunctor`
- `DeriveGeneric`
- `DeriveLift`
- `DeriveTraversable`

- `DerivingStrategies`
- `EmptyCase`
- `ExistentialQuantification`
- `FlexibleContexts`
- `FlexibleInstances`
- `FunctionalDependencies`
- `GADTs`
- `GeneralizedNewtypeDeriving`
- `InstanceSigs`
- `KindSignatures`
- `LambdaCase`
- `MultiParamTypeClasses`
- `MultiWayIf`
- `NamedFieldPuns`
- `OverloadedStrings`
- `PatternSynonyms`
- `RankNTypes`

- `ScopedTypeVariables`
- `StandaloneDeriving`
- `TupleSections`
- `TypeApplications`
- `TypeFamilies`
- `TypeFamilyDependencies`
- `TypeOperators`

This is a lot, and a few of them are likely to be more controversial than others. Since I do not imagine everyone will agree with everything in this list, I've broken it down into smaller chunks, arranged from what I think ought to be least controversial to most controversial, along with a little bit of justification why each extension is in each category. If you're interested in coming up with your own list of extensions, the rest of this section is for you.

Trivial lifting of standards-imposed limitations

A few extensions are tiny changes that lift limitations that really have no reason to exist, other than that they are mandated by the standard. I am not sure why these restrictions are in the standard to begin with, other than perhaps a misguided attempt at making the language simpler. These extensions include the following:

- `EmptyCase`
- `FlexibleContexts`

- `FlexibleInstances`
- `InstanceSigs`
- `MultiParamTypeClasses`

These extensions have no business *not* being turned on everywhere. `FlexibleContexts` and `FlexibleInstances` end up being turned on in almost any nontrivial Haskell module, since without them, the typeclass system is pointlessly and artificially limited.

`InstanceSigs` is extremely useful, completely safe, and has zero downsides.

`MultiParamTypeClasses` are almost impossible to avoid, given how many libraries use them, and they are a completely obvious generalization of single-parameter typeclasses. Much like `FlexibleContexts` and `FlexibleInstances`, I see no real reason to ever leave these disabled.

`EmptyCase` is even stranger to me, since `EmptyDataDecls` is in Haskell 2010, so it's possible to define empty datatypes in standard Haskell but not exhaustively pattern-match on them! This is silly, and `EmptyCase` should be standard Haskell.

Syntactic conveniences

A few GHC extensions are little more than trivial, syntactic abbreviations. These things would be tiny macros in a Lisp, but they need to be extensions to the compiler in Haskell:

- `LambdaCase`

- `MultiWayIf`
- `NamedFieldPuns`
- `TupleSections`

All of these extensions are only triggered by explicit use of new syntax, so existing programs will never change behavior when these extensions are introduced.

`LambdaCase` only saves a few characters, but it eliminates the need to come up with a fresh, unique variable name that will only be used once, which is sometimes hard to do and leads to worse names overall. Sometimes, it really is better to leave something unnamed.

`MultiWayIf` isn’t something I find I commonly need, but when I do, it’s nice to have. It’s far easier to read than nested `if...then...else` chains, and it uses the existing guard syntax already used with function declarations and `case...of`, so it’s easy to understand, even to those unfamiliar with the extension.

`NamedFieldPuns` avoids headaches and clutter when using Haskell records without the accidental identifier capture issues of `RecordWildCards`. It’s a nice, safe compromise that brings some of the benefits of `RecordWildCards` without any downsides.

`TupleSections` is a logical generalization of tuple syntax in the same vein as standard operator sections, and it’s quite useful when using applicative notation. I don’t see any reason to not enable it.

Extensions to the deriving mechanism

GHC's typeclass deriving mechanism is one of the things that makes Haskell so pleasant to write, and in fact I think Haskell would be nearly unpalatable to write without it. Boilerplate generation is a good thing, since it defines operations in terms of a single source of truth, and generated code is code you do not need to maintain. There is rarely any reason to write a typeclass instance by hand when the deriving mechanism will write it automatically.

These extensions give GHC's typeclass deriving mechanism more power without any cost. Therefore, I see no reason *not* to enable them:

- `DeriveFoldable`
- `DeriveFunctor`
- `DeriveGeneric`
- `DeriveLift`
- `DeriveTraversable`
- `DerivingStrategies`
- `GeneralizedNewtypeDeriving`
- `StandaloneDeriving`

The first five of these simply extend the list of typeclasses GHC knows how to derive, something that will only ever be triggered if the user explicitly requests GHC derive one of those classes. `GeneralizedNewtypeDeriving` is quite possibly one of the most important extensions in all of Haskell, since it dramatically improves `newtype` s' utility. Wrapper types can inherit instances they need without any boilerplate, and making increased type safety easier and more accessible is always a good thing in my book.

