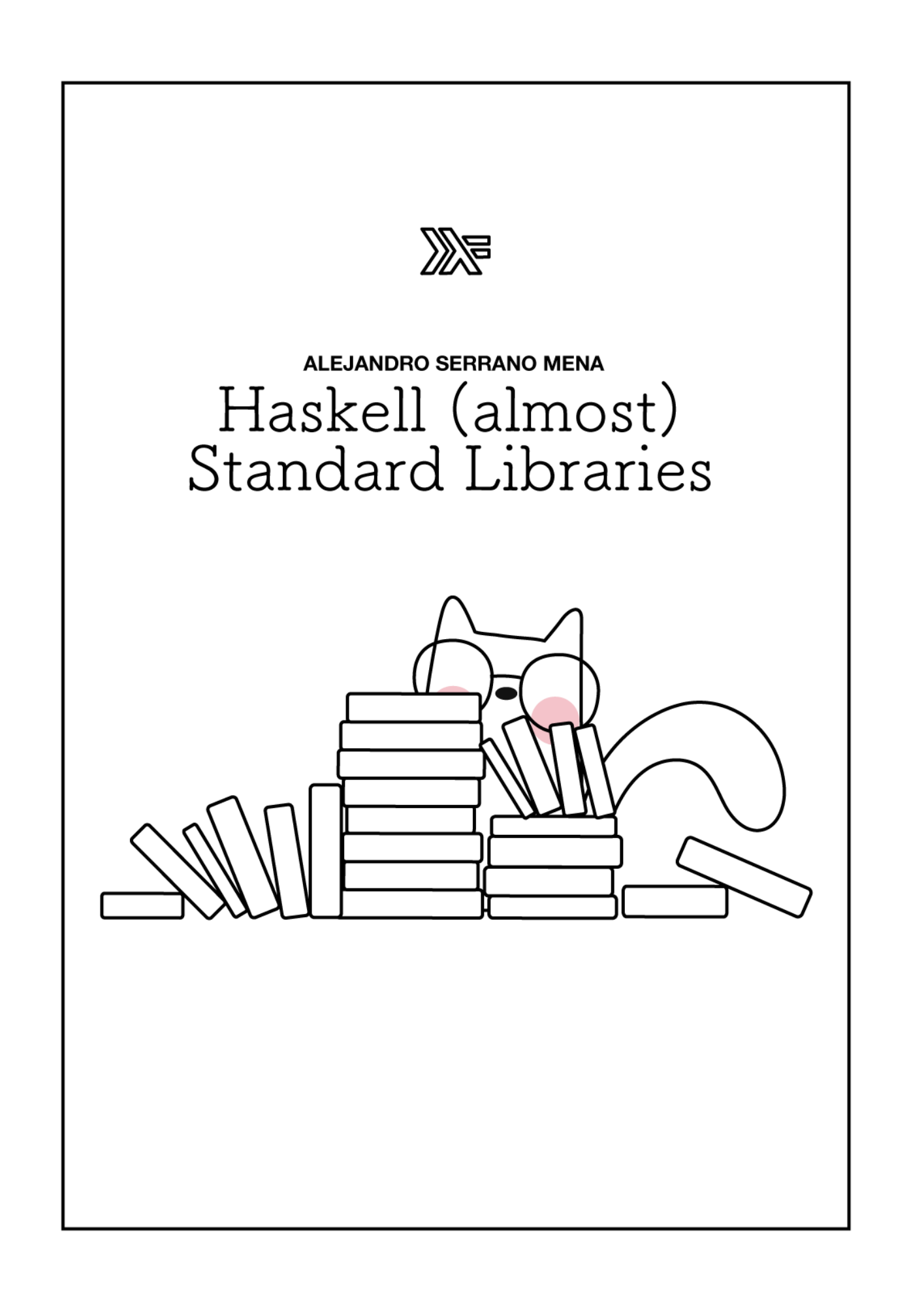
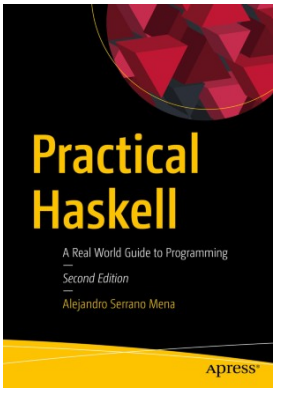


