Science of Computer Programming 201 (2021) 102523

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Which monads Haskell developers use: An exploratory study

Ismael Figueroa [a,1], Paul Leger [b], Hiroaki Fukuda [c]

[a] Ingeniería en Información y Control de Gestión, Universidad de Valparaíso, Chile
[b] Escuela de Ingeniería, Universidad Católica del Norte Coquimbo, Chile
[c] Shibaura Institute of Technology, Tokyo, Japan

ARTICLE INFO

Article history:
Received 30 January 2020
Received in revised form 30 May 2020
Accepted 22 July 2020
Available online 6 August 2020

Keywords:
Monads
Empirical study
Use of monads
Haskell
Hackage
Mining software repositories

ABSTRACT

Monads are a mechanism for embedding and reasoning about notions of computation such as mutable state, I/O, exceptions, and many others. Even though monads are technically language-agnostic, they are mostly associated with the Haskell language. Indeed, one could argue that the use of monads is one of the defining characteristic of the Haskell language. In practical terms, monadic programming in Haskell relies on the standard `mtl` package library, which provides eight core notions of computation: identity, error, list, state, reader, writer, RWS, and continuations. Despite their widespread use, we are not aware of any empirical investigations regarding which monads are the most used by developers. In this paper we present an empirical study that covers a snapshot of available packages in the Hackage repository—covering 85135 packages and more than five million Haskell files. To the best of our knowledge this is the first large-scale analysis of Hackage with regards to monads and their usage as dependencies. Our results show that around 30.8% of the packages depend on the `mtl` package, whereas only 1.2% depend on alternative, yet compatible implementations. Nevertheless, usage patterns for each specific monad remain similar both for `mtl` and alternatives. Finally, the state monad is by far the most popular one, although all of them are used. We also report on the distribution of packages that use `mtl`, regarding their category and stability level.

1. Introduction

The Haskell language [1] is an example of *pure* functional programming languages. What makes a functional language *pure* is its emphasis in the absence of *side-effects*, such as mutable state, I/O or exceptions, which has several benefits regarding equational reasoning, and easy parallelization of programs, amongst others. In contrast to functional languages such as Scheme or the ML family, which provide standard side-effecting operations on top of a functional approach, Haskell uses *monads* [2,3] as a kind of "design pattern"—with strong theoretical foundations—for the specification and execution of *notions of computation* that can represent the aforementioned side-effects. Moreover, this style of programming is strongly supported by Haskell compiler developers by means of special-purpose syntax and libraries, which results in a language that supports general-purpose, practical monadic programming. Indeed, Haskell provides a standardized interface for monadic programming in the form of the *monad transformers library*—simply known as the `mtl`.

However, despite the existence of `mtl` and the prevalence of monads in Haskell, to the best of our knowledge we are not aware of any empirical investigation on how Haskell developers actually *use* monads. As this is a broad question, in this

E-mail addresses: ismael.figueroa@uv.cl (I. Figueroa), pleger@ucn.cl (P. Leger), hiroaki@shibaura-it.ac.jp (H. Fukuda).
[1] Funded by FONDECYT Postdoctoral Project 3150672.

https://doi.org/10.1016/j.scico.2020.102523
0167-6423/

paper we focus on presenting several quantitative results obtained from an empirical investigation of *Hackage* [4], a platform that defines itself as "[the Haskell's] community's central package archive of open source software".[2] Hackage packages are build using the *Cabal* system [6], which serves as a tool to download, build and install packages from Hackage, as well as from local sources. Crucially, a requirement for Hackage packages is to provide a well-formed `.cabal` file that specifies the build options, dependencies, and other required metadata.

In this paper we perform an exploratory and quantitative study that consists of the massive analysis of package dependencies—based on Cabal files' metadata—to address the following research questions:

RQ1. *How many packages directly depend on the* `mtl` *library? What is their distribution with respect to package metadata, such as names, versions, stability or categories?*

RQ2. *What monads, or notions of computation, defined in the* `mtl` *library are the most popular?*

RQ3. *How popular are alternative implementations to the* `mtl` *library?*

Although we could compute indirect dependencies on `mtl`, we work under the assumption that only packages that directly depend on this library make a relevant, non-trivial, use of monads—meaning that a core part of their implementation is based on monads, rather than merely using a library that is implemented using monads. This design decision helps us trim the otherwise huge number of packages that indirectly would depend on the `mtl`.

By understanding which monads are used, we hope to provide language researchers with empirical information for the design and development of novel monadic libraries, or the refinement of existing alternative approaches to using monads in Haskell. Indeed, as we show later in the paper, we found several actionable insights that could be applied even beyond Haskell itself. Even though in this paper we focus on `mtl`, because it is by far the most widespread standardized implementation of monads in Haskell, we still quantify packages that use alternative implementations. In this work we process a complete snapshot of the Hackage repository, from January 2018, comprised of 85135 packages/version combinations, with a total size of around 30 GB. By parsing and analyzing their package metadata *and the source code* of all involved modules, we get a dataset that enables us to answer these research questions. Our datasets and processing scripts are available online [7].

Paper roadmap. We first present necessary background on monadic programming and empirical research on Hackage (Section 2). Then we explain our methodology and processing pipeline, as well as the structure of the datasets used in this work (Section 3). After that, we present and discuss the empirical results for all the research questions (Section 4) with a special focus on understanding which computations are the most frequently used. Next, we discuss in more detail an unexpected outcome of this research, the important usage of the modules for monad transformers (Section 5). Before finishing we discuss relevant threats to the validity of this study (Section 6), and then we discuss related work to finally conclude (Sections 7 and 8).

Please notice that readers should only need a passing familiarity with functional programming and with the general concept of a monad as a design pattern for purely functional computations.

