02

December 2020

CHANGE HERE for the FP LINE

Monthly Digital Functional Programming Magazine

{- 1 INTRODUCTION

We are delighted to present our second issue of Bind The Gap (BTG), which also happens to be the special festive winter issue!

The year 2020 was challenging for all of us. But at the same time, many exciting events happened in the Haskell community. In this month's edition, we are going to overview notable milestones in the Haskell ecosystem, highlight some features and discuss the general roadmap for Haskell in the upcoming year.

Many people responded positively to BTG's first issue, and we are extremely happy about that because we put our souls into work on it! We are grateful for those who helped us spread the news about this new project across different FP communities and who left constructive feedback about the pilot issue. Taking that into account and inspired by your kind words, we continued working on some of our permanent rubrics, but we are also exploring new ways to share the news with everyone.

In this edition, we have many new surprises for you as well. **Chris Penner** joined us with the part about beloved lenses. **Impure Pics** is ready to bring joy to our readers one more time. We are grateful for their work. We also want to say thanks to **Richard Eisenberg** for taking his time to speak to us, and also to **Cate Roxl** for such tremendous help with proofreading!

We hope you enjoy our festive Edition of Bind The Gap. Merry Christmas and Happy New Year 2021 everyone! Enjoy the reading!

DMITRII KOVANIKOV ❄ ❄ VERONIKA ROMASHKINA,
EDITORS-IN-CHIEF

-}

MAP LINE

YEAR SUMMARY

Summary of 2020 for Haskell

3-4

LOCO-MOTIVE

Introduction to **Dependent Types** and interview with *Richard Eisenberg*

5-10

THE -WALL STREET ANALYTICS

Analytics of -Wmissing-export-lists

11

DERIVING ALL THE WAY

Tour to deriving in 2020

12-13

WHINE SOMMELIER

Review of the time library

14-16

THE WONDERFUL WIZARD OF OS

Update copyrights using headroom

17-18

FOLD LINE

LENSALOT

Chris Penner crusades lenses and other optics

19-21

RANK N ROLL

10 blog posts about Haskell in 2020

22-23

OVERCOMING STEREO TYPES

Using type-level programming in Haskell to parse tricky JSON

24-25

PURE GOLD BY IMPURE PICS

Illustrations by *Impure Pics*

26

BASELINE

Comparing all list *sorting* functions from base

27-29

FUN FOR THE WHOLE TYPE FAMILY

FP Humour, challenges, puzzles and surveys

29-31

Year 2020 Summary

It is the end of the crazy year 2020, phew. Looking into what has happened, it turned out that so many exciting things happened to Haskell during this time! We want to share these positive memories with all of you, so we can start a new year inspired by the accomplishments we all together achieved during these extraordinary times.

Amazingly, Haskell celebrated its **30th** anniversary this year! The first version of the Haskell report was published on 1 April 1990. But Haskell came a long way to become what we all use and love now, and it is a significant milestone.

One of the most positively met things happened to Haskell in 2020 is the creation of the **Haskell Foundation** — an organisation dedicated to broadening the adoption of Haskell, by supporting its ecosystem of tools, libraries, education, and research. Read the pilot issue of Bind The Gap, where we interviewed Simon Peyton Jones, to learn more about the Haskell Foundation.

A lot of progress has been made on the **IDE** front. In January, during the Bristol Hackathon, the *Haskell IDE Engine (HIE)* team joined forces with the *ghcide* team to work on **Haskell Language Server (HLS)**. And now, the VSCode Haskell plugin is one of the most reliable and featureful IDEs in Haskell. Yes, no more "Haskell has no IDE" complaints! But it is still the start of exciting technologies.

* HLS logo by George Thomas

Development of **GHC** is in full swing. The main Haskell compiler continues to provide more awesome releases. The GHC developers team released during 2020 the stable version GHC 8.8.4, new GHC 8.10 with the first version of the groundbreaking non-moving garbage collector, and even the first release candidate of GHC 9.0 with lots of fantastic stuff!

GHC's innovation side also experienced a lot of good buzz. The compiler received and accepted a lot of significant proposals, shaping the future of Haskell:

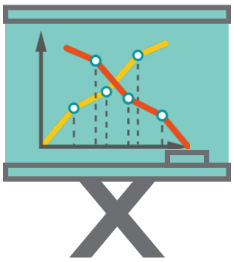
- **RecordDotSyntax** — solution to the record problem in Haskell.
- **GHC2021** — established set of enabled by default Haskell extensions.
- **LinearTypes** — more advances in the type system which was even implemented for GHC 9.0.
- **Ergonomic Dependent Types** — decision on whether we want to move Haskell and GHC towards support for Dependent Types.

ON AIR

Due to obvious reasons, a lot of things moved from *offline* to **online**, which surprisingly was not all bad for gathering from all corners of the world. We saw more online conferences (HaskellLove, ICFP, Haskell eXchange), meetups, more streams, more video tutorials. And generally, it became easier in some sense to gather up from different countries without leaving the bed (and staying at home!).

Haskell participated in **Google Summer of Code 2020 (GSOC)**, which shows the great interest and support of the community. This year again, many outstanding projects were involved in improving the state of the Haskell ecosystem and were completed successfully! Read about them in the Haskell.Org GSOC summary.

Unfortunately, the year disastrously affected lots of people's jobs. At the same time, we also noticed much more **job announcements** in Haskell than in previous years!



Hope this means that significantly more companies started using functional principles to build their products which will lead to the broader adoption and growth of the Haskell community.

Lots of important **materials** were released this year as well. Different learning resources, guides, video tutorials, etc. Several **books** were published during this period, and more books announced as being worked on:

- **Abstractions in Context**
  by *William Yao*
- **Algebra-Driven Design**
  by *Sandy Maguire*
- **Algorithm Design with Haskell**
  by *Richard Bird* and *Jeremy Gibbons*
- **Functional Design and Architecture**
  by *Alexander Granin*
- **Isomorphism - Mathematics of Programming**
  by *Liu Xinyu*
- (In progress) **Do Pure Haskell Interview**
  by *Dmitrii Kovanikov* and *Veronika Romashkina* (*Kowainik*)
- (In progress) **Pragmatic type-level design**
  by *Alexander Granin*
- (In progress) **Production Haskell**
  by *Matt Parsons*
- (In progress) **Sockets and Pipes**
  by *Chris Martin* and *Julie Moronuki* (*Type Classes*)

And we also want to reflect a bit on what we – **Kowainik** – managed to achieve in this weird and quite difficult year for us. Actually, it was extremely productive! We:

- Founded this monthly FP magazine **Bind The Gap**
- Made the **Learn4Haskell** course, during which we reviewed 480+ pull requests and helped ~200 people to start with Haskell
- Implemented Stan — Haskell static analyser, a big project that used modern GHC features such as HIE files to provide robust static code analysis
- Participated in the *Bristol Hackathon*, where we developed policeman — Haskell PVP adviser — and later wrote our experience report about our project
- Gave **4 talks** this year — two talks about Stan at the Haskell Love conference and Haskell Implementers Workshop, one talk about alternative standard library relude at the Haskell Amsterdam meetup, and Immutable Conversations with Alejandro Serrano
- Wrote **13 blog posts**, including guides, tutorials, experience reports, and created two awesome lists (awesome-haskell-sponsorship and awesome-cabal)
- Developed and published to Hackage **10 Haskell libraries and tools** (autopack, colourista, extensions, prolens, policeman, stan, validation-selective, trial, trial-optparse-applicative, trial-tomland), not to mention maintenance and updates to existing libraries (Summoner 2.0 with GitHub actions support, major tomland update, relude-0.7.0.0 and much more)
- Created a **mailing list** for a better experience for our readers

As you can see, Haskell is rapidly evolving in all means. The language and its community are very active. And more wonderful things are yet to come to Haskell next year!

LOCO MOTIVE

There always were a lot of talks around **Dependent Types (DT)** in Haskell, both within the community and outside. Finally, closer to the end of the year 2020, GHC (the main Haskell compiler) received a critical proposal regarding **Ergonomic Dependent Types [1]**. This is huge news. We are lucky to be in the community at such times when we can watch and be involved in historical events and changes in the language. In order to keep up with the important trends that may affect us all in many functional communities, we decided to thoroughly explore the feature and highlight the main parts you need to know about Dependent Types.

We start with answering straightaway a very popular counter-argument that people usually use against Haskell: *"If Haskell is so type-safe, why doesn't it have Dependent Types?"* The thing is, when Haskell was created, DT weren't so popular, so the language was designed without keeping in mind the ergonomic work with them. Therefore, Haskell specification doesn't have any mention and design ideas for this feature. However, recently more and more people started showing their interest in applying this concept to Software Development. Haskell already has multiple features in the area of types, so for many people Dependent Types seems like a natural next step for the language. Currently, DT is discussed to be designed as an add-on to the language called extensions in Haskell. We even talked to the leading advocate of DT and the author of the proposal **Richard Eisenberg** to clarify this feature's vision in the ecosystem.

The *ergonomic dependent types* proposal is vital for Haskell not only as a feature but also because it defines the language roadmap and further direction. Some people are even too extreme about this position, and they see Haskell of the future only with Dependent Types and are confident that this would become its killer-feature that helps Haskell reach the mainstream. So, learning a bit about this possible significant change to the language would benefit all of us. We hope that this section will help you by providing more information about Dependent Types and its implications to the entire Haskell world.

Let's start with the simple definition of DT:

DEPENDENT TYPE IS A TYPE THAT DEPENDS ON VALUES OF FUNCTION ARGUMENTS (HENCE THE NAME).

In Haskell, types are static, which means that you get what you write, or what the compiler figured out for you due to the type inference. At most, types can depend on other types, e.g. with the Type Families feature in Haskell. But dependent types blur the line between types and values making types first-class entities. First-class entities are not a new thing for Haskell, and it is one of the functional programming concepts. We all love and use functions as first-class values, which means that we can pass functions as the arguments to other functions, like in the map function. Similarly to this common concept, with types as first-class entities, functions can take types as arguments and return types as their result or, most importantly, return values of different types depending on arguments values.

It sounds like a handy property to use, so it seems like there is a big DT application area. However, when looking at all examples advocating for DT's usefulness and necessity, we always see some artificial, small and not really what you would write at your job, for instance. Examples usually involve Inductive Peano Natural numbers or typed lambda calculus, but to be honest, we haven't seen many people writing their own data types for Naturals daily. So let's try something new, and see if we could explain DT and use them on some minimal code example which could be easily found in some small application program.

Here is the deal: we want to write a function that configures the settings of the application. In particular, it either returns a default application configuration or reads it from the file, depending on the provided settings method. We can model this situation in Haskell with a sum type and a function that pattern matches on it:

```haskell
data Settings
    = Default
    | FromPath FilePath

configure :: Settings -> IO Config
configure Default = pure defaultConfig
configure (FromPath path) =
    readFile path >>= decodeConfig
```

It is straightforward and absolutely valid Haskell, but notice some redundancy: when the settings is Default, we still require our function to work in IO, so we wrap the config in pure. Though in reality, IO is only needed to read from the file in another method. It is not a problem per se for now, but let's say that we want to implement a pure test and we want to avoid the possibility of doing any IO in this test. And this is exactly where we can apply DT to solve it! While this example may look superficial, it should demonstrate the capabilities and behaviour of dependent types.