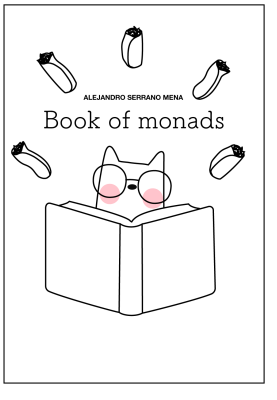
ALEJANDRO SERRANO MENA

Haskell (almost) Standard Libraries

Reviewer: Dave Parfitt

Other books by the same author





Contents

Summary of libraries

Preludes

`base` → standard library bundled with GHC
`relude` → removes unsafe functions like `head`
`classy-prelude` → additional type classes
`protolude` → basis for a Prelude with light dependencies

Utilities

`extra` → useful utility functions not found in `base`
`split` → functions to split lists in different ways
`safe` → replacements for unsafe functions like `head`
`foldl` → combine folds for better performance

Containers

`containers` → sets, trees, maps
`unordered-containers` → sets and maps based on hashing
`mono-traversable` → type classes for monomorphic containers
`fgl` → graphs and most traditional algorithms
`algebraic-graphs` → algebraic graphs, highly-customizable algorithms

Text

`text` → everything to deal with strings in a performant way
`text-icu` → algorithms for Unicode strings
`regex-base` → type classes for regular expressions
`regex-posix`, `regex-pcre`, `regex-tdfa`, `regex-parsec`
→ implementations of regular expressions as instances of `regex-base`
`regex-applicative` → regular expressions based on `Applicative`

Bytes
`bytestring` → packed sequences of bytes
`array` → arrays with custom indices, largely superseeded by `vector`
`vector` → efficient arrays indexed by `Int`s

Mutability
`primitive` → mutable references, generalizes `IO` and `ST`
`vector` → mutable arrays indexed by `Int`s
`vector-algorithms` → algorithms over mutable vectors

Serialization
`aeson` → serialize from and to JSON
`binary` → serialize and deserialize binary formats

Validation
`validation` → validation with error accumulation

Parsing
`attoparsec` → fast backtracking parsers for `ByteString` and `Text`
`megaparsec` → general parser library with an eye in error messages
`parser-combinators` → common functions for parsers, based on type classes
`replace-attoparsec`, `replace-megaparsec` → match and replace with parsers

Optics
`optics`, `microlens` → manipulation of immutable data
`aeson-optics`, `microlens-aeson` → optics for JSON documents

Controlling evaluation
`deepseq` → fully evaluate values in thunks

Exceptions
`safe-exceptions` → safer versions of `base`'s `Control.Exception`
`exceptions` → type classes which generalize the `throw` and `catch` operations
`retry` → policies for working with possibly-failing computations

Resources
`resourcet` → safe and deterministic resource management
`resource-pool` → management of pools of resources

Files
`filepath` → manipulate file names and paths
`directory` → operations over files and directories (move, copy…)
`temporary` → create temporary files and folders

Processes
`optparse-applicative` → parsing command-line arguments
`typed-process` → start and communicate with external processes

Streaming
`conduit`, `pipes` → high-level streaming pipelines

Randomness
`random` → generate random values of Haskell types
`mwc-random` → random generator suitable for statistical applications
`uuid` → generate and work with UUIDs

Time
`time` → represent and work with times

Async
`async` → spawn and manage threads for `IO` actions
`stm` → communication between threads based on optimistic locking
`stm-chans`, `stm-containers` → queues, maps, and sets for STM
`monad-par` → parallelism based on dataflow programming
`parallel` → parallelism based on strategies