2. Background

Monadic programming. In practical terms, monads are used in Haskell as a mechanism for embedding and reasoning about computational effects, such as mutable state or I/O. The approach was first suggested by Moggi [2] and Wadler [3], and then it was extended by Liang et al. [8] through the introduction of *monad transformers*. Monadic programming in Haskell is standardized through the standard *monad transformers library*—known just as `mtl`—which defines a set of monads and monads transformers that can be flexibly composed together. Although alternatives such as `transformers`, `monads-tf`, or `monads-fd` (described later in Section 4.3) do exist, and custom implementations are also possible, we focus on the `mtl` mainly due to its widespread usage. The `mtl` defines monads and monad transformers for the following notions of computation:

- *Identity*: represents pure computations in a monadic setting, it has no computational effect.
- *Error*: represents computations that may fail, propagating error messages if necessary.
- *List*: represents computations that may yield multiple, non-deterministic results.
- *State*: represents computations that have access to mutable state.
- *Reader*: represents computations that offer a read-only operation, *e.g.* for passing around configuration values.
- *Writer*: represents computations that offer a write-only operation, *e.g.* for logging.
- *RWS*: combines the Reader, Writer and State monad into a combined notion of computation.
- *Continuations*: represents computations that can be suspended, passed around and resumed, based on application of continuations.

[2] This paper extends and subsumes the preliminary work presented at [5].

```
name:               xmonad
version:            0.14.1
synopsis:           A tiling window manager
...
category:           System
extra-source-files: README.md
                    CHANGES.md
                    CONFIG
                    STYLE
                    tests/*.hs
                    tests/Properties/*.hs
                    tests/Properties/Layout/*.hs
                    man/xmonad.1.markdown
                    man/xmonad.1
                    man/xmonad.1.html
                    util/GenerateManpage.hs
                    util/hpcReport.sh
cabal-version:      >= 1.8

library
  exposed-modules: XMonad
                   XMonad.Config
                   XMonad.Core
                   XMonad.Layout
                   XMonad.Main
                   XMonad.ManageHook
                   XMonad.Operations
                   XMonad.StackSet
  other-modules:   Paths_xmonad
  hs-source-dirs:  src
  build-depends:   base                  >= 4.9 && < 5
                 , X11                   >= 1.8 && < 1.10
                 , containers
                 , data-default
                 , directory
                 , extensible-exceptions
                 , filepath
                 , mtl
                 , process
                 , setlocale
                 , unix
                 , utf8-string           >= 0.3 && < 1.1
  ghc-options:     -funbox-strict-fields -Wall -fno-warn-unused-do-bind

executable xmonad
  main-is:       Main.hs
  build-depends: base, X11, mtl, unix, xmonad
  ghc-options:   -Wall -fno-warn-unused-do-bind
```

Fig. 1. A redacted version of `xmonad.cabal` file, to illustrate the structure and metadata used in the study.

- We also consider the use of monad transformers—denoted as *Trans*—as a building block provided by the `mtl` for the modular construction and integration of new monads.

Hackage repository. Hackage [4] is the *de-facto* repository for open source software written in Haskell. It features more than 85135 packages, when considering each version independently, or equivalently, 12462 "whole" packages, taking all versions as a single software product. In Hackage, developers can upload several versions of a package, alongside its metadata, following the conventions of the *Cabal* build system [6]. A package is described by a `.cabal` file, which declares several build options, such as dependencies, stability, categories, language extensions, etc. The `cabal-install` tool leverages this metadata in order to automatically install a package. Taken as a whole, Hackage and Cabal provide a rich environment for the development and distribution of Haskell software.

Cabal files. To illustrate the kind of package metadata processed in our study Fig. 1 shows a redacted version of the `.cabal` file from the XMonad package.[3] This figure shows the different sub-sections regarding, *e.g.*, package name, version, and

[3] http://hackage.haskell.org/package/xmonad.

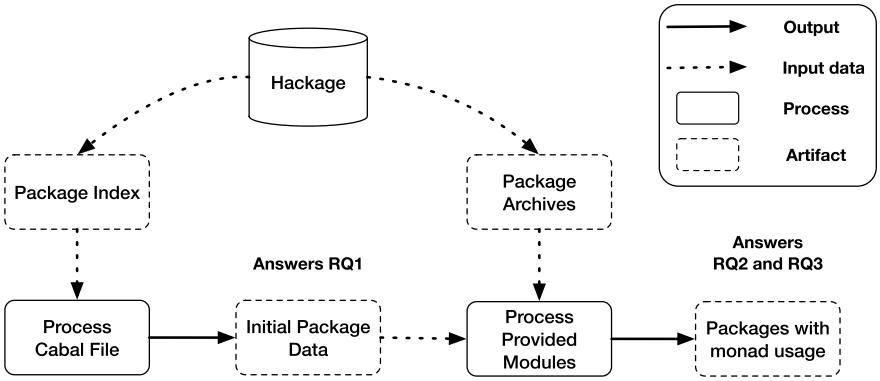


Fig. 2. Processing pipeline for gathering empirical data on monad usage. First, using the *package index*, downloaded from Hackage, we compute *initial package data*, which includes package dependencies, thus allowing us to answer RQ1. The next step is the per-package processing of source code, downloaded from the package archives, to find which modules are imported in order to answer RQ2 and RQ3.

category. Most crucial to our research is the `build-depends` sub-section that describes the package dependencies. Finally, it is interesting to remark that a given `.cabal` can specify the metadata for both *libraries* and *executables*, each with its own set of dependencies that must be considered in the analysis.