`DerivingStrategies` is new to GHC 8.2, but it finally presents the functionality of GHC's `DeriveAnyClass` extension in a useful way. `DeriveAnyClass` is useful when used with certain libraries that use `DefaultSignatures` (discussed later) with `GHC.Generics` to derive instances of classes without the deriving being baked into GHC. Unfortunately, enabling `DeriveAnyClass` essentially disables the far more useful `GeneralizedNewtypeDeriving`, so I do *not* recommend enabling `DeriveAnyClass`. Fortunately, with `DerivingStrategies`, it's possible to opt into the `anyclass` deriving strategy on a case-by-case basis, getting some nice boilerplate reduction in the process.

`StandaloneDeriving` is useful when GHC's deriving algorithms aren't *quite* clever enough to deduce the instance context automatically, so it allows specifying it manually. This is only useful in a few small situations, but it's nice to have, and there are no downsides to enabling it, so it ought to be turned on.

Lightweight syntactic adjustments

A couple extensions tweak Haskell's syntax in more substantial ways than things like `LambdaCase`, but not in a significant enough way for them to really be at all surprising:

- `BangPatterns`
- `KindSignatures`
- `TypeOperators`

`BangPatterns` mirror strictness annotations on datatypes, so they are unlikely to be confusing, and they provide a much more pleasant notation for annotating the strictness of bindings than explicit uses of `seq`.

`KindSignatures` are also fairly self-explanatory: they're just like type annotations, but for types instead of values. Writing kind signatures explicitly is usually unnecessary, but they can be helpful for clarity or for annotating phantom types when `PolyKinds` is not enabled. Enabling `KindSignatures` doesn't have any adverse effects, so I see no reason not to enable it everywhere.

`TypeOperators` adjusts the syntax of types slightly, allowing operators to be used as type constructors and written infix, which is technically backwards-incompatible, but I'm a little suspicious of anyone using `(!@#$)` as a type variable (especially since standard Haskell does not allow them to be written infix). This extension is useful with some libraries like `natural-transformations` that provide infix type constructors, and it makes the type language more consistent with the value language.

Polymorphic string literals

I'm putting this extension in a category all of its own, mostly because I don't think any other Haskell extensions have quite the same set of tradeoffs:

- `OverloadedStrings`

For me, `OverloadedStrings` is not optional. Haskell's infamous "string problem" (discussed in more detail at the end of this blog post) means that `String` is a linked list of characters, and all code that cares about performance actually uses `Text`. Manually invoking `pack` on every single string literal in a program is just noise, and `OverloadedStrings` solves that noise.

That said, I actually find I don't use the polymorphism of string literals very often, and I'd be alright with monomorphic literals if I could make them *all* have type `Text`. Unfortunately, there isn't a way to do this, so `OverloadedStrings` is the next best thing, even if it sometimes causes some unnecessary ambiguities that require type annotations to resolve.

`OverloadedStrings` is an extension that I use so frequently, in so many modules (especially in my test suites) that I would rather keep it on everywhere so I don't have to care about whether or not it's enabled in the module I'm currently writing. On the other hand, it certainly isn't my favorite language extension, either. I wouldn't go as far as to call it a necessary evil, since I don't think it's truly "evil", but it does seem to be necessary.

Simple extensions to aid type annotation

The following two extensions significantly round out Haskell's language for referring to types, making it much easier to insert type annotations where necessary (for removing ambiguity or for debugging type errors):

- `ScopedTypeVariables`
- `TypeApplications`

That the behavior of `ScopedTypeVariables` is *not* the default is actually one of the most common gotchas for new Haskellers. Sadly, it can theoretically adjust the behavior of existing Haskell programs, so I cannot include it in the list of trivial changes, but I would argue such programs were probably confusing to begin with, and I have never seen a program in practice that was impacted by that problem. I think leaving `ScopedTypeVariables` off is much, much more likely to be confusing than turning it on.

`TypeApplications` is largely unrelated, but I include it in this category because it's quite useful and cooperates well with `ScopedTypeVariables`. Use of `TypeApplications` makes instantiation much more lightweight than full-blown type annotations, and once again, it has no downsides if it is enabled and unused (since it is a syntactic addition). I recommend enabling it.

Simple extensions to the Haskell type system

A few extensions tweak the Haskell type system in ways that I think are simple enough to be self-explanatory, even to people who might not have known they existed. These are as follows:

- `ConstraintKinds`
- `RankNTypes`

`ConstraintKinds` is largely just used to define typeclass aliases, which is both useful and self-explanatory. Unifying the type and constraint language also has the effect of allowing type-level programming with constraints, which is sometimes useful, but far rarer in practice than the aforementioned use case.

`RankNTypes` are uncommon, looking at the average type in a Haskell program, but they're certainly nice to have when you need them. The idea of pushing `forall`s further into a type to adjust how variables are quantified is something that I find people find fairly intuitive, especially after seeing them used once or twice, and higher-rank types do crop up regularly, if infrequently.