Network
`req`, `wreq` → HTTP client libraries
`network-simple` → high-level TCP socket programming
`network` → low-level networking primitives

Web applications
`wai` → common Web Application Interface
`warp`, `warp-tls` → server for WAI-based applications
`wai-extra`, `wai-cors` → useful middleware for WAI applications

Functors
`base` → functor, contravariant, bifunctor
`profunctors` → profunctors

Transformers
`transformers` → most common monad transformers
`mtl` → monadic classes to facilitate working with transformers
`unliftio` → lifted version of `base` which works on `MonadIO`
`rio` → implementation of the `ReaderT` design pattern

Testing

`hspec`, `tasty` → test runners
`HUnit` → unit testing, integrations in `hspec` and `tasty-hunit`
`QuickCheck` → property-based testing, integrations in `hspec` and `tasty-quickcheck`

Logging

`fast-logger` → basic logging
`monad-logger`, `rio` → integration of logging into architecture

Monitoring

`ekg`, `ekg-wai` → remote monitoring of a running process

Reflection

`base` → reflection with `Data` and `Generics`
`syb` → generic traversals using the `Data` mechanism

1 Introduction

Welcome, traveler of the Haskell world! I'm sure you've learnt about lists, monads, and patterns, but maybe you're now faced with the question: what is my next destination? Unfortunately this book can't give you the answer, but instead tries to tell you about all the "boring" things you'll find along the way. At the end of the day, a Haskell program is like any other, and needs to manipulate text, talk to databases, or ping a server on the other side of the world.

Compared to the Java or .NET ecosystems, the `base` library – the "standard library" in this case – which comes with the GHC* distribution, is all but batteries included. This means that novices face the question of which libraries in the community repositories they need to use earlier and more often, and this is typically a source of frustration. Even worse, after checking a few of them, they might figure out that the "community-blessed" library is yet another one that hasn't been explored yet! For that reason, this book takes a very broad definition of what standard librarie**s** mean, pointing to the original meaning of the term: those libraries which are considered good by a fair share of Haskellers.

Think of this book as a high-level guide. The goal is for you to find your way among the catalogue of libraries, not to give a detailed description of each type, function, and class in each library. Some topics – like regular expressions – are treated because many other language ecosystems consider them part of their standard library, while other – like reflection with `Typeable` – discuss elements which are indeed part of Haskell's standard library but not so well-known.

*Glasgow Haskell Compiler, the main compiler used in the Haskell community.

1.1 The Haskell ethos

One fair question to ask is *why* is the `base` library so slim? This comes from the ethos of the Haskell community: *always keep improving*.[†] An important implication is that the barrier to bless a particular concept or design as part of the standard library is a very high one.

Take for example regular expressions. There are many ways in which one could solve this problem:

- Write a pure and straightforward implementation in Haskell,
- Write bindings to the native Perl PCRE library,
- Implement them on top of a generic parsing library.

Those solutions have different trade-offs with respect to speed, memory consumption, or portability. The Haskell community prefers to put the decision in the hands of the developer, and in fact `regex-tdfa`, `regex-pcre`, and `regex-parsec` are implementations of each idea.

But Haskell shines in abstraction, and there's a lot which can be abstracted among those libraries. Hence the existence of `regex-base`, which provides a common API which can later be instantiated to particular requirements. In fact, one could say those types or classes which make their way to `base` are those which express a very general concept: think of `Functor` or `Monad`.

The downside of this approach is, as we have discussed above, that beginners need to figure out which are the good libraries at a very early stage of their learning path. Depending on libraries even for trivial programs also adds a feeling of instability, but fear not, since most of the libraries discussed in this book have had their main API unchanged for more than a decade.

Keep in mind that even though these libraries form a good starting point, the Haskell community is a vibrant one, and new libraries may replace the old standard ones. If you have already worked with Haskell, your choice in some area may not coincide with the one in this book. My goal in any case is not to judge which library is "better", but to offer a good first selection; but others may be more advantageous with different coding styles or have different trade-offs.