Empirical research on Hackage. There is some background on empirical studies using the Hackage repository. Morris [9] analyzed Hackage to assess the usage frequency of a GHC extension named *OverlappingInstances*, in order to guide the design of the Habit[4] language. Another study was performed by Bezirgiannis et al. [10] to evaluate the adoption of generic programming features in Haskell, one year after their introduction. The authors report that between 2012 and 2013, there was a 585% increase in the use of the `Generic` type class. Another contribution of their work is the *gpah* tool, that automates the analysis performed in their work.

3. Methodology

We follow a simplified version of the standard pattern used in the *mining software repositories* (MSR) research [11], which is depicted in Fig. 2. Our pipeline features two broad stages: generation of *initial package data* followed by a more costly step, the generation of *monad usage data*. We use a combination of standard Python tools for data analysis, such as `numpy` and `pandas`, and specific Haskell programs, mainly for parsing and querying Haskell and `.cabal` files, using Cabal's own API and the standalone `haskell-src-exts (HSE)` parser.[5]

In this work we consider all the packages in Hackage that were available in January 2018. Although our pipeline is quite simple and unoptimized, we managed to process around 30 GB of package data in a single standard workstation—albeit the process took several days to complete. We had to restart the process several times, because it turns out that properly parsing Haskell files is surprisingly difficult. As far as we know the only viable options are using HSE as we did, or using GHC as a library, making us rely on its internal parser. Later in Section 6 we discuss about the issues regarding file parsing. All programs and datasets used in this work are available in the companion website [7].

Generating initial package data. The first input artifact is the *package index*, available online in Hackage, which organizes all packages, their versions, and their `.cabal` files in a hierarchical folder structure. To parse a `.cabal` file we use Cabal's own API in a simple Haskell program. For each package we obtain the following metadata: *version*, *stability*, *dependencies*, *categories*, the *provided modules*, and the *main modules*. Both stability and categories are free-form strings, whereas the other entries are well-structured, and can be traced to other entities inside Hackage. Using this data we can inspect the dependencies of each package to quantify how the `mtl` is directly stated as a requirement.

Generating monad usage data. The goal of the next process is to find out what specific modules are imported in the code of a package. This way, we can quantify the usage for each specific monad in the `mtl`. We address this issue in two steps: computing imported modules, and then analyzing monad usage. Computing imported modules amounts to parsing and analyzing the main and provided modules of a package. To do this we download the package source and feed it to another Haskell program, which uses HSE. With this, it is simple to tag and count each usage of a monad module.

Package information. To clarify the data generated in the processing pipeline we briefly describe the fields that are computed for each package:

[4] http://hasp.cs.pdx.edu.
[5] https://hackage.haskell.org/package/haskell-src-exts.

- **Package name**: a string that is unique in the Hackage catalog. No two packages can share the same name.
- **Package version**: a string that follows a numeric convention, for major, minor and patch increments. For instance, "0.0.0.1" and "1.12.20.3" are valid version strings. In the Cabal API versions are comparable and sortable, corresponding to lexicographical order of the version strings.
- **Stability level**: a free-form string, added by the package developer, informing about the package stability.
- **Categories**: a list of free-form strings that describe all categories the package belongs to. Categories are also defined and added by package developers.
- **Dependencies**: a list of package names and version ranges, *e.g.* "`base` $>= 2$ `&&` < 5", or "`mtl` $== 2.1.*$".
- **Provided modules**: all modules that are publicly available for use in projects that depend on this package.
- **Main modules**: a package can specify several executables, each of them with a driving *Main* module. This field is a list of all main modules for the package executables.
- **Imported modules**: the set of all module names imported in the source files of the package. Each module name appears only once, even if imported in several source files.
- **`mtl-direct` flag**: signals whether or not the package depends on the `mtl` package, that is, `mtl` appears in its dependencies field.
- **`Cont` flag**: signals if the package modules import at least one of the following modules, related to the continuation monad in the `mtl`:
  - `Control.Monad.Cont`
  - `Control.Monad.Cont.Class`
- **`Error` flag**: signals if the package modules import at least one of the following modules, related to the error monad in the `mtl`:
  - `Control.Monad.Error`
  - `Control.Monad.Error.Class`
- **`Except` flag**: signals if the package modules import the module `Control.Monad.Except`, which is related to the except monad in the `mtl`—available only since version 2.2.1.
- **`Identity` flag**: signals if the package modules import the `Control.Monad.Identity` module, related to the identity monad in the `mtl`.
- **`List` flag**: signals if the package modules import the module `Control.Monad.List`, related to the list monad in the `mtl`.
- **`RWS` flag**: signals if the package modules import at least one of the following modules, related to the RWS monad in the `mtl`:
  - `Control.Monad.RWS`
  - `Control.Monad.RWS.Class`
  - `Control.Monad.RWS.Lazy`
  - `Control.Monad.RWS.Strict`
- **`Reader` flag**: signals if the package modules import at least one of the following modules, related to the reader monad in the `mtl`:
  - `Control.Monad.Reader`
  - `Control.Monad.Reader.Class`
- **`Writer` flag**: signals if the package modules import at least one of the following modules, related to the writer monad in the `mtl`:
  - `Control.Monad.Writer`
  - `Control.Monad.Writer.Class`
  - `Control.Monad.Writer.Lazy`
  - `Control.Monad.Writer.Strict`
- **`State` flag**: signals if the package modules import at least one of the following modules, related to the state monad in the `mtl`:
  - `Control.Monad.State`
  - `Control.Monad.State.Class`
  - `Control.Monad.State.Lazy`
  - `Control.Monad.State.Strict`
- **`Trans` flag**: signals if the package modules import at least one of the following modules, related to the standard monad transformer library in the `mtl`:
  - `Control.Monad.Trans`
  - `Control.Monad.Trans.Class`