The solution to our problem in a hypothetical Haskell syntax (close to the currently proposed design of DT in Haskell) may be written as follows. First, we need to implement a function that takes a value of type Settings, a type (which represents our Config type) and either keeps the type unchanged or wraps it in IO.

```haskell
settingsType
    :: foreach (settings :: Settings)
    -> Type
    -> Type
settingsType Default t = t
settingsType (FromPath _) t = IO t
```

[1]: http://bit.ly/edt-proposal

This function represents the switcher of types and depending on the chosen settings method, and we will decide whether to wrap everything in IO (when we need to read the file) or keep it as it is, without adding any side-effects.

Now, we can use this function **in the type** of configure to allow the resulting type to depend on a value of type Settings. Note that we use settingsType as a function on the type-level to decide on the output type:

```
configure
    :: foreach (settings :: Settings)
    -> settingsType settings Config
configure Default = defaultConfig
    -- ^ type is 'Config' here
configure (FromPath path) =
    readFile path >>= decodeConfig
    -- ^ type is 'IO Config' here
```

The exact syntax is yet to be confirmed in the future. But you already can see how DT provide us with some flexibility. At the same time, the code becomes more complicated.

To recap what we did:

1. We defined a function that takes a Type and returns a new Type. The function uses the new "foreach" keyword to enable DT machinery. You can see how we can mix ordinary values (of type Settings) and types in the same function.
2. We use this function in the type of another function. When we pattern match on the Settings constructor, the function on types computes its result as well. It is possible because we know the pattern during the compile-time, and the compiler can figure out how the function behaves for that pattern and confirm if types match.

Dependent types exist for a while in other programming languages, for instance, Agda, Idris, Coq, etc. They are known to be used mostly in research areas (for writing proofs) or in verification systems where it is crucial to get everything correct (e.g. aeroplane software). But DT can also be used to solve the following more real-world-oriented small problems:

1. Type-safe printf function, that allows you to write printf "Name %s and value %d" and get a type-safe function that takes exactly two arguments — a string and an integer.

2. Regular expressions with result types depending on the expression. For instance, you can get the type "list of strings" if the regex matches multiple text entries.

3. The type of SQL query result can be automatically derived from the query text and the schema type.

4. Matrix multiplication, where matrix sizes can be on type-level, ensuring the resulting matrix size.

5. Neural networks with parameters on the type-level, allowing to specify the structure on the network in types and get more safety in ensuring that all math operations are correct, or even derive the whole neural network solely from types.

You can see that these problems occur in a daily programming job and are shared across different languages. And it makes sense that they already have solutions without using dependent types. However, DT will provide an alternative more type-safe implementation, but with different trade-offs.

Trade-offs are the central part of DT discussions in Haskell. The proposal's decision would depend on how the language is ready to handle different difficulties and pay attention to various aspects of the community. If the language decides to move towards supporting ergonomic dependent types, it will affect how people should write Haskell. According to the Dependent Haskell (DH) design, none of these changes immediately affect existing users. You still can continue writing Haskell as before. But most likely, the proposal will change the way people will write in Haskell, as many enthusiasts are interested in trying DT in their projects (hobby or even at work). If you are a maintainer, users might want to use your libraries with DH ergonomically, and you might need to use DT as well.

Here are a few controversial changes we will experience with the DT, that have been already brought up by different people:

1. Since values and types share the same namespace, it won't be possible to define constructors with the same name as the type. So no more data User = User { ... }. This is called **punning**, where identifiers of the same spelling are used at the term and type level. A solution is to add some prefix to constructors (e.g. Mk). Strictly speaking, you are still allowed to write data User = User {..}, but GHC will warn you in these cases.

2. Several standard primitive types will have different names. If you have DT, [Int] can mean either a list of integers or list storing types with Int as its single element. To disambiguate this situation, when you mean list as a type, you will need to write List Int, and use the [] syntax only for literals. The same goes for other primitive types such as tuples (,), etc.

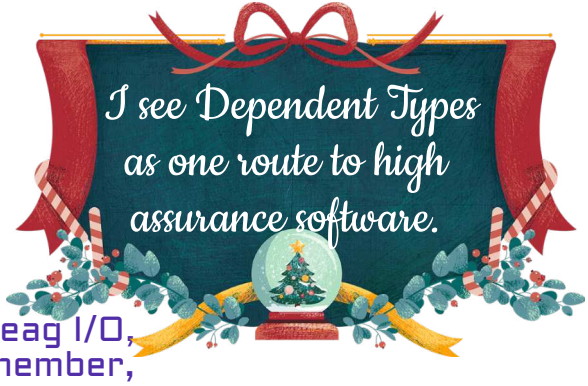
3. DataKinds becomes redundant. The namespace for types and values is the same, so no need to use apostrophe ' to promote constructors.

4. TypeFamilies could be deprecated as well, as you can use ordinary functions in types with dependent types.

5. No need to use the data Proxy a = Proxy type, as you can pass types directly as arguments.

As you can see, these are colossal changes to how we used to write Haskell code. And we definitely will need tools to refactor code, if *ergonomic dependent types* are going to be supported. Not to mention, almost every Haskell tutorial and guide will become outdated or at least confusing.

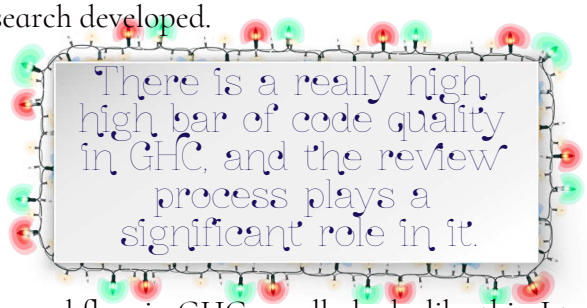
We spoke to **Richard Eisenberg** to learn more about the plans on DT, how the problems are going to be managed and how he sees the future of the proposal.



Principal Researcher at Tweag I/O,
GHC Steering Committee member,
core contributor to GHC

[Q]: *Can you tell us more about your GHC development involvement and your role in the GHC Steering committee? What does the GHC development workflow usually look like?*

Richard: I first got involved with GHC when I was a PhD student. I think I got lucky by being in the right place at the right time and working with Stephanie Weirich at the University of Pennsylvania. She connected me with Simon Peyton Jones, who she had been working with closely during a sabbatical in Cambridge. And that connection then turned into a few contributions to Haskell and Template Haskell, which then grew to more and more contributions around GHC as my research developed.



My workflow in GHC usually looks like this: I tend to spend more time doing larger features in GHC instead of fixing smaller bugs. I wish I had the time to also resolve them, but it is hard to balance it all out. So I usually tend to work on a patch for several weeks. Then it should pass a fairly intensive review process. There is a really high, high bar of code quality in GHC, and the review process plays a significant role in it. It usually clarifies some moments, mostly by paying attention to documentation as well, and it makes sure everything will stand the test of time. Only after we merge it and go to the next task.

[Q]: *So Dependent Types is one of these big features for you?*

Richard: Yes, the Dependent Types will be one of these features. My work recently has been more laying the groundwork for DT than actually implementing them. I've wanted to get to the meat of implementation for four years now. In all truth, GHC is a massive project. And, when we start looking at how this one little feature DT is going to fit, we realise: "Oh, the rest of GHC won't support that until we have the strength in other parts", which is actually good for everyone, not just for DT. That ends up being accumulated into this long-ish diversion until we can shore up that one part after which eventually we'll get back to doing the main core of the work.

[Q]: *Let's talk about the "Ergonomic Dependent Types" proposal. When did you start working on DT? Was the initial goal for it to land into the compiler eventually, or did it start as a research project?*

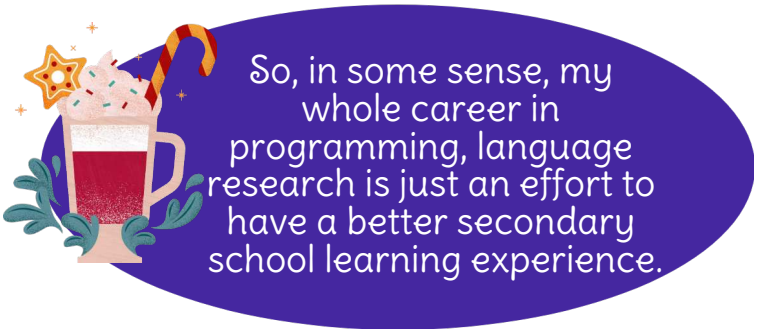
Richard: From the very beginning, the goal was actually to get it into the compiler. And at that time, I don't think anyone thought it was going to take this long. I still feel optimistic that it is going to be right around the corner. And I wish I could tell you why it is taking so long, other than the fact that GHC is colossal and there are a lot of different features to interact with. I've seen comments in the codebase dating from 1994! There is a lot built up over time, and it takes effort to workaround.

My involvement with DT was also essentially from the beginning of my PhD. I wanted to get a PhD in Programming Language Theory. My initial goal was in bringing DT to the masses. Interestingly, my first vision was dependent types in Java actually, before I knew Haskell!

Let me tell you about my motivation around dependent types. Before getting my PhD, I taught Computer Science at secondary school for eight years. And I wanted a way for students to have a richer level of interaction with the compiler. Too often, students would make mistakes, which made them get stuck and frustrated by. As I was the only person to teach and there were maybe 15 students in my class, I couldn't be everywhere at all times. So I wanted the compiler to be able to help me guide students through.

I have this notion that if we could explain to the compiler precisely what we mean through DT, it could help us figure out why our code is wrong. Though, we are decades away from being able to do this. In reality, I didn't know that at the time, I was naive. I thought that DT would solve all the problems back then. I still believe that it will: once we have DT, then lots of time figuring out the right way to use them and the right compiler design. It would be fantastic if in 30 years I can step back into a secondary school classroom and students there are programming with Dependent Types, which allows each of them to work at their own pace in a much better way than I was able to offer them the last time I was teaching.

So, in some sense, my whole career in programming, language research is just an effort to have a better secondary school learning experience. *(*laughs*)*



There are a lot of other fun things that I've had along the way, but in some sense, that's all where it started from.

[Q]: *Currently, ergonomic Dependent Types stands as the "decision" rather than "action", which should only affect the decisions on the direction Haskell is taking in other proposals. But if generally speaking about DT as a feature, how will it look for end-users? Is DependentTypes going to be a single or a set of extensions?*

Richard: I think it would be convenient for users to have one -XDependentTypes extension, that may imply other extensions. It is a little unclear what the best design for that is at the moment. Some people would say that we already have dependent types because we can mimic any program written with DT using singletons. So you could say that we have DT in Haskell already through a variety of extensions and encoding techniques. I don't really agree with that, but in the end, I think having one extension would be nice.

[Q]: *How are people going to use it? How would the interaction of DT code with standard Haskell work? Is it a project-level, module-level or function-level feature?*

Richard: I think it's going to take a learning curve for us as a community to figure out the best way of using DT.