1.2 Alternative `Preludes`

`Prelude` is the name of the module which is implicitly imported in every Haskell file.[‡] The `base` package defines one such module, but we can decide to remove

[†] Haskell's semi-official motto is *avoid "success at all costs."*

[‡] Unless you enable the `NoImplicitPrelude` extension in GHC.

that dependency and obtain the `Prelude` module from somewhere else. There is in fact no shortage of those *alternative* `Prelude`s in the Haskell community, ranging from slimmer incarnations to `Prelude`s which import almost every type we are going to discuss in the rest of the book. Popular alternative `Prelude`s are `relude`, which focus on adding an additional layer of type-safety by removing functions like `head`, `classy-prelude`, which adds additional type classes not found in `base`, and `protolude`, which tries to stay light while providing the most useful functions.

The general consensus in the community is that these alternative `Prelude`s are very useful in applications, especially as a way to keep a whole team in sync with respect to dependencies. However, they are advised against for libraries, as depending on that package would pull the alternative `Prelude` and maybe many more dependencies alongside it, increasing compile times and the possibility of a version conflict.

2 Dependencies

Many introductory Haskell materials glance over build and dependency management, a must for any project with more than a hundred lines. This chapter introduces the main names and concepts. In this regard Haskell does not reinvent the wheel; build tools are quite similar to what is found in other ecosystems.

All the information to build a Haskell project is stated in a *Cabal*[*] *file*, which resides in the root of the project and whose name must be that of the project followed by `.cabal`. In most cases the folder where the project is located also shares the name with the Cabal file. The following is a typical folder organization.

```
cool-project/
├── src/ ..... code files
│   └── CoolProject.hs ..... module CoolProject
├── test/ ..... test files
├── cool-project.cabal ..... project file
└── README.md ..... other files
```

Cabal files are written in a YAML-esque format, with a few syntactical quirks.[†] The file starts with a few key-value pairs describing the project as a whole – name, version, author – and then a sequence of stanzas are defined. Each *stanza* roughly corresponds to an artifact: a library, an executable, or a test suite. Each stanza also defines where to find the input files, dependencies upon other libraries, and options for the build.

[*] Common Architecture for Building Applications and Libraries.

[†] Some Haskellers think that using pure YAML is better, and have created `hpack`, an alternative format to describe Haskell projects.

By running the `cabal init --interactive` command inside a folder you can create a project skeleton quite easily. Here is a (trimmed down) result.

```
cabal-version:      2.4
name:               haskell-stdlibs
version:            0.1.0.0
author:             Alejandro Serrano

library
    exposed-modules:  MyLib
    -- other-modules:
    -- other-extensions:
    build-depends:    base ^>= 4.14
    hs-source-dirs:   src
    default-language: Haskell2010

executable haskell-stdlibs
    main-is:          Main.hs
    build-depends:
        base ^>= 4.14,
        haskell-stdlibs
    hs-source-dirs:   app
    default-language: Haskell2010
```

We are particularly interested in the `build-depends` property in each stanza. As you can see in the executable, this property takes a *list* of *package* names, separated by commas. In addition, you can specify a version number or range; in our case we specify that our project should build with any version of `base` from `4.14` onwards, but lower than the next major release `5.0`. In older Cabal files you may see `base >= 4.14 && < 5` instead.

In many cases throughout this book we'll talk about a package different from `base`. That means that if you want to use it, you need to add it to the corresponding `build-depends` property. Note that dependencies are not shared among stanzas, this is the reason why we need to specify `base` both for the library **and** the executable.

Once you have a Cabal file in place, you have two options to build the project: `cabal build` and `stack build`. True to its ethos of putting decisions in the hands

of developers, the Haskell ecosystem has two separate build tools, Cabal[‡] and Stack. The main different among the two are:

1. Stack takes care of downloading the compiler and setting up the toolchain if not present, whereas Cabal expects the toolchain to be readily available. In the latter case, you can use `ghcup` to set your environment up.
2. Cabal defaults to using Hackage as a repository and the latest version of each package which satisfies the constraints in the `build-depends` property, whereas Stack requires you to specify a snapshot which dictates the version of every package.