4. Empirical results and discussion

Before answering the specific research questions posed at the beginning, we first show an overview of the Hackage repository. More specifically, we describe global statistics related to the quantity of packages, versions, amount of provided modules, and other global quantitative information. The aim of this is to characterize packages and serve as a baseline of

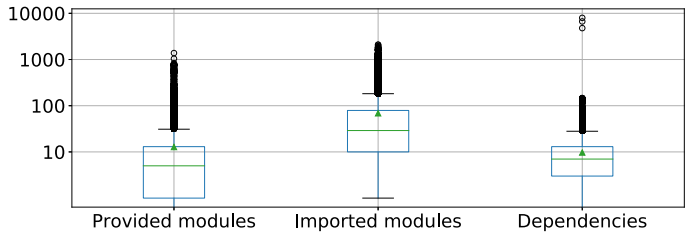


Fig. 3. Quantity of Imported/provided modules and package dependencies. Triangles mark the mean, lines inside boxes mark the median. Circles represent outliers.

what the "average" package looks like. The total size of uncompressed packages that were downloaded is around 30 GB. This is a non-trivial amount of data, although still manageable on a single computer. To clarify the terminology used throughout the rest of this section, we consider the following distinction between *packages* and *whole-packages*:

- A *package* is a specific version of a software product released in Hackage. For instance, versions 0.1 and 0.2 of XMonad are two different packages, although they refer to the same product.
- A *whole-package* refers to the software product as a whole, comprising all of its versions. Hence, all versions of XMonad count as only one unique package. The notion of whole-packages is only used in Section 4.1, regarding the answer of RQ1.

In total we processed 85135 packages or, equivalently, 16491 whole-packages from Hackage. In terms of modules, this amounts to 5842979 provided modules.

Distribution of dependencies, imported and provided modules. Fig. 3 shows the distribution of packages regarding the quantity of imported and provided modules, as well as the number of dependencies. The boxplots show that:

1. Most packages (75%) provide between 1 and 10 modules, while outliers range between 10 and 1000 provided modules. The average of provided modules is a bit more than 10.
2. Half of the packages import between 10 and 100 modules, when considering all unique imports in all files in a package. Other 25% of packages import between 1 and 10 modules, whereas outliers go above 1000 imported modules. The average is almost 100 imported modules per package.
3. Half of the packages declare between 1 and 10 dependencies on other packages from Hackage. Outliers can go up to hundreds of dependencies, whereas only three packages declare more than 4000 dependencies. The average is almost 10 dependencies per package.

Considering all the above we can argue that, in general, packages rely on a few dependencies that provide several modules. Given that on average each package provides 10 modules, and each package also depends on 10 other packages, it is likely that packages import almost all of the modules of their dependencies, thus reaching the average of 100 imported modules. This situation suggests that packages are loosely coupled—each focused on a single specific task—and relying on external dependencies for auxiliary operations.

Outliers. As mentioned above, there are three packages that declare more than 4000 dependencies. They correspond to three different versions of the `acme-everything` package,[6] which is sort of a "joke" package that "requires the entirety of Hackage to be built". Another set of outliers declaring more than 100 dependencies corresponds exclusively to several versions of the `yesod-platform`[7] package. Yesod is a framework for developing web and REST-based applications, and the `yesod-platform` package has many dependencies because its purpose is to provide specific versions of its dependencies in order to avoid configuration problems upon installation. Between 10 and 100 dependencies, there are many unrelated packages that simply happen to have many dependencies.

Distribution of versions per whole-package. Fig. 4 shows that 75% of the packages has around 2 and 9 versions. Indeed the average number of versions per package is 8. Outliers range from 24 versions, up to the maximum of 167 versions. The `purescript` package, a strongly- and statically-typed language that compiles to Javascript, has the most number of versions. Other packages with more than 100 versions are: `egison` and `hakyll`, two domain-specific languages implemented in Haskell; the `lens` library, composed of classes and combinators for traversal and update of functional datastructures; `shelly`, a package for POSIX access to execution of shell commands; `pandoc`, a general-purpose tool for document generation and conversion; `http-conduit`, an HTTP and HTTPS client used in the Yesod platform; and `llvm-general`, a

[6] https://hackage.haskell.org/package/acme-everything.
[7] https://hackage.haskell.org/package/yesod-platform.

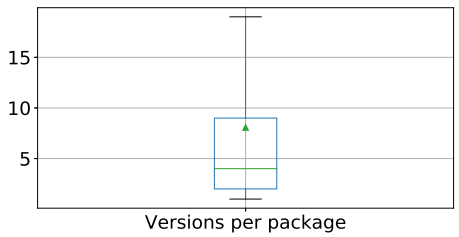


Fig. 4. Distribution of quantity of versions per whole-package. Outliers are omitted but range between 24 and 167.