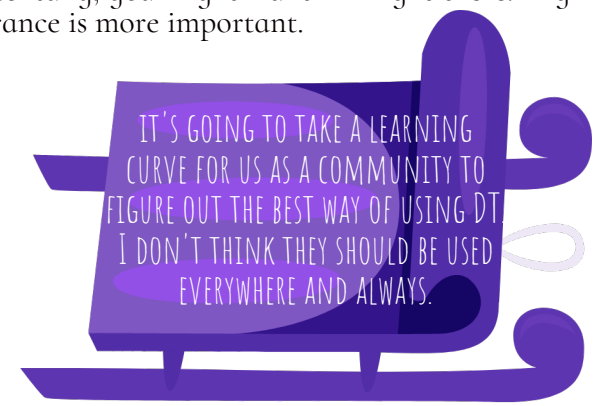
I don't think they should be used everywhere and always. DT are a powerful tool, but they are also an expensive tool. And that means that they should be used just where we need either extra assurance or just where we have some algorithm that is hard to express in a non-dependently typed language.

And so, ideally, we would have a few places, a few key libraries or a few key parts of libraries that use DT and then expose a simply typed interface to make them easier to use and more applicable. And that also answers your question about determining if it's module-level or package-level. Again, I think some of that has to be learned on the ground once we have the Dependent Types. But I expect it to be quite local.

[Q]: *Who would benefit from DT the most? How do you see companies benefit from that?*

Richard: I see dependent types as one route to high assurance software. Instead of writing lots of tests for a certain function or set of functions or a data type, we can imagine using dependent types to be able to mathematically prove that our functions work or our data type maintains its invariants. So when we have a key type or a key set of functions in a library where we want that high assurance for (because we have to do that proof), it takes more time to develop this. That's the time to use DT.

So who would benefit from it? Again, it depends on what a company is doing. If you are writing some Web-based API that serves low-security information, you may not need DT. On the other hand, if you are writing an encryption library where we want to be sure never to mix up on different quantities of different bit widths accidentally, you might want DT right there. High assurance is more important.



[Q]: *DT is quite a radical and heavyweight change requiring many tradeoffs considered to be accepted. Are there plans to implement DT in some compiler fork and have enthusiasts crush-test it there first?*

Richard: I don't think that's necessary because DT are not going to be disruptive in that way. I see that there is a common misconception out there that *"having dependent types"* means *"losing lots of features that we have in Haskell today"*. People will say: *"Oh, you can't have dependent types and type inference in the same language!"*, or *"DT requires termination"*, or something else like that. I don't think either of those things is true. There are a lot of other misconceptions out there also. So in particular, I think every program that we have working in Haskell today, I expect those to continue working. So there wouldn't be a need to fork the compiler. The first versions of DT are indeed going to have a few mistakes. And as we introduce features, invariably we're building something new. Some of these will be design mistakes, which might mean that people who are using these bleeding-edge features the way that we implement at times might need to adapt to frequent changes over releases. The next version might invalidate the code that uses the first version of DT. But none of that should change out of the bedrock of Haskell that is solidified already today.

[Q]: *A quote from the proposal:*

> That would likely lead to some brain drain. I'm aware of a number of active contributors to our community who are excited about the possibility of dependent types. And rejecting this proposal might signal to them that Haskell is not interested in what they have to offer.

Does this mean that the GHC driving force is more focused on preserving some concrete representatives of the community rather than focusing on making it more diverse as a whole?

Richard: I have to say I don't quite see the connection between the idea of this "brain drain" that I mentioned and the diversity goals. Diversity is really, really important. We want to broaden the adoption of Haskell. And, yes, it's true, I won't disavow that comment. I do think that some people in our community are really excited about dependent types. And so if we as a community decided to go away from DT, those people might leave.

But that by itself doesn't say whether or not other people might join. It's not a zero-sum game.

[Q]: *Some folks are already intimidated by Linear Types, Impredicative types and other features. Do you think DT can add to the Haskell intimidation factor?*

Richard: Maybe? But I think it can be mitigated. In my opinion, Haskell has done a poor job of mitigating this intimidation factor.

The idea of *"language levels"* is one thing I've wanted in Haskell for a number of years, and others disagree with me here, so there is room for debate. Another language Racket is the primary example of these language levels. And the idea here is that when you start programming, you give some indication to the compiler of what total language you are programming against. And then the compiler can tailor its error messages accordingly. In a language like Haskell, if we said that we are new to Haskell and then get confused between the term level namespace and the type level namespace, the compiler wouldn't then say, *"Did you mean to enable DataKinds?"* We shouldn't expect a Haskell programmer on their first day of programming to start using fancy kinds. The problem right now is that, yes, we have this extension mechanism that says what portions of the language are components of the input. Still, the extension mechanism doesn't really control what the error messages are. There should be some other way to handle that. I don't know exactly what it is; we have the design work to do here, that emphasises on the questions like *"Where is this user?"*, *"How can we then give error messages that are appropriate for that user?"* And this would decrease the intimidation level overall for Haskell and allow us to do this high-end addition and development without losing that lower-end accessibility.

And actually, part of my research grant for DT that I've been working with is recognising this exact problem. The mentioned part of that research grant is funding a retooling of the error message mechanism within GHC so that we can start to imagine these differing error messages. Right now, our GHC infrastructure is tough to work with around error messages. Therefore, we are going to do a complete overhaul of that, actually in an effort to support better DT, but by giving us better control over error messages. Both will work well with introductory users as well as advanced users.

[Q]: *So improving the error messages is one of the parts to make Dependent Types more approachable?*

Richard: Yes, it is an inescapable part of the feature. I don't think we can keep our current level with DT. With our current interaction story, when you write some code and try to compile it, you can get a full screen of error messages. This is not sustainable. This is not the way it should be. So we need to be able to fix that before being able to add even more features.

> The idea of "language levels" is one thing I've wanted in Haskell for a number of years. And this would decrease the intimidation level overall for Haskell and allow us to do this high-end addition and development without losing that lower-end accessibility.

[Q]: *How do you handle concerns that Dependent Types will fork the community? Is this issue addressed in any way in the implementation plan?*

Richard: All of that community outreach and education is very important. That is definitely going to be the part of this effort. One main challenge is that as the design around DT evolves, those resources will necessarily become out of date. Unfortunately, there is going to be a significant lag between when DT first comes out and when those resources become mature enough to allow people an easy way in. We have to be patient to let things settle. It would be a shame to write a book about DT in Haskell as implemented in its first version. In this case, only after two years, most of that book will become outdated. This is not going to be productive. Instead, what we are going to end up doing is figuring out what the design is, having it settle somewhat. During that time, people will inevitably write blog posts about such. But then, once that is settled, we can do more of that community outreach.

As far as forking and these different designs flavours of language. Yes, that is true that there are certain existing features of Haskell which don't play very well with DT. In particular, separate namespaces. This means that people who want to program with DT may use different type synonyms than others.

Will this fork the language? I don't think so. Everything remains inter-compatible really easily. So modules will be able to import each other. It is just going to come down to what names are used in an individual module. So for a reader of Haskell code, they might have to be aware of both using brackets to denote a list as well as writing the word List to indicate the list type. But beyond that, I don't think it would cause too much trouble. There is more fear around this than there will be trouble in reality.

[Q]: *Maybe the Haskell Foundation can help with one official, maintainable guide?.*

Richard: Yes. I think the Haskell Foundation right now is very concerned with outreach and such. We want to make the Haskell community welcoming to everyone. As far as I know, educational pieces and tutorials don't seem to be something that the Foundation is embarking on in the near future, but maybe it will become a part of it later.

[Q]: *Is GHC in the proper state to implement DT at the moment? Is there a roadmap for bringing in DT? What are the estimations on DT implementation in GHC?*

Richard: I've been burned too many times by giving estimations, so I don't wish to do that. I will say that GHC is in the right state. But, to be honest, that is a hard question to answer. *(*thinking*)*

I have several months worth of patches to write in order to prepare for what we've identified as *"needed to happen before we can start DT"*.

The biggest due change is to use **homogeneous equality** internally. At the moment, GHC depends on **heterogeneous equality**. And we figured out in a paper, that we wrote four years ago, that homogeneous is the better one, but it turns out that it is really hard to make this change.

There is some technical work that we have to do before doing the dependent types. All of this proposal's work is more about designing the surface language, and that is going on in parallel.

If I and maybe some other GHC developers dropped everything else and just did DT, we could probably finish it in 6 to 8 months.

But in reality, it is hard to do that. For instance, there is another major project I'm working on right now – introducing *"lightweight existentials"* into Haskell. That will, for example, permit better integration of Liquid Haskell into the DT story. Liquid Haskell is doing a fantastic job of an easier way of doing verification. And my hope is that the Liquid Haskell story and the Dependent Haskell story can one day meet up and be two parts of the same thing. But this existentials project takes away a significant amount of time from implementing the DT, but it is really important. It is all a matter of priorities. I wish I had more time to recharge.

We are grateful to Richard for his time and for lightening this topic for us more. It is exciting to see the expert's perspective on such an important feature for Haskell.

DT is not a completely new thing. It already exists in some proof-assistant languages. But what we are hoping for with this proposal in Haskell, is that a lot more attention will be paid to where DT would be actually used throughout the whole Haskell community that exists at the moment. We would love to see the committee to base their decision on different points of view. Therefore, we think that some kind of survey for the companies and people that are currently using Haskell in production, in their projects, could help to decide whether this would be a helper or the obstacle for their products. Moreover, we would love to see all the trade-offs considered before the final decision is made, as Haskell is a growing community, so each such step requires a lot of thinking.

The -Wall Street Analytics

GHC options, warnings and flags

Our beloved Haskell compiler **GHC** is indeed an indispensable tool of every developer. It can do so many things for you, and make your life easier in so many ways. The compiler usually provides such features through its options that you can configure for your project. Today we will focus on one part of such, which could help you with export lists, if you ask the compiler politely.

So, without further ado, let's talk about the -Wmissing-export-lists option (GHC warning).

What problem does it solve? In Haskell, all top-level functions, types and classes are exported with the default module header. For example, let's say we have a following module:

```
module MyPackage.MyModule where

data MyType = A | B

myFunction :: ...
anotherFunction :: ..
```

You can see that we do not explicitly say what exactly will be accessible from this module, if you decide to import this module elsewhere. If you do not specify this, then everything is exported in the order of appearance in the source file (this is important for documentation rendering).

However, you can also specify what you export from this module manually. For instance, the exact equivalent of the previous module example will be the following:

```
module MyPackage.MyModule
    ( MyType (..)
    , myFunction
    , anotherFunction
    ) where
...
```

Although, note, that it is recommended to write export lists explicitly, this gives much more flexibility and additional features to your hands. Like in this example of the same module:

```
module MyPackage.MyModule
    ( -- * Type
      MyType (A)

      -- * How to Use
      -- $usage

      -- * Main API
    , myFunction
    ) where
...
```

The export list example above illustrates several features of manual exports in Haskell. First of all, it allows you to restrict your exports: see how we allow to export only one constructor or only some functions. Also, this aligned export list separates all functions and types by sections and provides additional metadata: headers, section names and arbitrary documentation using named documentation chunks. All export list documentation is written using Haddock syntax (Haskell documentation tool).