If you choose Cabal, remember to run `cabal update` from time to time to refresh the package information. If you use Stack, before building anything you should run `stack init` at the root of your project. This command selects the best stable snapshot and records that choice in a `stack.yaml` file.

2.1 Repositories

Hackage, at `hackage.haskell.org`, is the community package repository. Everybody can upload their packages and make them available to humankind. Package names are dealt with on a first-come first-served basis: once a user uploads a package with a certain name, only they, their collaborators, and the Hackage trustees can upload a new version of the same package. Apart from hosting the code, Hackage generates documentation for each library based on the Haddock comments – Haskell's rendition of docstrings, Javadoc, and the like.

Hackage has a search feature which by default searches by name and description. My personal recommendation is to order the results by number of downloads in the last month (the column named *DLs*), because it serves as a good proxy of the popularity of the library. For example, these are the results for "json".

Name	DLs	Rating	Description	Tags	Last U/L	Maintainer
json	287	1.75	Support for serialising Haskell to and from JSON	(bsd3, library, web)	2020-01-14	DonaldStewart, IavorDiatchki, SigbjornFinne
aeson	3131	2.75	Fast JSON parsing and encoding	(bsd3, json, library, text, web)	2022-01-01	AdamBergmark, BasVanDijk, BryanOSullivan, HerbertValerioRiedel, phadej, lyxia

[‡]Yes, we all find it confusing that project files are referred to as Cabal files even if you don't build them using Cabal-the-tool.

You can see that even though `json` has literally what we searched for in its name, `aeson` is way more popular. In fact, when talking about JSON serialization, we'll take the latter as the "standard library" for that purpose.

When you click the name of the library you go directly to its documentation. Nice packages have an introduction or tutorial right there, below the list of modules. At the right-hand side you can find the released versions of that package; here's what it looks like for `aeson` at the end of January 2022.

1.5.4.1, 1.5.5.0, 1.5.5.1, 1.5.6.0, 2.0.0.0, 2.0.1.0,
2.0.2.0, **2.0.3.0** (info)

You can use `aeson` as a dependency by simply adding it to your `build-depends`. Most of the times you want to support from the latest minor release to the next major release, which is achieved with a `^>=` range.

```
library
    ...
    build-depends: base ^>= 4.14, aeson ^>= 2.0
```

A common pain when maintaining large Haskell codebases is to keep the dependencies up-to-date. Unfortunately many libraries are still in their `0.x` series, which do not guarantee backwards- nor forwards-compatibility. Cabal comes with a powerful constraint solver which can figure out a good solution for a build plan, but unfortunately it cannot foresee problems caused by function or types changing from one version to the other. Stackage follows a different route: instead of trying each time to figure out a build plan, it fixes the versions of an entire set of libraries. Such a *snapshot* is only updated in bulk, and an automated process ensures that compiling any subset of dependencies from it always succeeds.

Stackage, at `stackage.org`, lists the available snapshots in its home page. Most of the time you either want to use the latest Long-Term Support (LTS) release, or check that the project still compiles with the latest GHC version by using the Nightly release.

Latest releases per GHC version

- Stackage Nightly 2022-01-27 (ghc-9.0.2), 4 days ago
- LTS 18.23 for ghc-8.10.7, published a week ago
- LTS 18.8 for ghc-8.10.6, published 5 months ago

Clicking on the snapshot name shows the packages and versions included in it. At the moment of writing the latest LTS, 18.23, lists `aeson-1.5.6.0`, which means that it lags behind the current 2.x series. This is common, since such a major release