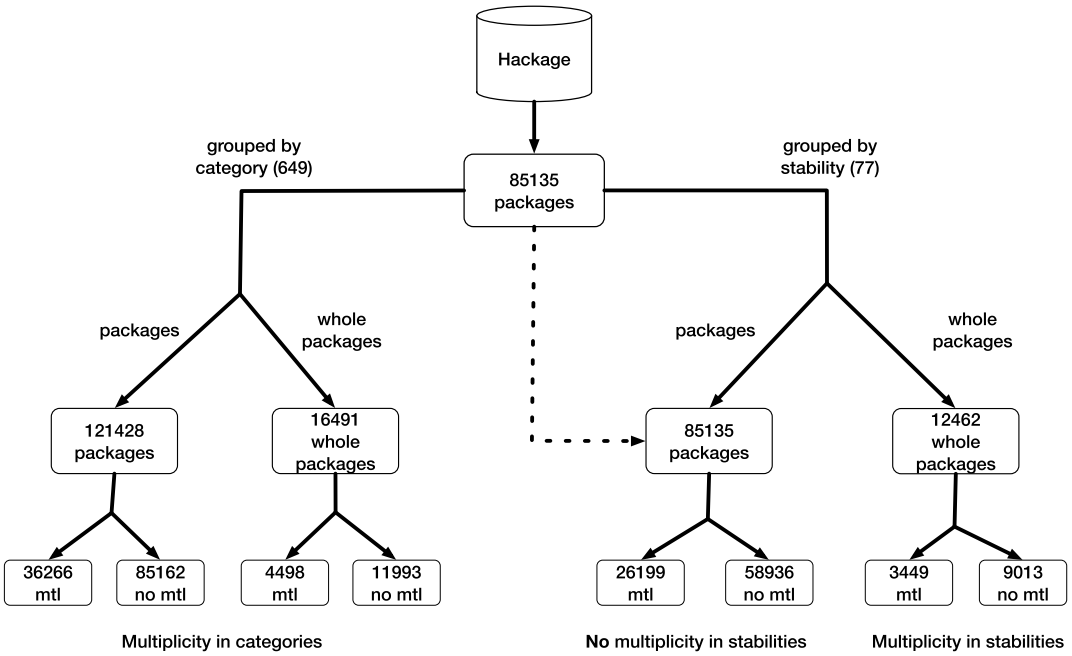


Fig. 5. Baselines considered for the analyses in this section.

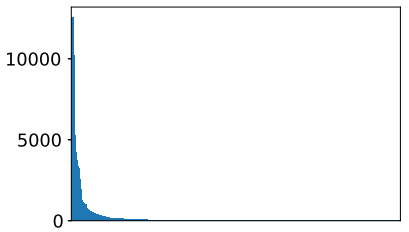
package that provides Haskell bindings to the LLVM platform. In general, after a manual inspection, we see that packages with a large number of versions are mostly those related to languages and compilers, as well projects originated around 2010 that are still under active development.

`mtl`-*packages.* We denote a package that directly depends on the `mtl` as an `mtl`-*package*. This definition is used throughout the rest of the paper, to contrast these packages with those that do not directly state `mtl` as a dependency.

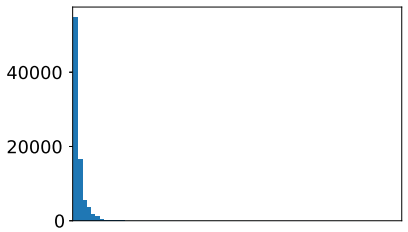
4.1. Regarding RQ1

RQ1 is the most basic questions in this exploratory study: how many packages depend on the `mtl`? However, answering this question is not simple. To provide an in-depth answer to RQ1 we need to consider the two interpretations given above for *packages* and *whole-packages* in Hackage. In addition, RQ1 also inquiries about the distribution of `mtl` packages regarding package metadata, namely, *category* and *stability* fields. Due to the *multiplicity* of categories and stability, *i.e.*, the situation in which a package can belong to one or more categories or stability levels at the same time, the quantitative analysis must consider several *baseline values*, which ultimately determine the calculation of percentual ratios between packages, categories, and stability levels. Note that multiplicity of stability levels happens only when merging the packages as whole-packages, because although each package has a unique stability version, different versions can contain different stability levels, which should be expected from the evolution of the software. On the other hand, multiplicity of categories happens directly in the `.cabal` fields.

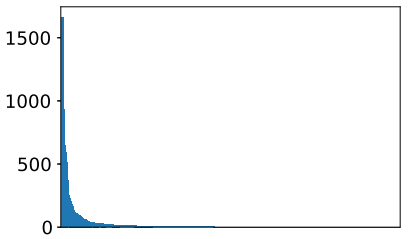
In order to clarify the different baselines, Fig. 5 depicts the overall situation. Starting from the 85135 total packages found in Hackage, grouping them by category as packages, and due to the multiplicity of categories, we get 121428 packages by category. Similarly, by grouping the starting packages by category as whole-packages—that is, merging the categories of all versions of a given software product—we get 16491 whole-packages by category. On the other hand, when taking packages



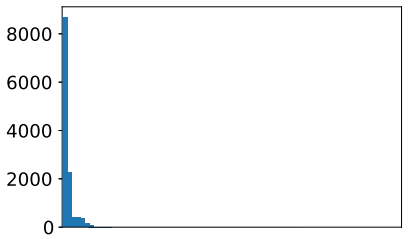
(a) Package quantity by category. Baseline of total packages by category: 121428.



(b) Package quantity by stability level. Baseline of total packages: 85135.



(c) Whole-Package quantity by category. Baseline of total whole-packages: 16491.



(d) Whole-Package quantity by stability level. Baseline of total whole-packages: 12462.

Fig. 6. Empirical distribution of the number of packages with respect to category and stability level. There are 649 categories and 77 stability levels.

Table 1
Top 10 categories by package quantity, comprising 49.2% of all packages. The baseline considers the multiplicity of packages that belong to more than one category.

Category	Package quantity	Percent of total (baseline of 121428)
web	12564	10.3%
data	10243	8.4%
network	7947	6.5%
text	5259	4.3%
development	4847	4.0%
control	4222	3.5%
graphics	4068	3.4%
system	3742	3.1%
language	3496	2.9%
database	3354	2.8%

grouped by stability, we get the same 85135. However, merging the starting packages by stability level, we get 12462 whole-packages.