As you can see, you can do quite a lot of things with export lists, so you might not want to miss them. So, if you enable -Wmissing-export-lists warning (which is a part of -Wall already), you'll get warnings when your modules don't have explicit export lists.

And there's a good reason to write them due to multiple advantages:

- You can have more low-level control of what you are exporting (e.g. you can provide smart constructors instead of exporting the whole data types).
- You have more control over the generated documentation: order of functions, section names, named documentation chunks to explain topics better, etc.
- You have a clear and explicit separation between private (internal) and public interfaces.
- Your code can run faster with explicit lists, because GHC optimizes internal functions more efficiently on average.

As you can see, the quality of code can be improved significantly, and we strongly recommend enabling -Wall and utilise the -Wmissing-export-lists sanity check!

DERIVING ALL THE WAY

Deriving through newtype
In a one-line JSON class
Using the Generic type
DerivingAnyclass
Removing boilerplate
Making syntax bright
What fun it is to generate
All instances tonight

Deriving Show, deriving Ord
Deriving all the way
Oh, what fun it is to derive
In a one-line all of them

The **deriving** mechanism in Haskell is so much like Christmas! Look for yourself, what is the best gift for Haskell developers, if not another way to reduce boilerplate and write even more maintainable code? Fortunately, Haskell Santa compiler brings a wholesome bag of awesome deriving features! Haskellers don't need to write tons of boilerplate and error-prone instances, Santa's Elf helpers can do this for us, freeing us from doing tedious work, and allowing us to enjoy Christmas holidays without bugs in production during this festive season!

So no way we leave our festive edition of Bind The Gap without deriving. Our present is going to be a recap on all deriving news through 2020.

Kowainik started preparing for Christmas in advance, and earlier this year we wrote a comprehensive guide about deriving in Haskell, which is also in Christmas thematic by the way. In our detailed post, we described everything you need to know about deriving in Haskell, categorised, explained and compared deriving mechanisms and strategies, all with real-life examples and best practices for deriving.

Deriving is a giant piece of work, so let's start reviewing it from small parts.
In our guide, we provide an analogy between typeclasses and letters to Santa:

When a little child writes a letter to Santa, they describe what toy they want, e.g. cute, plush, cartoon-character-alike. These toy properties can be interpreted as typeclasses. They exist independently from the actual toy. At the same time, you can characterise a toy like a plush bear by different qualities: plush, animal, smiling, etc. A letter to Santa is a function implemented in terms of some typeclasses and, depending on toy characteristics, Santa will choose a perfect match for a kid.

Continuing with this analogy, GHC (the Haskell compiler) is the Elf workshop. When using deriving, you only need to specify what you want in a declarative way. And Elves will do all the boring work for you. Though, you can give some instructions to Elves about how to do their work in the form of deriving strategies (there are four of them):

stock — create new toys using standard factory parts

newtype — copy the toy from your elf colleague

anyclass — throw all toy parts on the conveyor, hoping that somebody else knows how and will assemble them

via — take inspiration from other similar toys, but do on your own

Deriving strategies give you more control over how you want your typeclasses to look like, and we recommend to **always specify strategies explicitly**. This is especially relevant when you have lots of newtypes and tons of typeclasses.

GHC 8.8 (currently the most popular GHC version according to 2020 State of Haskell Survey results) introduced the -Wmissing-deriving-strategies warning, that warns when strategies are not written explicitly. Later, GHC 8.10, released in 2020, implemented the -Wderiving-defaults warning (enabled by default, even without -Wall) that warns about possible strategy ambiguity in the presence of the GeneralizedNewtypeDeriving and DeriveAnyClass extensions. We can say with confidence that GHC encourages Haskell developers to consider deriving strategies in their code.

The whole existing deriving machinery wasn't planned beforehand and was implemented on top of an already existing basis. It grew naturally from the needs of different people and their ideas. Kids want to play with more and more toys, and Elves need to be able to build everything! So it is not a surprise that different syntax constructions can simulate some semantically equivalent features. In our guide, we've noticed that the DeriveAnyClass and DefaultSignatures extensions can be simplified in favour of DerivingVia. Later, *Matt Parsons* noticed this property as well in a blog post about simplifying deriving, and he went even further in these reasonings. Matt proposed several more interesting ideas of the deriving simplification. Most of the ideas are based on the fact that the DerivingVia feature is very powerful and can substitute many existing Haskell features.

The **via** construction turns out to be extremely powerful, even outside deriving. *Baldur Blöndal*, the main author of the DerivingVia proposal, also authored the ApplyingVia proposal, that allows using the behaviour of different newtypes easier in more places. The DerivingVia extension itself was attempted to be improved by allowing underscores in the deriving clause. It is interesting how some features of reducing boilerplate give birth to other innovative ideas.

Being able to derive boilerplate is so convenient, so people do this a lot. 2020 was a year when people heavily used DerivingVia everywhere, even in an unexpected way. And it turned out to be quite handy and exciting. Check out these ideas:

```haskell
newtype MyTime = MT { lt :: LocalTime }
    deriving Binary via
        Time "%M/%d/%y %H:%M:%S"
```

```haskell
data Person = Person
  { age  :: Int
  , name :: String
  } deriving ToJSON
      via Override Person
            '[ String `As` CharArray
             , "age" `As` Decimal
             ]
```

```haskell
newtype Tile a = Tile
    { runTile :: Double -> Double -> a
    } deriving stock (Functor)
      deriving Applicative
        via (Compose
               ((->) Double) ((->) Double)
            )
```

```haskell
data User = User
    { userId :: Int
    , userFullName :: String
    } deriving Generic
      deriving (FromJSON, ToJSON)
        via JSON '[ StripPrefix "user"
                  , SnakeCase ] User
```

Haskell has many ways of enabling developers to focus more on solving real problems and doing less tedious work. Especially when it comes to boilerplate instances of typeclasses: you can use deriving, various deriving extensions, TemplateHaskell or Generics. Of course, there is a separate question of trade-offs for all these approaches. You can write typeclasses by yourself, ask Santa's Elves, or outsource this problem to Elves from the South Pole. But it is nice to have different options for solving various issues, and we are looking forward to the next year, waiting to see what other exciting news deriving will bring to us!

WHINE Sommelier

~Degustate different libraries and check them against our appetite*~

Time is an integral part of our universe. No surprises that in programming, libraries to work with time are also essential.

Every programmer works with time data types daily in almost every program created for users. Work with time is a fundamental part of many algorithms; time itself is a crucial piece of most applications. That is why it is so important to have a good tool to help you work with time, as everyone knows, it is a challenging task itself!

In Haskell, the most common library to work with time is called **time****. So, today, let's review this library and test how the standard choice for time in Haskell passes the test of time and comfortability.

time is a *boot library*, meaning that it comes with the GHC installation, and you don't need to wait for it to build when depending on the package and using the version bundled with GHC.

time provides data types to get the current time, parse and format timestamps, work with UTC timestamps, time zones, parts of a timestamp such as year, month and day, and handle different calendars.

We have been using **time** a lot in different applications and projects, so we know a lot about the library in the battle. Let's look at **time** under different angles and see how suitable it is for production needs. We are going to examine the latest version from Hackage at the moment, which is 1.11.1.1.

* https://bit.ly/btg-lib
** https://hackage.haskell.org/package/time

Documentation

Since **time** is the standard library that comes with the compiler and is maintained by the dedicated people of the Haskell Libraries Group, you may assume that the documentation should be the best in the class and Haskell generally. Having such high expectations, let's look mindfully at the existing documentation around the package.

Both README and .cabal header don't contain much information and provide a poor overview of the package. The documentation's main source is Hackage with the module-based documentation; there is no central docs web page, which is not convenient for the standard and community-representing library. The top-level module Data.Time contains some descriptions of types used in the library, but that's about it. There are no pointers to this place anywhere, so if you want to know how to use the library, you need to manually browse through modules on Hackage, read Haddock, and try to guess what module would have the information you would like to find. And even so not all exported functions are documented with Haddock.

There are several searchable high-level resources about **time**, such as the wiki page about **time** and A cheatsheet to the time library [1]. StackOverflow also contains multiple answered questions, e.g. how to get the current time with the time zone. So it is possible to learn about **time** through different resources outside the library.

Still, it would be nice to have usage examples of each function in Haddock and even test them with **doctest**. Moreover, @since annotations are almost not present. They are quite valuable for **time** because it is a boot library that comes with GHC, so it would be beneficial to see with which version of GHC each function comes when you read the API.

time is not the only library to work with time data types, but it doesn't contain a comparison with other libraries. As a boot library, maybe it shouldn't have, but it would be nice to have lots of links to such resources in the official place. Fortunately, there's a blog post [2] that gives a quick overview of three Haskell libraries to work with time types.

Ease of use

Being the standard time library, we expect the interface to be battle-tested and refined to its best version possible. Let's check how easy it is to have time as part of your equipment during project development.

Since time comes with GHC, you don't need to install it separately, and it doesn't consume any time during compilation when used. You can even work with time in GHCi without the need to configure anything. This is very convenient for testing and even for personal usage (to count days or weeks to some period of time in the future, get the difference between two timestamps in seconds, etc.)!

Lots of different modules are provided that are responsible for various concepts and time representations. You can export the Data.Time module qualified to get almost everything time-related. Though there are some data types and functions that share names, so can't be reexported simultaneously. That means that you need to browse modules to get everything you need. And module names are not always obvious to help you with that. For instance, the UTCTime data type comes from the module Data.Time.Clock. Though functions and types from time rarely have conflicts with identifiers from other libraries.

The library provides a maintainable minimal changelog, but it doesn't include the migration guide. And breaking changes are happening from time to time (who remembers the painful transition to 1.8, put your hands in the air).

Generally, it is not always trivial to convert to time types. For example, if you want to convert seconds to NominalDiffTime, you need to work with type Pico, which comes from base's Data.Fixed, and is not that common.

On the bright side, because **time** is the standard library, other libraries (databases, encoding and decoding, serialising, web APIs, testing) already provide integration with it, and this doesn't cost anything in terms of dependency size. While using a different time library requires bringing in an entirely different ecosystem or writing a lot of conversion functions manually.

[1]: https://williamyaoh.com/posts/2019-09-16-time-cheatsheet.html
[2]: http://bit.ly/3-time-packages

Maintenance

To its credit, the library is maintained on a high level, as you expect from the community's standard library.

Looking at the library's source code hosting page (which is GitHub), we can say that it is managed well. Pull requests are being reviewed and either accepted or at least answered. There are no stale patches that require attention. User requests and reports are also taken care of. All issues in the tracker receive responses very fast. We can say that maintainers take care of the library backlog promptly.

The library follows *PVP* and understands well that the consequences of any mistakes on this front are potentially disastrous due to the ubiquitous usage of the library. The project's CI integration checks its work with GHC versions back to 8.0.2. All top-level fields in the .cabal file are filled, and you can easily jump from Hackage to sources and issue tracker. Each release is tagged on GitHub and accompanied with the relevant changelog entries.

It would be very helpful if the library README contained badges to go to CI and Hackage for convenience. But that's a very minor detail that can be easily fixed. Great work on the maintenance side!