Intermediate syntactic adjustments

Three syntactic extensions to Haskell are a little bit more advanced than the ones I've already covered, and none of them are especially related:

- `ApplicativeDo`
- `DefaultSignatures`
- `PatternSynonyms`

`ApplicativeDo` is, on the surface, simple. It changes `do` notation to use `Applicative` operations where possible, which allows using `do` notation with applicative functors that are not monads, and it also makes operations potentially more performant when `(<*>)` can be implemented more efficiently than `(>>=)`. In theory, it sounds like there are no downsides to enabling this everywhere. However, there are are a few drawbacks that lead me to put it so low on this list:

1. It considerably complicates the desugaring of `do` blocks, to the point where the algorithm cannot even be easily syntactically documented. In fact, an additional compiler flag, `-foptimal-applicative-do`, is a way to *opt into* optimal solutions for `do` block expansions, tweaking the desugaring algorithm to have an $O(n^3)$ time complexity! This means that the default behavior is guided by a heuristic, and desugaring isn't even especially predictable. This isn't necessarily so bad, since it's really only intended as an optimization when some `Monad` operations are still necessary, but it does dramatically increase the complexity of one of Haskell's core forms.

2. The desugaring, despite being $O(n^2)$ by default, isn't even especially clever. It relies on a rather disgusting hack that recognizes `return e`, `return $ e`, `pure e`, or `pure $ e` expressions *syntactically*, and it completely gives up if an expression with precisely that shape is not the

final statement in a `do` block. This is a bit awkward, since it effectively turns `return` and `pure` into syntax when before they were merely functions, but that isn't all. It also means that the following `do` block is *not* desugared using `Applicative` operations:

```
do foo a b
   bar s t
   baz y z
```

This will use the normal, monadic desugaring, despite the fact that it is trivially desugared into `Applicative` operations as `foo a b *> bar s t *> baz y z`. In order to get `ApplicativeDo` to trigger here, the `do` block must be contorted into the following:

```
do foo a b
   bar s t
   r <- baz y z
   pure r
```

This seems like an odd oversight.

3. `TemplateHaskell` doesn't seem able to cope with `do` blocks when `ApplicativeDo` is enabled. I reported this as an issue on the GHC bug tracker, but it hasn't received any attention, so it's not likely to get fixed unless someone takes the initiative to do so.

4. Enabling `ApplicativeDo` can cause problems with code that may have assumed `do` would always be monadic, and sometimes, that can cause code that typechecks to lead to an infinite loop at runtime. Specifically, if

`do` notation is used to define `(<*>)` in terms of `(>>=)`, enabling `ApplicativeDo` will cause the definition of `(<*>)` to become self-referential and therefore divergent. Fortunately, this issue can be easily mitigated by simply writing `(<*>) = ap` instead, which is clearer and shorter than the equivalent code using `do`.

Given all these things, it seems `ApplicativeDo` is a little too new in a few places, and it isn't quite baked. Still, I keep it enabled by default. Why? Well, *usually* it works fine without any problems, and when I run into issues, I can disable it on a per-module basis by writing `{-# LANGUAGE NoApplicativeDo #-}`. I still find that keeping it enabled by default is fine the vast majority of the time, I just sometimes need to work around the bugs.

In contrast, `DefaultSignatures` isn't buggy at all, as far as I can tell, it's just not usually useful without fairly advanced features like `GADTs` (for type equalities) or `GHC.Generics`. I mostly use it for making lifting instances for `mtl`-style typeclasses easier to write, which I've found to be a tiny bit tricky to explain (mostly due to the use of type equalities in the context), but it works well. I don't see any real reason to leave this disabled, but if you don't think you're going to use it anyway, it doesn't really matter one way or the other.

Finally, `PatternSynonyms` allow users to extend the pattern language just as they are allowed to extend the value language. Bidirectional pattern synonyms are isomorphisms, and it's quite useful to allow those isomorphisms to be used with Haskell's usual pattern-matching syntax. I think this extension is actually quite benign, but I put it so low on this list because it seems infrequently used, and I get the sense most people consider it fairly advanced. I would argue, however, that it's a very pleasant, useful extension, and it's no more complicated than a number of the features in Haskell 98.

Intermediate extensions to the Haskell type system

Now we're getting into the meat of things. Everything up to this point has been, in my opinion, completely self-evident in its usefulness and simplicity. As far as I'm concerned, the extensions in the previous six sections have no business ever being left disabled. Starting in this section, however, I could imagine a valid argument being made either way.