The main finding after studying the distribution of `mtl`-packages with respect to metadata is that *category* and *stability* fields are open to spurious conclusions, due to the biased data found in them. Indeed, as both fields correspond to free-form strings that are arbitrarily filled by package maintainers, the empirical data in Fig. 6 shows that most packages are clumped together in a small set of categories or stability levels, which hampers further analysis. The detailed view of this situation is shown in Tables 1 to 4. We highlight that:

- In Table 1, when considering the categories of packages, 25.2% of the packages correspond to the three first categories: *web*, *data*, and *network*.
- In Table 2, when considering the categories of whole-packages, 24.9% of the packages belong to the same three categories, but in a different order.
- In Table 3, when considering the stability level of packages, 64.4% of the packages correspond to the first stability level, `n/a`, which actually reflects the absence of information in the metadata field.
- In Table 4, when considering the stability level of whole-packages, 69.7% of packages correspond to the `n/a` level, just as in the previous case.

Consequences. Our findings show that the metadata fields are not quite helpful to establish correlations between category/stability and the presence of `mtl`-packages—given that any package is highly likely to correspond to the top-10 categories and stability levels aforementioned. Nevertheless, we believe this still is a valuable finding because:

Table 2
Top 10 categories by whole-package quantity, comprising 50.5% of all whole-packages. The baseline considers the total number of whole-packages, as well as the multiplicity of their categories.

Category	Package quantity	Percent of total (baseline of 16491)
data	1662	10.1%
web	1507	9.1%
network	935	5.7%
text	790	4.8%
control	648	3.9%
system	647	3.9%
development	590	3.6%
language	536	3.3%
graphics	517	3.1%
math	490	3.0%

Table 3
Top 10 stability levels by package quantity, comprising 99.5% of all packages.

Stability level	Package quantity	Percent of total (baseline of 85135)
`n/a`	54809	64.4%
"experimental"	16670	19.6%
"stable"	5597	6.6%
"provisional"	3624	4.3%
"alpha"	1791	2.1%
"beta"	1343	1.6%
"unstable"	526	0.6%
"unstable interface"	45	0.1%
"seems to work"	92	0.1%
"highly unstable"	72	0.1%

Table 4
Top 10 stability levels by whole-package quantity, comprising 99% of all packages.

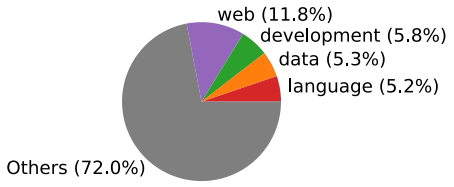
Stability level	Package quantity	Percent of total (baseline of 12462)
`n/a`	8684	69.7%
"experimental"	2259	18.1%
"provisional"	398	3.2%
"stable"	394	3.2%
"alpha"	347	2.8%
"beta"	149	1.2%
"unstable"	86	0.7%
"seems to work"	12	0.1%

- Organizations in charge of repositories such as Hackage can act upon this fact, which highlights the need for a more standardized protocol to assign such metadata, either by (semi-)automatic or manual means.
- It might suggest that package developers or maintainers tend to either not report the stability of their software, or to underestimate it, given that the second most populated stability level is *"experimental"*.

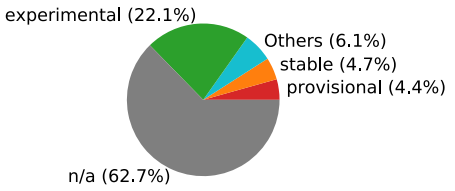
Current analysis. Despite the biases found in data, in the remainder of this section we show the results of our analysis regarding the distribution of packages with respect to metadata because: (i) it represents the actual state of the repository, and (ii) it shows our analysis pipeline is able to correctly interpret the distribution, regardless of data quality. It is crucial to remark that in the following, the categories and stability levels are not hardcoded directly into the analyses. It just happens that most `mtl`-packages happen to be attached to such metadata, which coincides with the contents of Tables 1 to 4. In other words, the analysis was not directed specifically to the aforementioned categories or stability levels.

4.1.1. As packages

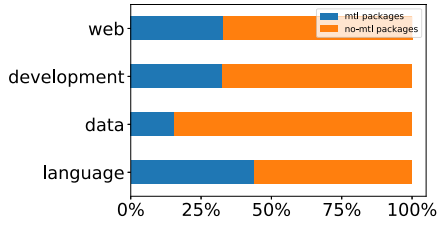
From the total of 85135 packages, we found that 26199 directly depend on the `mtl` package. In other words, 30.8% of analyzed packages directly import the `mtl`, while 69.2% do not import it. However, due to the multiplicity of categories, we consider a total number of 36266 `mtl`-packages. Regarding their distribution, Fig. 7 summarizes the distribution by category and stability level. In the pie charts, the percentages are related to the 36266 `mtl`-packages. The stacked barcharts show the proportion of `mtl` and no-`mtl` packages inside each category and stability level. We established a 5% threshold to determine relevant categories, and a 3% for relevant stability levels; those labels are shown in the plots. Other items below the threshold are merged into the "Others" label.



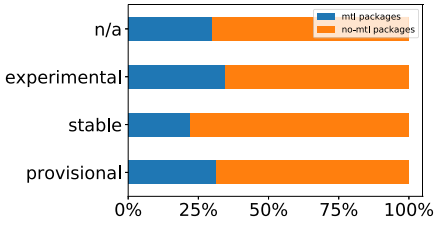
(a) `mtl` packages by category. Baseline: 36266 `mtl`-packages, when grouped by category. Threshold at 5%.



(b) `mtl` packages, by stability. Baseline: 26199 `mtl`-packages, when grouped by stability level. Threshold at 3%.



(c) Percentage of `mtl`-only packages by category. Baseline: total number of packages of each category.



(d) Percentage of `mtl`-only packages by stability. Baseline: total number of packages of stability level.