Code quality

Looking through the code in the **time** library is a pleasant journey. We can recommend this package source code to those who enjoy learning idiomatic Haskell code through reading the project's code.

The code in the package is easily readable and formatted prettily. time compiles with -Wall without any warnings produced by GHC, though no additional warnings are enabled besides -Wall.

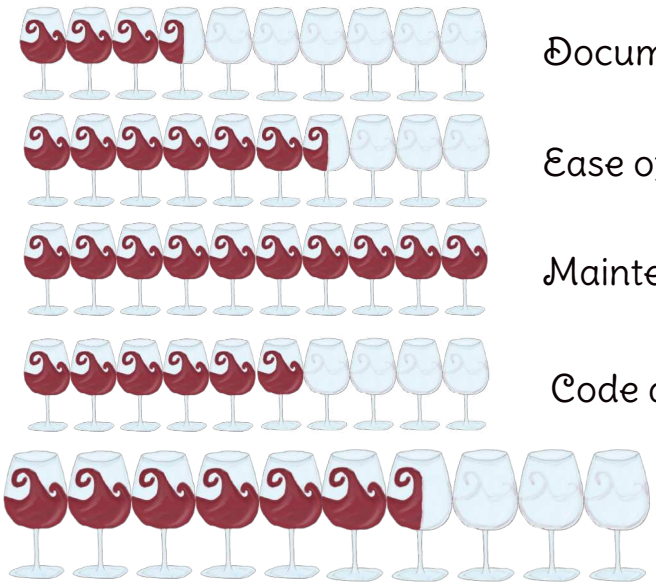
However, time is not taking the opportunity of code quality improvements by the tools like **HLint** or **Stan** — it contains 187 HLint suggestions (checked with hlint-3.2.3), 105 Stan observations (checked with stan-0.0.1.0) and even several Haddock warnings.

Some warnings are not so fearful, e.g. usage of space leaking functions foldl, sum, many lazy data type fields. But there are also usages of partial functions (fromJust, !!, last, init, read, undefined, Enum methods), as well as some missing explicit constructors instead of underscores during pattern matching.

The library implements a massive test-suite, including unit and property-based tests. But, as mentioned before, no **doctest** and code examples in Haddock.

Summary

The **time** library is quite powerful and provides a lot of useful features. It's a go-to library to work with time types. However, there are valid situations in which you are better using an alternative library for time in Haskell. For example, high-performance requirements or type-safe time units usage. For that, you would need to analyse your requirements against the library on your own and manually try to find the better option for you, which could be challenging, as there are no good reference descriptions in the defacto lead library for time about the pros and cons. The documentation for **time** could use some help, so we encourage everyone to submit patches with documentation improvements. It is a great opportunity to help the Haskell community!



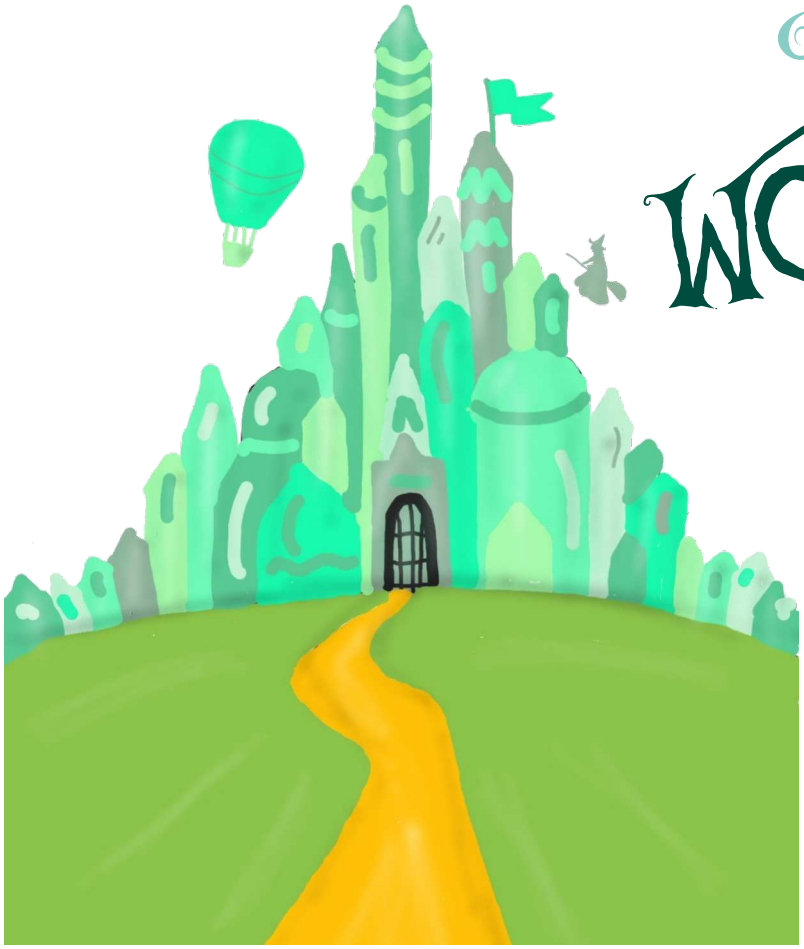
Documentation: 3.5 / 10 (Poor)

Ease of use: 6.5 / 10 (Good)

Maintenance: 10 / 10 (Best-in-class)

Code quality: 6 / 10 (Good)

Summary: 6.5 / 10 (Good)



The Wonderful Wizard of OS

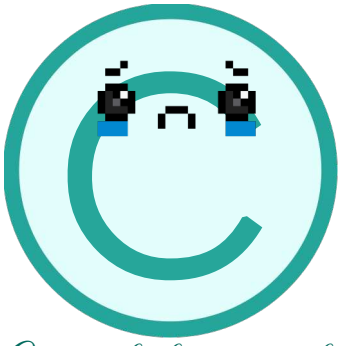
While people all around the world are counting days to the New Year and excited about what the next one will bring to their lives, Open Source maintainers are looking at this with a bit of sadness in their eyes. The New Year also means that all the copyright notices should be updated in all tools and projects that they provide, which doesn't sound like good entertainment for Christmas and New Year Eve. Luckily, there is a way to bring joy even into our lives! **Vaclav Svejcar [1]**, also an OS maintainer, created a tool called **headroom [2]**, which helps with the licensing headers in general. Thanks to this project, developers can breathe out and spend the holidays the way they want, leaving the headroom to deal with these problems.

Headroom is a CLI tool for updating contents of any source code files by introducing/updating file headers with the unified and configurable format. Such headers contain the license and copyright data, as well as some metadata information. You can use headroom to bring headers to each file, introduce consistent metadata fields across your project, update copyright years automatically, and so on. So it is quite handy to have such a flexible tool that can maintain all this boring information for you.

We already use headroom in our projects in Kowainik. Here is the file headers produced automatically by headroom example from Summoner, our Haskell project scaffolding tool:

```
{- |
Module                  : Summoner.GhcVer
Copyright               : (c) 2017-2020 Kowainik
SPDX-License-Identifier : MPL-2.0
Maintainer              : Kowainik <xrom.xkov@gmail.com>
Stability               : Stable
Portability             : Portable

Contains data type for GHC versions supported by Summoner
and some useful functions for manipulation with them.
-}
```



The great news is that the tool supports different programming languages (Java, Scala, C++, Rust, HTML, etc.). So you can recommend it to anyone regardless of their choice of the programming side or language. Anyone can install and use it on the project of their choice through the terminal.

The tool is very flexible and configurable, which is very convenient as you can either use the standard template for your language or, if you already have a working scheme you use for your projects, you can translate it into the template and apply it across the projects. You can specify your desired header format with Mustache templates. Headroom reads Mustache variables and other settings from YAML configuration and provides some additional variables as well for your convenience.

Headroom itself is written in the functional style, using Haskell language. It uses a bunch of common libraries for fulfilling its functionality:

- **optparse-applicative** for CLI interface that helps to provide a nice interface with useful help pages
- **yaml** and **mustache** for configurations
- **pcre-light** and **pcre-heavy** for regular expressions
- **time** for copyright years managing
- **file-embed** for embedding data files at compile-time and making the distribution easier

Internals are using rio custom prelude with a bunch of other helpful libraries. The project is also heavily supplied with tests and documentation.

You can install headroom easily in multiple ways: download binary from GitHub releases, use brew or build from sources.

So what are you waiting for? Give it a go and update all your projects as it is just about a time!

[1]: https://svejcar.dev/
[2]: https://github.com/vaclavsvejcar/headroom

by Chris Penner

Hi folks and welcome to Lensalot!

Today we'll be chatting about indexed optics. This tutorial is more of a sampler than any sort of comprehensive guide, but hopefully it introduces something that's new to you smile. For this article I recommend you already have an understanding of lenses and folds before diving in.

Indexes in optics are a sort of *"expansion pack"* to all the regular lensy things you're used to. Indexes enhance existing lenses, traversals and folds by allowing you to track information about your position within a structure as you dive deeper into it.

This information can be anything that you care about, but it's very often used with data structures that already have an inherent notion of **indexing**. For example, sequences like lists have indexes, values in Maps have **keys**, values in a tree can be identified by a unique **path**!

Most optics libraries support some form of indexed optics, with the notable exception of microlens. In fact, it's likely you've already used some indexed optics without even knowing! Indexed optics intermingle freely with *"normal"* optics but provide additional functionality when you ask nicely for it.

Here are some imports you'll need as we walk through this post:

```
import Control.Lens
import qualified Data.Map as Map
```

Let's look at a combinator you might have seen before. traversed is a Traversal which allows you to focus on each member of a Traversable container. We can see how it works on a few traversable structures by collecting a list of its focuses:

```
>>> toListOf traversed [0, 1, 2]
[0, 1, 2]

>>> toListOf traversed
  (Map.fromList
    [ ("Haskell", "Functional")
    , ("C", "Imperative")
    , ("Scala", "???")]
  )
["Imperative", "Functional", "???"]
```

But did you know that traversed is actually an **indexed** optic? That's right!

```
traversed
  :: Traversable t
  => IndexedTraversal Int (t a) (t b) a b
```

The type shows us that it's an IndexedTraversal with an Int index, it keeps track of the numeric index of each element it focuses! This behaviour is completely ignored in the previous examples, in fact the lens library uses some clever tricks to ensure that indexes aren't even computed unless they're used. We can collect the index of each focus along with the element itself by using itoListOf instead:

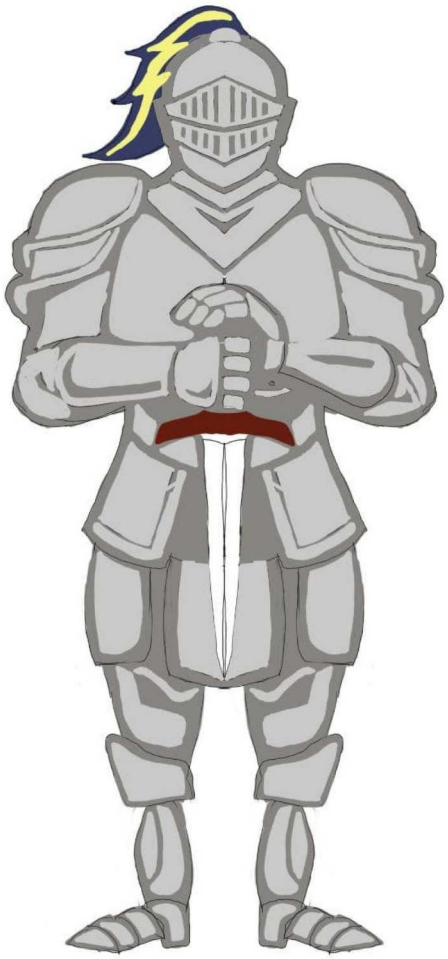
```
>>> itoListOf traversed [0, 1, 2]
[(0,0), (1,1), (2,2)]

>>> itoListOf traversed ['a', 'b', 'c']
[(0,'a'), (1,'b'), (2,'c')]

>>> itoListOf traversed
  (Map.fromList
    [ ("Haskell", "Functional")
    , ("C", "Imperative")
    , ("Scala", "???")]
  )
[ (0,"Imperative")
, (1,"Functional")
, (2,"???") ]
```

Due to the way optics inherit from one another, itoListOf works with any of these signatures:

```
itoListOf :: IndexedGetter     i s a -> s -> [(i, a)]
itoListOf :: IndexedFold       i s a -> s -> [(i, a)]
itoListOf :: IndexedLens'      i s a -> s -> [(i, a)]
itoListOf :: IndexedTraversal' i s a -> s -> [(i, a)]
```



And if you prefer to use operators, you can try the infix version (^@..).