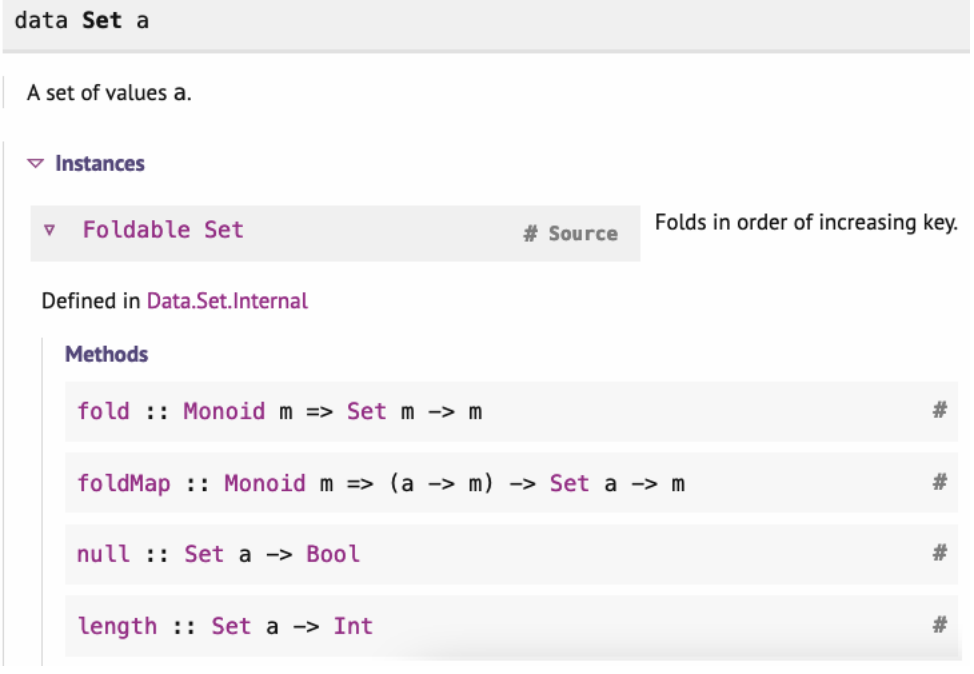
often involves lack of backwards-compatibility, and requires other packages in the snapshot to be updated.

Other than working by snapshots, Stackage works like Hackage. Clicking on a package shows its documentation, usually starting with an introduction or tutorial.

2.2 Search by type

In many cases, looking around the documentation is enough to find the function you need. However, there's a a core Haskell feature which makes it harder to find those functions directly: type classes.

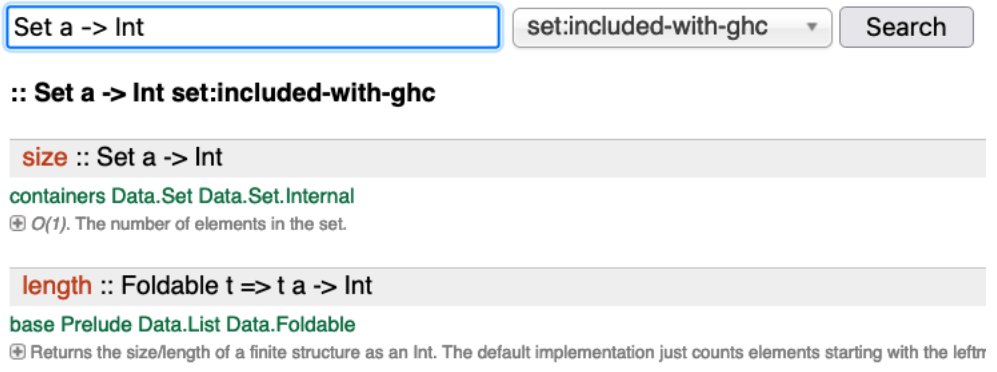
Let me give an example: you can use the `length` function in a `Set a`, one of the types defined in the `containers` library. However, in the documentation `length` is not mentioned at all;[§] where does it come from? In this case, from the `Foldable` instance. You can find it under the header *Instances* in the documentation for the type. Additionally, each entry in the list can be expanded to show the functions defined in that type class. There you'll find `length`, among dozens of other functions defined by `Foldable`.