Fig. 7. Distribution of `mtl` packages regarding category and stability level. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

By categories. In the analyzed data there are 649 categories, of which 361 feature `mtl`-packages and 288 do not. To understand how `mtl`-packages are distributed with respect to categories, we must consider the following:

- What percentage of the total quantity of `mtl`-packages can be attributed to each category?
- What percentage of each category is comprised of `mtl`-packages? Here we aim to find whether there are categories with disproportionately many monadic packages, in contrast to other categories.

Answering the first question, Fig. 7a shows the `web` category is by far the one with the largest amount of `mtl`-packages, with 11.8%, followed by `development`, `data`, and `language`. Other than the `web` category, all relevant categories have between 5-6% of packages. When taken together, all categories below the 5% threshold—which comprise a wide range of concepts such as games, gpu programming, number theory, or accessibility—contain 71.9% of `mtl`-packages.

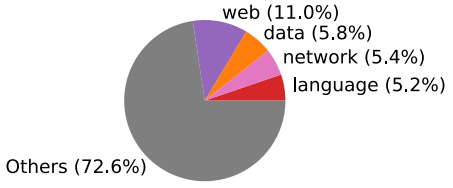
For the second answer, Fig. 7c depicts the distribution of `mtl`-packages for each of the most relevant categories. This shows that no category has more than 50% of `mtl`-packages. This might appear counter-intuitive, because a category such as `web`, which has to deal with user input and where one could assume that monads are more used in such packages, has only around 25% of `mtl`-packages.

By stability levels. Similar to the categories' analysis, we study stability levels with respect to the total number of `mtl`-packages, as well as the percentage of `mtl`-packages within each stability level. There are 77 declared stability levels, of which 41 feature `mtl`-packages and 36 do not. In their distribution, 62.7% of `mtl`-packages do not have any assigned level. In Fig. 7b this appears as the `n/a` label, but in practice the stability field is absent in the `.cabal` files of those packages. In addition to this, it appears that the `mtl` is mostly used as a dependency in categories of software that is not so stable, such as: `experimental` or `provisional`; combined, these categories represent 26.5% of `mtl`-packages. Only a 4.7% of `mtl`-packages are declared into the `stable` category. Regarding the proportion of `mtl`-packages inside each stability level, all of them have less than 50%, as shown in Fig. 7d.

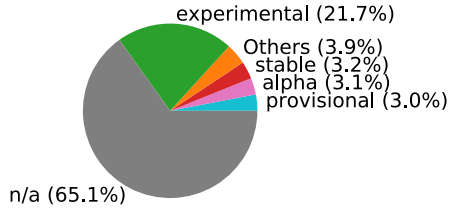
Note that in the raw data from Hackage there are repeated stability levels, such as "`experimental`" and "`experimental.`" (with a dot), showing the lack of a systematic mechanism to assign the stability level of a package. For this work we chose not to process this data by means of clustering or other grouping algorithms. We cannot conclude much from this, as the free form of the stability field is difficult to interpret.

4.1.2. As Whole-packages

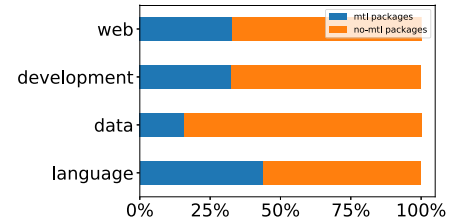
We now describe the same information regarding distribution by categories and stability levels, but now considering all whole-packages. As we must perform a grouping operation, to consider all versions of the software products, we consider the union of all categories and stability levels in the package, regardless of the particular version. In cases where the versions differ in their dependency to `mtl`, we still consider it as `mtl`-package. Fig. 8 depicts the distribution of whole-packages with respect to category (Fig. 8a) and stability level (Fig. 8b). It also includes the proportion of `mtl`-packages inside each category (Fig. 8c) and stability level (Fig. 8d). We can see that both cases are quite similar to those of Section 4.1.1 (Fig. 7); indeed,



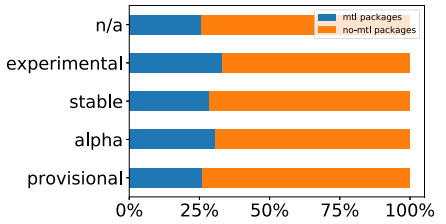
(a) `mtl` whole-packages by category. Baseline: all 4498 `mtl` whole-packages, grouped by category. Threshold at 5%.



(b) `mtl` whole-packages, by stability. Baseline: all 3449 `mtl` whole-packages, grouped by stability level. Threshold at 3%.

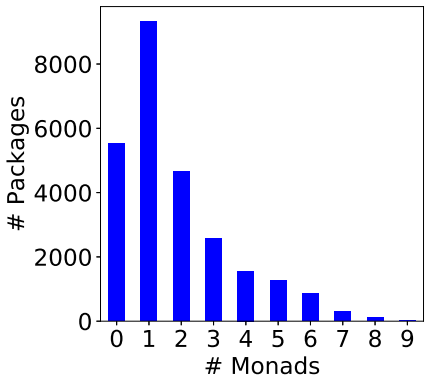


(c) Percentage of `mtl` whole-packages by category. Baseline: total number of whole-packages of each category.

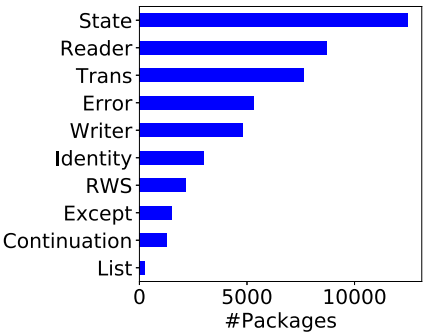


(d) Percentage of `mtl` whole-packages by stability. Baseline: total number of whole-packages of each stability level.