Tracking the numeric index is great and all, but for a Map of keys and values we'd really love to use the key as an index instead! Not to worry, for that we've got itraversed which is a bit smarter. The lens library exports a TraversableWithIndex typeclass for traversable types that have some kind of index associated with them. Most types you could want are implemented for you already, but if you've got your own datatypes you can implement an instance yourself.

Let's see how it differs from its simpler cousin traversed.

```
-- lists ALWAYS use numeric indexes
>>> itoListOf itraversed [0, 1, 2]
[(0,0), (1,1), (2,2)]

-- Maps use keys as their indexes
>>> itoListOf itraversed
  (Map.fromList
    [ ("Haskell", "Functional")
    , ("C", "Imperative")
    , ("Scala", "???")
    ])
[ ("C", "Imperative")
, ("Haskell", "Functional")
, ("Scala", "???")
]
```

The ideal index for lists is still an Int, so that hasn't changed, but we see that the Map now provides the key alongside each value. At first glance it doesn't appear to provide much added value over using something like Map.toList, but have faith! Indexes start to pay off when we start to look at more complex deeply nested values and containers.

Consider this map of owners to their pets:

```
pets :: Map.Map String [String]
pets = Map.fromList
  [ ( "Jon"
    , ["Garfield", "Odie"]
    )
  , ( "Charlie"
    , ["Woodstock", "Snoopie"]
    )
  ]
```

It's easy enough to call Map.toList to get a list of owners and their pets, but what if we want to normalize the data? For instance, let's say we want to print out each pet alongside their owner. Since both maps and lists are traversable we could focus the pets using traversed . traversed but we'd lose track of who owns which pet. We need to *keep track of some context from higher up in our optics path*; exactly what indexed optics are good at!

The first step is to use itraversed in our first step so we track the keys of the outer Map; then we need to indicate that we want to keep the index of the first itraversed and ignore the index of the traversal over the list. The lens library provides combinators like <. and .> which allow you to keep the index from one combinator or another as you compose optics. This means we can dive into our list of pets while still keeping track of the owner:

```
>>> itoListOf
      (itraversed <. traversed)
      pets
[ ("Charlie", "Woodstock")
, ("Charlie", "Snoopie")
, ("Jon",     "Garfield")
, ("Jon",     "Odie")
]
```

Now we can easily print out each combination:

```
>>> itraverseOf_
      (itraversed <. traversed)
      (\owner pet -> putStrLn $
        pet <> " belongs to " <> owner
      )
      pets

Woodstock belongs to Charlie
Snoopie belongs to Charlie
Garfield belongs to Jon
Odie belongs to Jon
```

If we like we can keep track of **both** the Map's index and the list's index; the <.> combinator will keep the indexes to its left AND right by pairing them up as a tuple.

```
>>> itraverseOf_
      (itraversed <.> traversed)
      (\(owner, num) pet -> putStrLn
        $  pet
        <> " belongs to "
        <> owner
        <> " | Pet #"
        <> show num
      )
      pets

Woodstock belongs to Charlie | Pet #0
Snoopie belongs to Charlie | Pet #1
Garfield belongs to Jon | Pet #0
Odie belongs to Jon | Pet #1
```

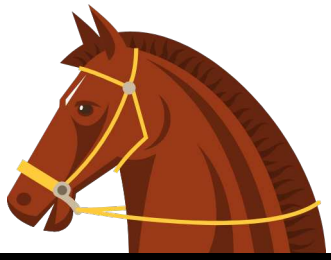
There are many tools in the lens library which make use of indexes, for instance we can use elemIndicesOf to list all owners who have a pet named "Garfield":

```
>>> elemIndicesOf
      (itraversed <. traversed)
      "Garfield"
      pets
["Jon"]
```

We can find the indexes of the first 10 even fibonacci numbers:

```
>>> fibs = 0 : 1 :
      zipWith (+) fibs (tail fibs)
>>> take 10 $ findIndicesOf
      traversed
      even
      fibs
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Or perhaps we want to truncate a paragraph, but want to ensure we only do so at a valid word break. We can use indexed combinators to make smart choices which depend on both the index of a character and the character itself.

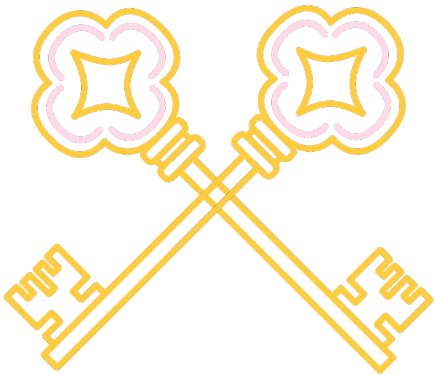


```
smartTruncate
  :: Int
  -> String
  -> String
smartTruncate numChars text =
  -- Take characters until we hit our
  -- size limit, then continue until
  -- a word break
  (text ^.. itakingWhile
    (\n c ->
      n < numChars
      || not (isSpace c)
    ) traversed
  )
  <> "..."

>>> smartTruncate 15 "A spoonful of
sugar helps the medicine go down"
"A spoonful of sugar..."

>>> smartTruncate 10 "This is
supercalifragilisticexpialidocious
isn't it?"
"This is
supercalifragilisticexpialidocious..."
```

This is only a small taste of what indexed optics can accomplish, and of course most examples that are small enough to be helpful are also easily accomplished without optics. Trust me that there are dozens of other uses which dove-tail nicely with many other common tasks in the wild, and they'll start to appear once you know how to find them. I hope this helps make the world of indexed optics just a little less intimidating.



RANK'N'ROLL

In this edition of RankNRoll, we want to highlight 10 blog posts (in no particular order) that rocked 2020. We added to this list write-ups that we find enchanting, curious, useful, maybe not receiving enough deserved attention, or generally fascinating. This year, there are so many of them, but this list is just a part of variegated posts, which hopefully can help everyone find what hit the spot.

Let's Rank'N'Roll, baby!

1

The year 2020 for Haskell started with the blog post by **Stephen Diehl** called *"Haskell Problems For a New Decade"*. It set the tone for the upcoming future by highlighting several problems requiring some attention for improving the Haskell ecosystem. And, indeed, during this year already we saw movements towards many of them: the delimited continuations primops GHC proposal, HLS and VSCode plugin, multiple proposals related to dependent types, etc.! Read the full blog post for inspiration for your next project, and who knows, maybe you will be the person to solve one of the challenges!

3

Application structure and Design patterns are hot topics in Haskell this year, which is a big step in the right direction. And **Felix Mulder** shares his views on the subject in the *"Revisiting application structure"* post. The blog post describes several widespread approaches to structuring subsystems in the big application and provides a comparison of trade-offs. It is an exciting read for everyone wanting to learn how to design bigger applications in Haskell.

2

Olle Fredriksson wrote an interesting blog post named *"Speeding up the Sixty compiler"*. The blog post describes multiple things that helped improve the Sixty language compiler's performance. But the interesting thing is that the suggestions are quite general, and can be applied to any project: web-backend, CLI tool, decoding library, etc.

4

Debugging in Haskell is very different from other mainstream languages. Due to purity and laziness, Haskell enables an approach called equational reasoning. **Gil Mizrahi** describes this approach in the post called *"Substitution and Equational Reasoning"*. It is a very powerful technique! Fun story: when we were working on the prolens library that provides Profunctor encoding of optics, we've implemented one of the primitive operations wrongly, and we've used equational reasoning to debug our implementation.

Rank'n'Roll

5

Haskell has a very long story. It is a 30-years old language! A lot of things happened to Haskell within this time period. **Type Classes** compiled a list of all significant milestones and great moments in the *"Haskell Timeline"*. It is a colossal work, and we are very grateful for this fascinating journey for everyone who wants to dive into the history of the Haskell language.

8

Vidar Holen shared with us *"Lessons learned from writing ShellCheck, GitHub's now most starred Haskell project"*. It is always enjoyable to read about the journey of such noticeable and huge projects! ShellCheck brings joy to many developers, and the story behind such a project contains a lot of interesting details, both technical and personal.

9

One of the Haskell distinguished features is purity by default. But IO is also a powerful tool, let's not forget that! **Alejandro Serrano** describes in detail *"The power of IO in Haskell"*. It is an excellent guide about the IO machinery, exceptions, asynchronous and multithreading computations, resource pools and streaming. We recommend reading it to everyone!

6

Getting started with Haskell can be challenging. It is an entirely new world! And the UX is not always great. **School of FP** wrote several guides to help beginners, starting from *"Setting Up Haskell Development Environment: The Basics"*. It is a series of three blog posts that explains Haskell toolchain, and how to start with both build tools cabal and stack. The narrative way makes it interesting even for experienced users!

10

Our own blog post *"Haskell mini-patterns handbook"* received a lot of attention this year, and we are grateful to everyone supporting our work and sharing it with everyone! It is good to know how to structure the application in Haskell on a high-level, but it is also helpful to be aware of some small programming patterns that solve lower-level problems. And our mini-patterns handbook contains nine such examples, that proved to be used by every FP developer.

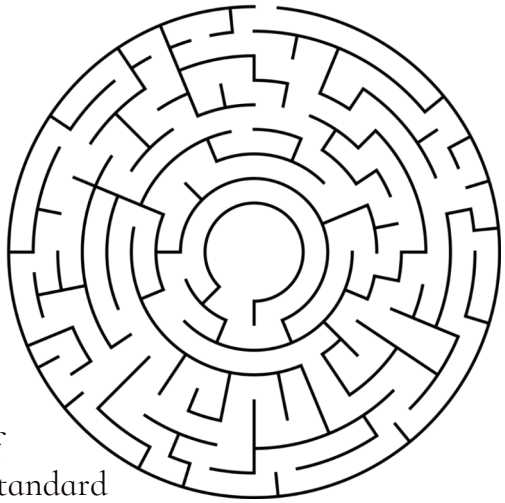
7

Haskell is infamous for being not so well supported on Windows. The situation is improving, and one of the Haskell Foundation's top priorities is to fix Windows support for Haskell. But as developers, we all can play a role in providing smooth UX for using our Haskell tools on Windows. **Iori Matsuhara** wrote helpful instructions on how to *"Create a Windows installer for your Haskell project"*. It is great to see the accessible instructions on providing something magical!

Of course, that only a tiny portion of all the awesome content written by the amazing community during this year. **Haskell Weekly** publishes dozens of blog posts each week. People write on very different topics, touching very diverse parts of the ecosystem. It is captivating how the community evolves and how many improvements are happening at this exact moment. It is a pleasure to be a part of such a lively and active community!

Overcoming stereo types

Section with explanations of some advanced concepts and type-level programming



Haskell has many ways to reduce boilerplate, and some of them require usages of advanced features. However, not all Haskellers recommend doing that and accept the tradeoffs that such type-level features carry. And a lot of troubles are coming from the fact that it's not always clear *when* to use such features. In this section, we want to demonstrate some ways to use them efficiently, showing the best ways to apply advanced techniques on real-life examples.

This time, let's review one problem similar to the one we were solving for production. The situation is the following. We need to query an external REST API. This API provides multiple endpoints for querying data types of the same structure but under different names. The response is always a JSON object with a single key and array of integers. But the key name is different for different REST endpoints. For example, here are two samples of endpoints and results they return:

```
-- /data/ranks/values.json
{ "data-ranks": [42, 10, 15]
}

-- /data/keys/values.json
{ "data-keys": [7, 3]
}
```

We can solve this problem easily by writing some boilerplate: duplicate data types for each endpoint and each type of response and write manual, repetitive JSON instances. But, as with any boilerplate, this approach becomes tedious if we need to add and remove such data types periodically or test our integrations.

Here we want to present a solution using type-level strings. Required Haskell extensions for the solution: DataKinds, ScopedTypeVariables, TypeApplications.

Haskell allows having string literals as part of types. The base standard library provides such type-level strings as a kind (type of a type) Symbol in addition to the typical kind Type that most of the basic data types have. Symbol is an opaque data type promoted to the kind-level, one of the few kinds that base provides naturally.

base also provides handy functionality to work with type-level strings. You can perform some primitive operations with them on the type level using exported type families, e.g. comparing type-level strings. But you also can convert those type-level strings to runtime values. This makes sense because if the value is known at compile-time, you definitely can have it in the runtime as well.

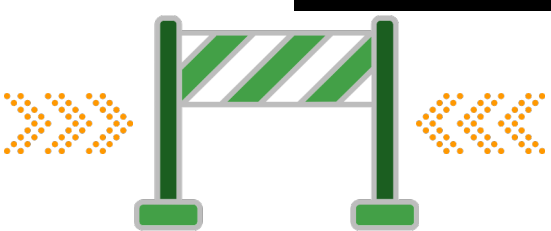
So, for our solution, we are going to introduce a newtype with a type-level string as a phantom type parameter:

```
newtype ResponseData (path :: Symbol) =
ResponseData
    { unResponseData :: [Int]
    }
```

This data type is a wrapper around the list of integers, and it stores the relevant dynamic part as a type-level tag. So in our particular case, we will work with values of type ResponseData "ranks" and ResponseData "keys" to represent the API responses.

Next, we want JSON instances to depend on this type-level string. Fortunately, this also can be done! We are going to achieve this using the symbolVal function that allows converting type-level strings to runtime strings:

```
symbolVal
    :: KnownSymbol n
    => proxy n -> String
```



You see that this function imposes the KnownSymbol constraint for "n". This constraint allows us to reflect compile-time strings to runtime. As always, we can play in GHCi to see how this function works:

```
ghci> :set -XDataKinds
ghci> :set -XTypeApplications
ghci> import Data.Proxy
ghci> import GHC.TypeLits

ghci> :t Proxy
Proxy :: Proxy t

ghci> :t Proxy @"some string"
... :: Proxy "some string"

ghci> symbolVal
        $ Proxy @"some string"
"some string"

ghci> :t symbolVal
          $ Proxy @"some string"
... :: String
```

And here goes our FromJSON instance:

```
instance
    KnownSymbol path =>
    FromJSON (ResponseData path)
  where
    parseJSON = withObject
      ("ResponseData " ++ pathStr)
      $ \o -> do
        let topKey =
              "data-" <> toText pathStr
        values <- o .: topKey
        pure $ ResponseData values
      where
        pathStr :: String
        pathStr = symbolVal
          $ Proxy @path
```

Now, that's a lot going on here! Let's disassemble this FromJSON instance definition:

1. We define a single instance for ResponseData path with the KnownSymbol constraint for "path" to get type-level strings in the runtime.
2. In the pathStr helper, we use the symbolVal function to pass Proxy parameterised by our path type variable (using TypeApplications to specify type variables)
3. Now, having our String value, we can concatenate it with the "data-" string or other things to get our relevant parts of API.
4. The rest is the standard FromJSON instance.

That is already useful, but it's not all we can get from having type-level strings! We are also using servant-client to query API, so it is possible to define a single API only once, and then just substitute relevant parts with type-level strings:

```
type DataValuesApi (path :: Symbol)
    =  "data"
    :> path
    :> "values.json"
    :> Get '[JSON] (ResponseData path)
```

To summarise this approach, when implemented to solve the real problem, it gave us the following benefits:

1. **Zero boilerplate**. We only needed to write instances and endpoints querying once. And only specify missing parts on usage.
2. **Great maintainability**. The client API wasn't changing, so we never needed to patch JSON instances. It was enough to implement the logic, write tests and forget about this problem. Whenever we wanted to query another type of data (or remove some existing type), it was only a two-line addition (or deletion) in the code.
3. **Easy migration**. When we needed to query an API under a different path, it was pretty straightforward to migrate the existing scheme: just change the names.
4. **Newcomer-friendly**. Since the underlying implementation is rarely changing, and all implementation details are hidden behind the type-level API, newcomers can easily add new types to the query by copying existing usages of this API. But, of course, we also write documentation to our code, and this helps people as well.

Using advanced features in Haskell can be challenging, but we hope that our guides can help with using them more efficiently!

Challenge: implement the ToJSON instance for ResponseData that satisfies the roundtrip property.

Pure Gold

by Impure Pics

DO NOTATION

DO NOTATION VERSION

```
main = do
  putStrLn "Tell me something"
  s ← getLine
  let revS = reverse s
  putStrLn $ "Reversed: " ++ revS
```

DESUGARED VERSION

```
main =
  putStrLn "Tell me something" >>
    getLine >>= \s →
      let revS = reverse s
      in putStrLn $ "Reversed: " ++ revS
```

`let revS = reverse s`

NEW DEFINITION
BOTH SIDES ARE INTERCHANGEABLE

DESUGARED TO

```
let {definition}
in {rest of the code}
```

`s ← getLine`

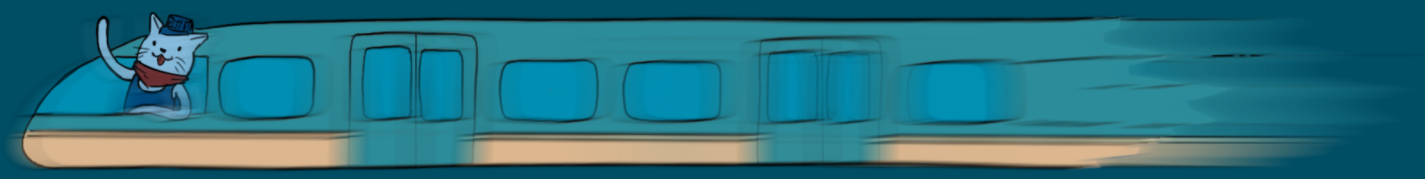
BINDS THE RESULT OF THE COMPUTATION TO THE NAME

DESUGARED TO `>>=` PASSING THE RESULT INTO THE LAMBDA `(\s →)`

`putStrLn "Tell me something"`

A REGULAR MONAD ACTION
DOESN'T CREATE A BINDING

SEQUENCED FURTHER WITH `>>`



Impure PICS

FP Artist & Content maker

Distilling functional programming for the good of all

BASELINE

Interesting parts of the base standard library

Sorting algorithms is one of the favourite topics on many whiteboard job interviews. However, in real life, you don't need to implement it by hand. You should be able just to take one such magic function that sorts it out for you, and apply it where you need.

Though, the standard Haskell library base contains several functions for sorting lists. Even if knowing algorithms under hood is not required, you still need to know about all of these provided functions and their pros and cons to decide on the best choice for your case.

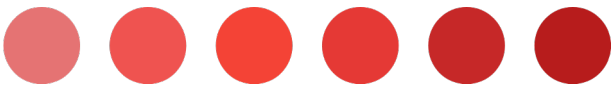
The sorting functions in Haskell have different types, different performance characteristics, and not all of them are widely known. So let's discuss all of them to understand once and for all how to sort lists properly in Haskell!

sort

The basic function for sorting in Haskell is **sort**. It is not exposed by default from Prelude, so you need to import it from the Data.List module. The function has the following type:

```
sort :: Ord a => [a] -> [a]
```

The type signature reads "if you give me a list of elements that can be compared with each other, I'll return you a list of elements of the same type". And the name suggests that the resulting list will be sorted. After playing with **sort** in GHCi, we observe that the list is sorted in the ascending order.



```
ghci> sort [3, 1, 2, 1]
[1, 1, 2, 3]

ghci> sort [0.0, -3.2, 5.55]
[-3.2, 0.0, 5.55]

ghci> sort ["words", "in", "list"]
["in", "list", "words"]
```

Interesting historical facts about **sort** function in base: Initially, this function was implemented using the *Quick Sort* algorithm up until 2002. The old implementation was then replaced with the *Merge Sort* algorithm, which was superior and guaranteed O(n log n) in worst-case scenarios as it was shown in benchmarks. However, this is also not the final version of this function in base. In 2009 the classical *Merge Sort* algorithm was given up, and the newer one was introduced, which can be called *Smooth Applicative Merge Sort*.

However, the **sort** function doesn't fit all use cases:

You can have only one instance of Ord per type, but sometimes you want different orders (e.g. order rows by name, date, rating, etc.), which is impossible to achieve with one data type.

Some types don't have Ord instances due to different reasons, but you still may want to order them.

Let's see if other functions can help us with these issues.

sortBy

To resolve the non-comparable elements issue of **sort**, another function from Data.List called **sortBy** comes to help:

```
sortBy
    :: (a -> a -> Ordering)
    -> [a] -> [a]
```

You can supply **sortBy** with a custom *"comparator"*, and it will sort the list according to the given order. For example, to sort the list of pairs by the second element, use it like this:

```
ghci> sortBy
        (\(_, x1) (_, x2) ->
          compare x1 x2
        )
        [(1, 3), (10, 1)]

[(10,1), (1,3)]
```

This pattern is very common, so the Data.Ord module has a helpful function called comparing

```
comparing
    :: Ord a
    => (b -> a)
    -> b -> b -> Ordering
```

Using this function, you can write the above sorting shorter:

```
sortBy
    (comparing snd)
    [(1, 3), (10, 1)]
```

sortOn

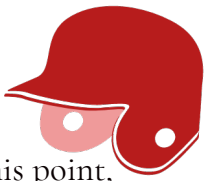
The pattern of sorting some values by some field or by the result of some function is also very common. So, Data.List implements one more function for exactly those reasons! Meet **sortOn**:

```
sortOn
    :: Ord b
    => (a -> b) -> [a] -> [a]
```

Using **sortOn**, we can sort tuples by the second element (as in the previous example) very smoothly. And it reads as a natural language!