Figuring out this kind of information would be almost impossible, had Hoogle not been invented. Available at `hoogle.haskell.org`, Hoogle gives you the super-

[§]Truth being told, maybe you should use the more specific `size` in this case.

power of searching by *type* of the desired function. In our example above, we know that a function returning the size or length of a `Set` must in any case return an `Int`. We type `Set a -> Int` in the search bar, and look through the results.



Hoogle understands Haskell's type system, so it can return a more polymorphic function that what you asked for, since that would also work. In fact, `length` has the type `Foldable f => f a -> Int`, which would be specialized to `Set a` to get the function you are looking for. You may also try with an even more specialized type like `Set String -> Int`, and the same results would be returned.

Note that in the image above the "Included with GHC" package set has been chosen, because looking around the whole of Hackage gives just too many results. You can also select a particular package, or a particular Stackage snapshot.

2.3 Advice on choosing a library

Given the huge amount of libraries in Hackage, it may be hard to choose one for a particular task, especially once we go out of the realm of super-popular libraries like the ones we treat in this book. The choice of a library is an investment, since you need to study its documentation, and it becomes part of the maintenance budget of the project. Here are some tips.

Popularity. Those libraries on which many others depend are usually a good option; they've already had some scrutiny about their set of features, how performant they are, and its trade-offs. Maybe even somebody has written a blog post about it. Popularity is hard to measure, but here are some indicators:

- The number of monthly downloads in Hackage, which you can find in the *DLs* column. My suggestion is to order by this column for any search you make in Hackage.

- The number of reverse dependencies of the package, which you can check at `packdeps.haskellers.com/reverse`. Although this page only indexes packages in Hackage, knowing that many other people depend on a package is a strong signal.

Maintenance. The feeling of continuous experimentation in the Haskell community sometimes leads to packages being left unmaintained. Depending on such a package involves a risk. Hackage gives the date of the last upload of the package, which you can use as an initial measure.

However, some packages are "finished", in the sense that authors are not adding more features. Still, sometimes updates are required when a GHC version with a breaking change becomes available, like the latest move from GHC 8.10 to 9.0. A fairer indication can be obtained from these two sources:

- Whether it's part of the latest Stackage LTS release. Stackage has an automated process to ensure that libraries are updated in response to breaking changes in dependencies and the compiler; being there ensures that somebody is taking care of the duty of maintenance.
- Look at the issue tracker for the project – found on its Hackage page – and check that there are no important issues regarding maintenance or compatibility lingering for a long time.

Finally, note that both Cabal and Stack allow depending on a source repository, like Git, instead of a published package. This is useful for quickly checking the development version of a library if the published one has some conflict with the rest of your project.

3 Utilities

The default `Prelude` in `base` exports many common data types, like lists, tuples, or string, alongside a bunch of useful functions over those. However, it does not export *everything* that `base` has to offer, so it's useful to know where to look for more.

Although Hoogle remains the first best option, it's less useful with basic types like integers, because so many functions have type `Int -> Int`. Talking about integers, the `Data.Int` module contains types ranging from `Int8` to `Int64` which have explicit bounds, as opposed to `Int` which does not guarantee them.

The modules `Data.List` and `Data.Maybe` also contain a wide range of small utility functions. My favorite one is `mapMaybe`, which allows you to map and filter in one single step. Other functions which work over those types and many others live in `Control.Monad`; I don't know how anybody could live without importing `guard`, which allows a very declarative way to express validations (more in the *Validation and parsing* chapter).

```
validPerson first last age = do
  guard $ not (null first)
  guard $ not (null last)
  guard (age >= 0)
  -- do more things later
```

`Data.Monoid` contains many data type definitions which can be used to select a particular operation when a type supports several of them, via `newtype`s. This is the case of Booleans, for which both conjunction `(&&)` and disjunction `(||)` form monoids. A general pattern when using container types is to aggregate information using the `foldMap` method from `Foldable`, as we discuss in the *Containers* chapter.