The following three extensions add some complexity to the Haskell type system in return for some added expressive power:

- `ExistentialQuantification`
- `FunctionalDependencies`
- `GADTs`

`ExistentialQuantification` and `GADTs` are related, given that the former is subsumed by the latter, but `GADTs` also enables an alternative syntax. Both syntaxes allow packing away a typeclass dictionary or equality constraint that is brought into scope upon a successful pattern-match against a data constructor, something that is sometimes quite useful but certainly a departure from Haskell's simple ADTs.

`FunctionalDependencies` extend multi-parameter typeclasses, and they are almost unavoidable, given their use in the venerable `mtl` library. Like `GADTs`, `FunctionalDependencies` add an additional layer of complexity to the typeclass system in order to express certain things that would otherwise be difficult or impossible.

All of these extensions involve a tradeoff. Enabling `GADTs` also implies `MonoLocalBinds`, which disables let generalization, one of the most likely ways a program that used to typecheck might subsequently fail to do so. Some might argue that this is a good reason to turn `GADTs` on in a per-module basis, but I disagree: I actually want my language to be fairly consistent, and given that I know I am likely going to want to use `GADTs` *somewhere*, I want `MonoLocalBinds` enabled *everywhere*, not inconsistently and sporadically.

That aside, all these extensions are relatively safe. They are well-understood, and they are fairly self-contained extensions to the Haskell type system. I think these extensions have a very good power to cost ratio, and I find myself using them regularly (especially `FunctionalDependencies`), so I keep them enabled globally.

Advanced extensions to the Haskell type system

Finally, we arrive at the last set of extensions in this list. These are the most advanced features Haskell's type system currently has to offer, and they are likely to be the most controversial to enable globally:

- `DataKinds`
- `TypeFamilies`
- `TypeFamilyDependencies`

All of these extensions exist exclusively for the purpose of type-level programming. `DataKinds` allows datatype promotion, creating types that are always uninhabited and therefore can only be used phantom. `TypeFamilies` allows the definition of type-level functions that map types to other types.

Both of these are minor extensions to Haskell's surface area, but they have rather significant ramifications on the sort of programming that can be done and the way GHC's typechecker must operate.

`TypeFamilies` is an interesting extension because it comes in so many flavors: associated type synonyms, associated datatypes, open and closed type synonym families, and open and closed datatype families. Associated types tend to be easier to grok and easier to use, though they can also be replaced by functional dependencies. Open type families are also quite similar to classes and instances, so they aren't *too* tricky to understand. Closed type families, on the other hand, are a rather different beast, and they can be used to do fairly advanced things, *especially* in combination with `DataKinds`.

I happen to appreciate GHC's support for these features, and while I'm hopeful that an eventual `DependentHaskell` will alleviate many of the existing infelicities with dependently typed programming in GHC, in the meantime, it's often useful to enjoy what exists where practically applicable. Therefore, I have little problem keeping them enabled, since, like the vast majority of extensions on this list, these extensions merely lift restrictions, not adjust semantics of the language without the extensions enabled. When I am going to write a type family, I am going to turn on `TypeFamilies`; I see no reason to annotate the modules in which I decide to do so. I do not write an annotation at the top of each module in which I define a typeclass or a datatype, so why should I do so with type families?

`TypeFamilyDependencies` is a little bit different, since it's a very new extension, and it doesn't seem to always work as well as I would hope. Still, when it doesn't work, it fails with a very straightforward error message, and when it works, it is legitimately useful, so I don't see any real reason to leave it off if `TypeFamilies` is enabled.

Extensions intentionally left off this list

Given what I've said so far, it may seem like I would advocate flipping on absolutely every lever GHC has to offer, but that isn't actually true. There are a few extensions I quite intentionally do *not* enable.

`UndecidableInstances` is something I turn on semi-frequently, since GHC's termination heuristic is not terribly advanced, but I turn it on per-module, since it's useful to know when it's necessary (and in application code, it rarely is). `OverlappingInstances` and `IncoherentInstances`, in contrast, are completely banned—not only are they almost always a bad idea, GHC has a better, more fine-grained way to opt into overlapping instances, using the `{-# OVERLAPPING #-}`, `{-# OVERLAPPABLE #-}`, and `{-# INCOHERENT #-}` pragmas.

`TemplateHaskell` and `QuasiQuotes` are tricky ones. Anecdotes seem to suggest that enabling `TemplateHaskell` everywhere leads to worse compile times, but after trying this on a few projects and measuring, I wasn't able to detect any meaningful difference. Unless I manage to come up with some evidence that these extensions actually slow down compile times just by being *enabled*, even if they aren't used, then I may add them to my list of globally-enabled extensions, since I use them regularly.

Other extensions I haven't mentioned are probably things I just don't use very often and therefore haven't felt the need to include on this list. It certainly isn't exhaustive, and I add to it all the time, so I expect I will continue to do so in the future. This is just what I have for now, and if your favorite extension isn't included, it probably isn't a negative judgement against that extension. I just didn't think to mention it.