```
sortOn snd [(1, 3), (10, 1)]
```

sortWith



This is already quite nice, and at this point, most people end their journey to the world of list-sorting functions. But there is still one more function! It is called **sortWith** and it comes from a surprising place — the GHC.Exts module:

```
sortWith
    :: Ord b
    => (a -> b) -> [a] -> [a]
```

You can notice that it has the same type as **sortOn**, so a fair question would be "What is the difference?" And the difference is in performance. Let's look at the implementations of both **sortWith** and **sortOn**:

```
sortWith
  :: Ord b => (a -> b) -> [a] -> [a]
sortWith f = sortBy
  (\x y -> compare (f x) (f y))

sortOn
  :: Ord b => (a -> b) -> [a] -> [a]
sortOn f =
  map snd
  . sortBy (comparing fst)
  . map (\x ->
      let y = f x in y `seq` (y, x))
```

Aha! The implementation of **sortWith** is probably what you expected from **sortOn**, and the implementation of **sortOn**, in reality, turned out to be more complicated. But what **sortOn** actually does is caching the result of the function application in the first element of a pair, sorting the pair by the first element, and then returning elements themselves.

So, the **sortOn** function actually requires more memory and is a bit slower than **sortWith** because of that. However, there is a reason for such implementation. If the "comparator" – function, by which you want to compare – is slow (e.g. list's length), **sortOn** will call this function only once for each element, while **sortWith** will call it many more times, and the final sorting will be slower.

Haskell Sorting Functions Table

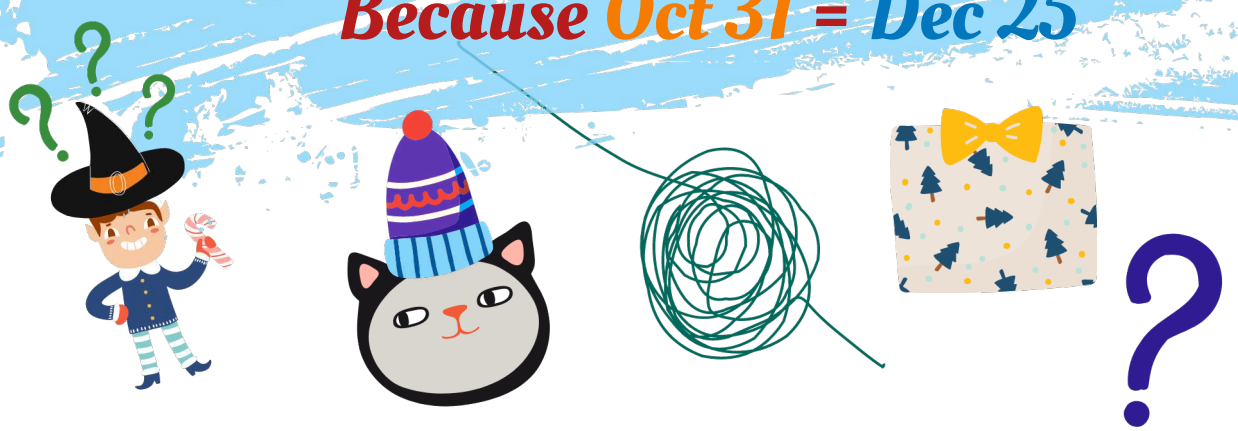
That's a lot of sorting functions! In order not to get lost, here are our compact info table and recommendation on when to use each:

sort	`Ord a`	`[a] -> [a]`	`Data.List`	The type has the "Ord" instance, and you want to sort in the ascending order
sortWith	`Ord b`	`(a -> b) -> [a] -> [a]`	`GHC.Exts`	You want to sort by record field, or by part of the type, which is "free' (only extracting fields, not computing anything)
sortOn	`Ord b`	`(a -> b) -> [a] -> [a]`	`Data.List`	You want to sort using some expensive function (e.g. list's length)
sortBy		`(a -> a -> Ordering) -> [a] -> [a]`	`Data.List`	For completely custom sorting behaviour

HUMOURMORPHISM

Why can't programmers tell the difference between Halloween and Christmas?

Because Oct 31 = Dec 25



CHALLENGE yo self

Let's first sum up the last month' competition. The challenge of our last month was to implement the list reverse function using sorting. There were lots of smart and interesting solutions. But we need to choose one!

The winner of this challenge with the most creative (and evil) solution that uses Incoherent Instances is **@utdemir**:

```haskell
{-# LANGUAGE FlexibleInstances #-}

import Data.List

instance Eq a where (==) _ _ = False
instance {-# INCOHERENT #-} Ord a where
  compare _ _ = GT

reverse' :: [a] -> [a]
reverse' = sort

main = print $ reverse' [1..10]
  -- [10,9,8,7,6,5,4,3,2,1]
```

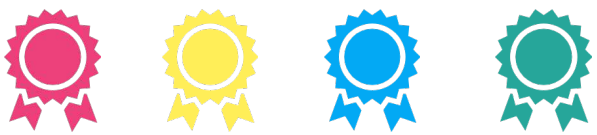
Congratulations, @utdemir! Send us an email with your contact details to xrom.xkov@gmail.com or a message to @bind_the_gap to claim your prize!

We are thankful to everyone participated! Here are a few more highlights from you, our creative readers.

In the same spirit, **@ajnsit** wrote a clever solution that exploits laziness in Haskell:

```haskell
reverse = sortOn (\_ _ -> undefined)

instance Eq a where
  _ == _ = False
instance Ord a where
  compare _ _ = GT
```



A less evil solution that uses safe Haskell abstractions is written by **@noaheasterly**:

```haskell
sortAsReverse :: forall a. [a] -> [a]
sortAsReverse = coerce (sort @(Rev a))

newtype Rev a = Rev a

instance Ord (Rev a) where
  compare _ _ = GT

instance Eq (Rev a) where
  _ == _ = False
```

The shortest solutions were provided by **@marcellourbani**, **@chrislpenner** and **@gilmi**:

```haskell
myreverse = map snd . sort . zip [0,-1..]

reverse = fmap snd . sortOn fst . zip [0,-1..]

reverse = map snd . sort . zip
    [(maxBound :: Int), (maxBound - 1) .. 0]
```

Thanks, everyone participating in our fun challenge, and we prepared a new one for you!

Implement the function that finds the **first digit of a number**, using as fewer characters as possible (excluding imports and language extensions, get creative!). A possible output:

```
ghci> firstDigit (-5264)
5
```

Send us your solutions to ***xrom.xkov@gmail.com***,
or tag ***@bind_the_gap*** on your solution in Twitter
and we will highlight the most elegant and creative solutions in the following issue!

TYPE emojination

Guess the standard function by the following type, written in emojis:

(💰➡️🎄🧸)➡️🎁💰➡️🎄(🎁🧸)

SURVEYVOR

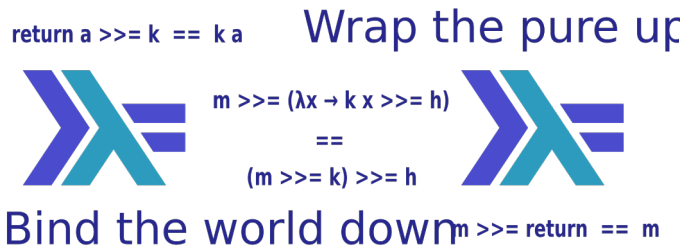
~ Monthly BTG survey, important community surveys and results ~

Tell us what you want to see happening in the Haskell world in 2021!

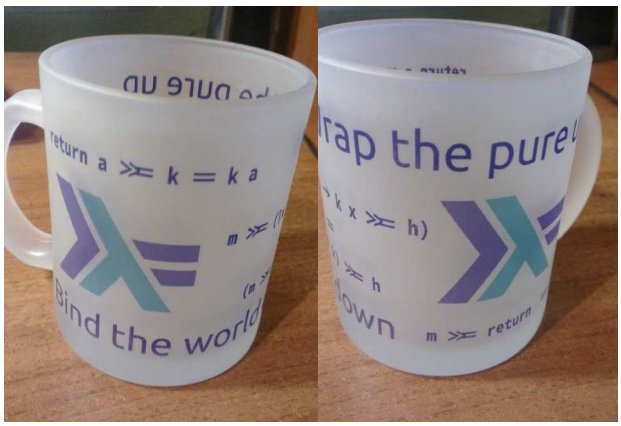
https://bit.ly/btg-survey-dec2020

PRESENT FROM READERS

We are happy to receive amazing content from our readers! After our pilot issue, Gleb Popov shared with us Haskell art for the cup.



Look how gorgeous it is in the real world!



Closing Words

Thanks a lot for reading our magazine. The second issue of Bind The Gap is brought to you by **Kowainik** — **Veronika Romashkina** and **Dmitrii Kovanikov**. The year was not easy for most of us, so we tried to bring a bit of the holiday mood into our programming lives through this edition. We hope you enjoyed it and it made you smile!

Besides BTG, we do a lot of open-source development, tutorials and guides writing, mentorship. You can visit our website to read more about our work:

https://kowainik.github.io/

We have plenty of ideas and plans for future issues. Work on the magazine takes a lot of time and effort. So your support is highly appreciated! You can support our work and BTG in particular on Ko-Fi or via GitHub Sponsorship:

https://ko-fi.com/kowainik
https://github.com/sponsors/vrom911
https://github.com/sponsors/chshersh

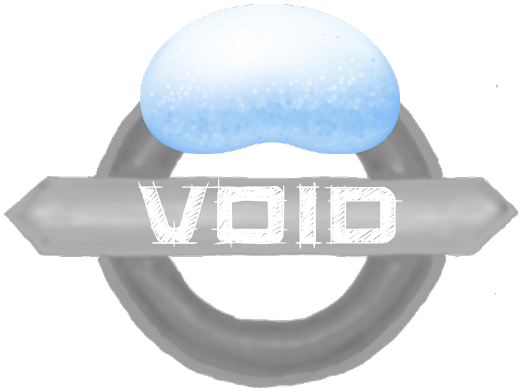
Oh, and the special present for those who asked us about the Bind The Gap merchandise, you can now get the first version of T-Shirt!

https://teespring.com/stores/kowainik

Merry Christmas & Happy New Year!
Hope to see you in 2021, folks ;)

Way out →

This is



where this issue terminates

ALL CHANGE PLEASE

Please remember to take all your

Monads

with you when you leave the train