The `elem` function, which checks whether a value lives in the data structure, can be defined as the result of aggregating the function "is this the element we are looking for?" using disjunction – one single `True` value is enough to answer "yes" to the question. The corresponding `newtype` in this case is `Any`.

```
elem :: (Eq a, Foldable t) => a -> t a -> Bool
e `elem` xs = getAny $ foldMap (\x -> Any (x == e)) xs
```

We do not produce a `Bool`, but an `Any`, hence the need for the additional constructor call. As a consequence, the result of `foldMap` is of type `Any`, from which we obtain the underlying `Bool` via `getAny`. Had we chosen the other Boolean operation via `All`, we would instead be answering the question "are all the elements in this container equal to `e`?"

Still you may find yourself writing a small utility function and thinking "that should be in the standard library". In most cases you'll find it in the `extra` library. For each module, like `Data.List`, the library defines a `Data.List.Extra` with more functions. One particular example: functions like `dropEnd`, `takeEnd`, `breakEnd`, ... which start their behavior at the end of the list. Something that in many cases you could have done by a clever combination of the basic functions and `reverse`, but that can also get hairy very quickly if you need to do it yourself.

One area in which `base` and `extra` still fall a bit short is in separating parts of a list in different ways. The `split` library comes to the rescue here. For example, imagine you need to have a sliding window of 3 elements, starting every 2 elements. The `divvy` function is there for you:

```
> import Data.List.Split
> divvy 3 2 [1 .. 7]
[[1,2,3],[3,4,5],[5,6,7]]
```

Another interesting utility library is `safe`, whose goal is to provide safe variants of unsafe functions in Prelude. The main example is `head`, which throws an exception when confronted with an empty list.

```
> head []
*** Exception: Prelude.head: empty list
```

The `safe` library defines different variations of `head`, by either optionally returning or having a default value:

```
headMay :: [a] -> Maybe a
headDef :: a -> [a] -> a
```

Its popularity is witnessed by the great amount of libraries in Hackage which depend on `safe`. As we discussed in the *Introduction*, some alternative `Prelude`s like `relude` make this kind of safety a core design decision, and fully replace the unsafe variants with safe ones.

3.1 Better folds

Many problems in Haskell can be solved using a *fold*, a higher-order function which aggregates the information from a container into a single value. Almost every introductory material for Haskell talks about `foldr` and `foldl`, the concrete functions for lists. You can define the `length` and the `sum` of a list using `foldl`.

```
> mySum = foldl (+) 0
> myLength = foldl (\l _ -> l + 1) 0
> (myLength [1,2,3], mySum [1,2,3])
(3,6)
```

When we are using several folds over the same structure, we can do better. Even in the small example above we are traversing the list twice, when one traversal is actually enough. If we don't mind manually constructing and deconstructing tuples, we can write a version which uses `foldl` yet only traverses the list once.

```
> myLengthSum = foldl (\(l, s) x -> (l + 1, s + x)) (0, 0)
> myLengthSum [1,2,3]
(3,6)
```

It's clear, though, that having to write these functions for every combination of folds is not sustainable. Fortunately, the `foldl` package comes to our rescue.

The key idea in `foldl` is that if we define the shape of the fold independently of its application, we can optimize it before being applied to a particular data structure. The language we use for combining folds is `Applicative`, which also plays an important role in the *Validation and Parsing* chapter. As a small reminder of that type class, it gives us the ability to combine computations via `(<$>)` and `(<*>)`, applying a function at the end. In the example below, we combine the results of `Fold.length` and `Fold.sum` using the tuple constructor.

```
> import qualified Control.Foldl as Fold
> lengthSum = (,) <$> Fold.length <*> Fold.sum
> Fold.fold lengthSum [1,2,3]
(3,6)
```

Applying `lengthSum` using `Fold.fold` is guaranteed to traverse the data structure only once. The magic is that all the micro-managing of tuples is baked in the library and hidden from us.