Libraries: a field guide

Now that you're able to build a Haskell project and have chosen which handpicked flavor of Haskell you are going to write, it's time to decide which libraries to use. Haskell is an expressive programming language, and the degree to which different libraries can shape the way you structure your code is significant. Picking the right libraries can lead to clean code that's easy to understand and maintain, but picking the wrong ones can lead to disaster.

Of course, there are *thousands* of Haskell libraries on Hackage alone, so I cannot hope to cover all of the ones I have ever found useful, and I certainly cannot cover ones that would be useful but I did not have the opportunity to try (of which there are certainly many). This blog post is long enough already, so I'll just cover a few categories of libraries that I think I can offer interesting commentary on; most libraries can generally speak for themselves.

Having an effect

One of the first questions Haskell programmers bump into when they begin working on a large application is how they're going to model effects. Few practical programming languages are pure, but Haskell is one of them, so there's no getting away from coming up with a way to manage side-effects.

For some applications, Haskell’s built-in solution might be enough: `IO`. This can work decently for data processing programs that do very minimal amounts of I/O, and the types of side-effects they perform are minimal. For these applications, most of the logic is likely to be pure, which means it’s already easy to reason about and easy to test. For other things, like web applications, it’s more likely that a majority of the program logic is going to be side-effectful by its nature—it may involve making HTTP requests to other services, interacting with a database, and writing to logfiles.

Figuring out how to structure these effects in a type-safe, decoupled, composable way can be tricky, especially since Haskell has so many different solutions. I could not bring myself to choose just one, but I did choose two: the so-called “`mtl` style” and freer monads.

`mtl` style is so named because it is inspired by the technique of interlocking monadic typeclasses and lifting instances used to model effects using constraints that is used in the `mtl` library. Here is a small code example of what `mtl` style typeclasses and handlers look like:

```haskell
class Monad m => MonadFileSystem m where
  readFile :: FilePath -> m String
  writeFile :: FilePath -> String -> m ()

  default readFile :: (MonadTrans t, MonadFileSystem m', m ~ t m') =
  readFile a = lift $ readFile a

  default writeFile :: (MonadTrans t, MonadFileSystem m', m ~ t m')
  writeFile a b = lift $ writeFile a b

instance MonadFileSystem IO where
  readFile = Prelude.readFile
  writeFile = Prelude.writeFile

instance MonadFileSystem m => MonadFileSystem (ExceptT e m)
instance MonadFileSystem m => MonadFileSystem (MaybeT m)
instance MonadFileSystem m => MonadFileSystem (ReaderT r m)
instance MonadFileSystem m => MonadFileSystem (StateT s m)
instance MonadFileSystem m => MonadFileSystem (WriterT w m)

newtype InMemoryFileSystemT m a = InMemoryFileSystemT (StateT [(File
  deriving (Functor, Applicative, Monad, MonadError e, MonadReader r

instance Monad m => MonadFileSystem (InMemoryFileSystemT m) where
  readFile path = InMemoryFileSystemT $ do
    vfs <- get
    case lookup path vfs of
      Just contents -> pure contents
      Nothing -> error ("readFile: no such file " ++ path)

  writeFile path contents = InMemoryFileSystemT $ modify $ \vfs ->
    (path, contents) : delete (path, contents) vfs
```

This is the most prevalent way to abstract over effects in Haskell, and it's been around for a long time. Due to the way it uses the typeclass system, it's also very fast, since GHC can often specialize and inline the typeclass dictionaries to avoid runtime dictionary passing. The main drawbacks are the amount of boilerplate required and the conceptual difficulty of understanding exactly how monad transformers, monadic typeclasses, and lifting instances all work together to discharge `mtl` style constraints.

There are various alternatives to `mtl`'s direct approach to effect composition, most of which are built around the idea of reifying a computation as a data structure and subsequently interpreting it. The most popular of these is the `Free` monad, a clever technique for deriving a monad from a functor that happens to be useful for modeling programs. Personally, I think `Free` is overhyped. It's a cute, mathematically elegant technique, but it involves a lot of boilerplate, and composing effect algebras is still a laborious process. The additional expressive power of `Free`, namely its ability to choose an interpreter dynamically, at runtime, is rarely necessary or useful, and it adds complexity and reduces performance for few benefits. (And in fact, this is still possible to do with `mtl` style, it's just uncommon because there is rarely any need to do so.)

A 2017 blog post entitled Free monad considered harmful discussed `Free` in comparison with `mtl` style, and unsurprisingly cast `Free` in a rather unflattering light. I largely agree with everything outlined in that blog post, so I will not retread its arguments here. I do, however, think that there is another abstraction that *is* quite useful: the so-called "freer monad" used to implement extensible effects.

Freer moves even further away from worrying about functors and monads, since its effect algebras do not even need to be functors. Instead, freer's effect algebras are ordinary GADTs, and reusable, composable effect handlers are easily written to consume elements of these datatypes. Unfortunately, the way this works means that GHC is still not clever enough to optimize freer monads as efficiently as `mtl` style, since it can't easily detect when the interpreter is chosen statically and use that information to specialize and inline effect implementations, but the cost difference is significantly reduced, and I've found that in real application code, the vast majority of the cost does not come from the extra overhead introduced by a more expensive `(>>=)`.

There are a few different implementations of freer monads, but I, sadly, was not satisfied with any of them, so I decided to contribute to the problem by creating yet another one. My implementation is called `freer-simple`, and it includes a streamlined API with more documentation than any other freer implementation. Writing the above `mtl` style example using `freer-simple` is more straightforward:

```
data FileSystem r where
  ReadFile :: FilePath -> FileSystem String
  WriteFile :: FilePath -> String -> FileSystem ()

readFile :: Member FileSystem r => FilePath -> Eff r String
readFile a = send $ ReadFile a

writeFile :: Member FileSystem r => FilePath -> String -> Eff r ()
writeFile a b = send $ WriteFile a b

runFileSystemIO :: LastMember IO r => Eff (FileSystem ': r) ~> Eff r
runFileSystemIO = interpretM $ \case
  ReadFile a -> Prelude.readFile a
  WriteFile a b -> Prelude.writeFile a b

runFileSystemInMemory :: [(FilePath, String)] -> Eff (FileSystem ':
runFileSystemInMemory initVfs = runState initVfs . fsToState where
  fsToState :: Eff (FileSystem ': effs) ~> Eff (State [(FilePath, St
  fsToState = reinterpret $ case
    ReadFile path -> get >>= \vfs -> case lookup path vfs of
      Just contents -> pure contents
      Nothing -> error ("readFile: no such file " ++ path)
    WriteFile path contents -> modify $ \vfs ->
      (path, contents) : delete (path, contents) vfs
```

(It could be simplified further with a little bit of Template Haskell to generate the `readFile` and `writeFile` function definitions, but I haven't gotten around to writing that.)

So which effect system do I recommend? I used to recommend `mtl` style, but as of only two months ago, I now recommend `freer-simple`. It's easier to understand, involves less boilerplate, achieves "good enough" performance, and generally gets out of the way wherever possible. Its API is designed to make it easy to do the sorts of the things you most commonly need to do, and it provides a core set of effects that can be used to build a real-world application.

That said, freer is indisputably relatively new and relatively untested. It has success stories, but `mtl` style is still the approach used by the majority of the ecosystem. `mtl` style has more library support, its performance characteristics are better understood, and it is a tried and true way to structure effects in a Haskell application. If you understand it well enough to use it, and you are happy with it in your application, my recommendation is to stick with it. If you find it confusing, however, or you end up running up against its limits, give `freer-simple` a try.

Through the looking glass: to lens or not to lens

There's no getting around it: `lens` is a behemoth of a library. For a long time, I wrote Haskell without it, and honestly, it worked out alright. I just wasn't doing a whole lot of work that involved complicated, deeply-nested data structures, and I didn't feel the need to bring in a library with such a reputation for having impenetrable operators and an almost equally impenetrable learning curve.

But, after some time, I decided I wanted to take the plunge. So I braced myself for the worst, pulled out my notebook, and started writing some code. To my surprise… it wasn't that hard. It made sense. Sure, I still don't know how it works on the inside, and I never did learn the majority of the exports in `Control.Lens.Operators`, but I had no need to. Lenses were useful in the way I had expected them to be, and so were prisms. One thing led to another, and before long, I understood the relationship between the various optics, the most notable additions to my toolkit being folds and traversals. Sure, the type errors were completely opaque much of the time, but I was able to piece things together with ample type annotations and time spent staring at ill-typed expressions. Before long, I had developed an intuition for `lens`.

After using it for a while, I retrospected on whether or not I liked it, and honestly, I still can't decide. Some lensy expressions were straightforward to read and were a pleasant simplification, like this one:

```
paramSpecs ^.. folded._Required
```

Others were less obviously improvements, such as this beauty:

```
M.fromList $ paramSpecs ^.. folded._Optional.filtered (has $ _2._Use
```

But operator soup aside, there was something deeper about `lens` that bothered me, and I just wasn't sure what. I didn't know how to articulate my vague feelings until I read a 2014 blog post entitled Lens is unidiomatic Haskell, which includes a point that I think is spot-on:

> Usually, types in Haskell are rigid. This leads to a distinctive style of composing programs: look at the types and see what fits where. This is impossible with `lens`, which takes overloading to the level mainstream Haskell probably hasn't seen before.
>
> We have to learn the new language of the `lens` combinators and how to compose them, instead of enjoying our knowledge of how to

> compose Haskell functions. Formally, `lens` types are Haskell function types, but while with ordinary Haskell functions you immediately see from types whether they can be composed, with `lens` functions this is very hard in practice.
>
> […]
>
> Now let me clarify that this doesn't necessarily mean that `lens` is a bad library. It's an *unusual* library. It's almost a separate language, with its own idioms, embedded in Haskell.

The way `lens` structures its types deliberately introduces a sort of subtyping relationship—for example, all lenses are traversals and all traversals are folds, but not vice versa—and indeed, knowing this subtyping relationship is essential to working with the library and understanding how to use it. It is helpfully documented with a large diagram on the `lens` package overview page, and that diagram was most definitely an invaluable resource for me when I was learning how to use the library.

On the surface, this isn't unreasonable. Subtyping is an enormously useful concept! The only reason Haskell dispenses with it entirely is because it makes type inference notoriously difficult. The subtyping relation between optics is one of the things that makes them so useful, since it allows you to easily compose a lens with a prism and get a traversal out. Unfortunately, the downside of all this is that Haskell does not truly have subtyping, so all of `lens`'s "types" really must be type aliases for types of roughly the same shape, namely functions. This makes type errors completely *baffling*, since the errors do not mention the aliases, only the fully-expanded types (which are often rather complicated, and their meaning is not especially clear without knowing how `lens` works under the hood).

So the above quote is correct: working with `lens` really *is* like working in a separate embedded language, but I'm usually okay with that. Embedded, domain-specific languages are good! Unfortunately, in this case, the host language is not very courteous to its guest. Haskell does not appear to be a powerful enough language for `lens` to be a language in its own right, so it must piggyback on top of Haskell's error reporting mechanisms, which are insufficient for `lens` to be a cohesive linguistic abstraction. Just as debugging code by stepping through the assembly it produces (or, perhaps more relevant in 2018, debugging a compile-to-JS language by looking at the emitted JavaScript instead of the source code) makes for an unacceptably leaky language. We would never stand for such a thing in our general-purpose language tooling, and we should demand better even in our embedded languages.

That said, `lens` is just too useful to ignore. It is a hopelessly leaky abstraction, but it's still an abstraction, and a powerful one at that. Given my selection of default extensions as evidence, I think it's clear I have zero qualms with "advanced" Haskell; I will happily use even `singletons` where it makes sense. Haskell's various language extensions are sometimes confusing in their own right, but their complexity is usually fundamental to the expressive power they bring. `lens` has some fundamental complexity, too, but it is mostly difficult for the wrong reasons. Still, while it is not the first library I reach for on every new Haskell project, manipulating nested data without `lens` is just too unpleasant after tasting the nectar, so I can't advise against it in good faith.

Sadly, this means I'm a bit wishy-washy when it comes to using `lens`, but I do have at least one recommendation: if you decide to use `lens`, it's better to go all-in. Don't generate lenses for just a handful of datatypes, do it for *all* of them. You can definitely stick to a subset of the `lens` library's features, but don't apply it in some functions but not others. Having too many different, equally valid ways of doing things leads to confusion and inconsistency, and inconsistency minimizes code reuse and leads to duplication and spaghetti. Commit to using `lens`, or don't use it at all.

Mitigating the string problem

Finally, Haskell has a problem with strings. Namely, `String` is a type alias for `[Char]`, a lazy, singly linked list of characters, which is an awful representation of text. Fortunately, the answer to this problem is simple: ban `String` in your programs.

Use `Text` everywhere. I don't really care if you pick strict `Text` or lazy `Text`, but pick one and stick to it. Don't ever use `String`, and *especially* don't ever, *ever*, ***ever*** use `ByteString` to represent text! There are enormously few legitimate cases for using `ByteString` in a program that is not explicitly about reading or writing raw data, and even at that level, `ByteString` should only be used at program boundaries. In that sense, I treat `ByteString` much the same way I treat `IO`: push it to the boundaries of your program.

One of Haskell's core tenets is making illegal states unrepresentable. Strings are not especially useful datatypes for this, since they are sequences of arbitrary length made up of atoms that can be an enormously large number of different things. Still, string types enforce a very useful invariant, a notion of a sequence of human-readable characters. In the presence of Unicode, this is a more valuable abstraction than it might seem, and the days of treating strings as little different from sequences of bytes are over. While strings make a poor replacement for enums, they are quite effective at representing the incredible amount of text humans produce in a staggeringly large number of languages, and they are the right type for that job.

`ByteString`, on the other hand, is essentially never the right type for any job. If a type classifies a set of values, `ByteString` is no different from `Any`. It is the structureless type, the all-encompassing blob of bits. A `ByteString` could hold anything at all—some text, an image, an executable program—and the type system certainly isn't going to help to answer that question. The only use case I can possibly imagine for passing around a `ByteString` in your program rather than decoding it into a more precise type is if it truly holds opaque data, e.g. some sort of token or key provided by a third party with no structure guaranteed whatsoever. Still, even this should be wrapped in a `newtype` so that the type system enforces this opaqueness.

Troublingly, `ByteString` shows up in many libraries' APIs where it has no business being. In many cases, this seems to be things where ASCII text is expected, but this is hardly a good reason to willingly accept absolutely anything and everything! Make an `ASCII` type that forbids non-ASCII characters, and provide a `ByteString -> Maybe ASCII` function. Alternatively, think harder about your problem in question to properly support Unicode as you almost certainly ought to.

Other places `ByteString` appears are similarly unfortunate. Base-64 encoding, for example, could be given the wonderfully illustrative type `ByteString -> Text`, or even `ByteString -> ASCII`! Such a type makes it immediately clear why base-64 is useful: it allows transforming arbitrary binary data into a reliable textual encoding. If we consider that `ByteString` is essentially `Any`, this function has the type `Any -> ASCII`, which is amazingly powerful! We can convert *anything* to ASCII text!

Existing libraries, however, just provide the boring, disappointingly inaccurate type `ByteString -> ByteString`, which is one of the most useless types there is. It is essentially `Any -> Any`, the meaningless function type. It conveys nothing about what it does, other than that it is pure. Giving a function this type is scarcely better than dynamic typing. Its mere existence is a failure of Haskell library design.

But wait, it gets worse! `Data.Text.Encoding` exports a function called `decodeUtf8`, which has type `ByteString -> Text`. What an incredible function with a captivating type! Whatever could it possibly do? Again, this function's type is basically `Any -> Text`, which is remarkable in the power it gives us. Let's try it out, shall we?

```
ghci> decodeUtf8 "\xc3\x28"
"*** Exception: Cannot decode byte '\x28': Data.Text.Internal.Encodir
```

Oh. Well, that's a disappointment.

Haskell's string problem goes deeper than `String` versus `Text`; it seems to have wound its way around the collective consciousness of the Haskell community and made it temporarily forget that it cares about types and totality. This isn't that hard, I swear! I can only express complete befuddlement at how many of these APIs are just completely worthless.

Fortunately, there is a way out, and that way out is `text-conversions`. It is the first Haskell library I ever wrote. It provides *type safe*, *total* conversions between `Text` and various other types, and it is encoding aware. It provides appropriately-typed base–16 and base–64 conversion functions, and is guaranteed to never raise any exceptions. Use it, and apply the Haskell philosophy to your strings, just as you already do for everything else in your program.

Closing thoughts

Phew.

When I started writing this blog post, it used the phrase "short overview" in the introduction. It is now over ten thousand words long. I think that's all I have it in me to say for now.

Haskell is a wonderful language built by a remarkable group of people. Its community is often fraught with needlessly inflammatory debates about things like the value of codes of conduct, the evils of Hackage revisions, and precisely how much or how little people ought to care about the monad laws. These flame wars frustrate me to no end, and they sometimes go so far as to make me ashamed to call myself a part of the Haskell community. Many on the "outside" seem to view Haskellers as an elitist, mean-spirited cult, more interested in creating problems for itself than solving them.

That perception is categorically wrong.

I have never been in a community of programmers so dedicated and passionate about applying thought and rigor to building software, then going out and *actually doing it*. I don't know anywhere else where a cutting-edge paper on effect systems is discussed by the very same people who are figuring out how to reliably deploy distributed services to AWS. Some people view the Haskell community as masturbatory, and to some extent, they are probably right. One of my primary motivators for writing Haskell is that it is fun and it challenges me intellectually in ways that other languages don't. But that challenge is not a sign of uselessness, it is a sign that Haskell is *so close* to letting me do the right thing, to solving the problem the right way, to letting me work without compromises. When I write in most programming languages, I must constantly accept that my program will never be robust in all the ways I want it to be, and I might as well give up before I even start. Haskell's greatest weakness is that it tempts me to try.

Haskell is imperfect, as it will always be. I doubt I will ever be satisfied by any language or any ecosystem. There will always be more to learn, more to discover, better tools and abstractions to develop. Many of them will not look anything like Haskell; they may not involve formal verification or static types or effect systems at all. Perhaps live programming, structural editors, and runtime hotswapping will finally take over the world, and we will find that the problems we thought we were solving were irrelevant to begin with. I can't predict the future, and while I've found great value in the Haskell school of program construction, I dearly hope that we do not develop such tunnel vision that we cannot see that there may be other ways to solve these problems. Many of the solutions are things we likely have not even begun to think about. Still, whether that happens or not, it is clear to me that Haskell is a point in the design space unlike any other, and we learn almost as much from the things it gets wrong as we do from the things it gets right.

It’s been a wonderful two years, Haskell. I won’t be a stranger.

← Reimplementing Hackett’s type language: expanding to custom core forms in Racket

A space of their own: adding a type namespace to Hackett →