Fig. 8. Distribution of `mtl` whole-packages regarding category and stability level.



(a) Packages per quantity of monads. How many packages are there that import a given number of different monads.



(b) Packages that import an specific monad. State is the most imported one, followed by all others.

Fig. 9. Distribution of monad usage for `mtl`-packages, considering monads per-package, and per-monad usage.

we only see the `network` category displacing the `development` category, and the `alpha` stability level appearing as it is now above the 3% threshold.

Despite the similar results, it is relevant to make the comparison between these two forms of interpreting what a "package" is. We can now safely discard situations such as, *e.g.*, a small amount of whole-packages with many versions, which would skew the results when considering each version by itself.

4.2. Regarding RQ2

As described before in Section 2, the `mtl` library provides 8 notions of computation: Identity, Error, List, State, Reader, Writer, RWS, and Continuations. In addition we consider also the *Except* module, a more modern mechanism for handling errors, and the *Trans* module for the construction of monad transformers. Despite being enumerated as a single entity, in the `mtl` each of these elements is defined into one or more modules, which are ultimately imported in at least one module of `mtl` packages. In order to keep the current classification of computations, we have grouped together all usages of corresponding modules into a single label per monad, following the mappings described in the package data description, in Section 3.

Table 5
Comparison between top-level and class modules for the Continuation, Error, Reader, Writer, State, RWS, and Trans notions of computation.

Computation	Top-Level Module	Class Module
Continuation	Re-exports `MonadCont` and defines all the functions for running computations, *e.g.* `runCont`	Only exports the `MonadCont` type class
Error	Re-exports `MonadError` and `Error`, and defines helper functions. Also exports monad transformers for this computation. Notably, it does not re-export `liftEither`	Exports the `MonadError` and `Error` type classes, and the helper `liftEither` function that transforms values of type `Either` into a `MonadError` computation
Reader	Re-export contents from the class module. Also exports helper functions and definitions for the reader monad transformer	Exports the `MonadReader` type class and the `asks` helper function
Writer	Re-exports the **lazy variant** of the Writer computation. Both lazy and strict variants export the `MonadWriter` type class and the same set of helper functions	Exports the `MonadWriter` type class and the `listens` and `censors` helper functions
State	Re-exports the **lazy variant** of the State computation. Both lazy and strict variants export the `MonadState` type class and the same set of helper functions	Exports the `MonadState` type class and the `modify`, `modify'`, and `gets` helpers
RWS	Re-exports the **lazy variant** of the RWS computation. Both lazy and strict variants re-exports the class module and define the same set of helper functions	Exports the `MonadRWS` type class and re-exports the Reader, State, and Writer class modules
Trans	Re-exports the two class modules defined in the package.	Exports two class modules: `Control.Monad.Trans.Class` and `Control.Monad.IO.Class`

4.2.1. Overview

As a first coarse-grained analysis we count how many different monads are imported in `mtl`-packages, as well as the usage distribution for each specific monad. This is done by counting the number of indicator flags (Section 3) for each notion of computation. That is, we count whether the monad is imported in at least one module of the corresponding `mtl` package. For `mtl`-packages this result is shown in Fig. 9a. As a first remark, we see a decreasing trend, *i.e.*, there are not many packages that import many monads at once. Indeed, this is consistent with the descriptive statistics shown in Section 4: 75% of `mtl` packages import between 0 and 2 different monads. Strangely, there are 5534 `mtl`-packages that do not import any monad at all. This is due to parsing errors and other situations discussed in Section 6. Regarding the usage of each specific monad, Fig. 9b shows that without any doubts the state monad is the most used one. On the bottom, the List monad is the least used, and the Continuation monad is the second least used.

4.2.2. Distribution per notion of computation

Although we have shown the distribution for each notion of computation, the count is too coarse-grained because half of these are composed of several modules—only List, Trans, Identity and Except are directly mapped to a single module. Hence, it is relevant to understand the distribution inside each notion of computation. There are two situations of interest:

1. The usage of top-level and `Class` modules. For instance, the Continuation computation is split into the top-level `Control.Monad.Cont` module and the class module `Control.Monad.Cont.Class`. This a common programming pattern found in the `mtl` and other Haskell libraries, whose purpose is to promote information hiding by separating the internal implementation of a typeclass in the class module, from the abstract programming interface in the top-level module. Nevertheless, it is not uncommon that the top-level just includes and re-exports the class module.
2. Different implementation strategies for the same notion of computation. More precisely, for the State, RWS and Writer monads, there are lazy and strict variants. In this case the top-level module defaults to one of those strategies, but each of them can be explicitly imported if needed.

Top-level vs class modules. Table 5 summarizes the differences between the top-level and class modules for all notions of computations where it is pertinent. In general, class modules define and export the type class for the given computation, along with some helper functions. On the other hand, top-level modules define functions meant for end-users, namely functions to run the computations and perform computation-specific operations.

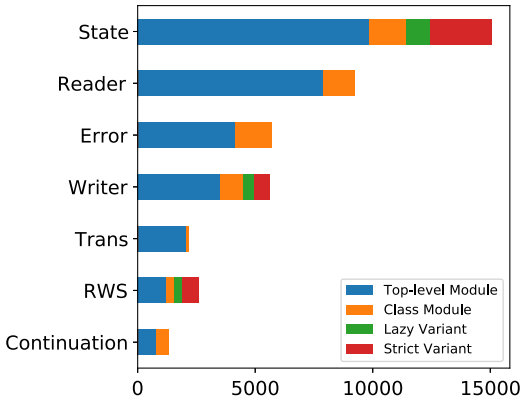


Fig. 10. Distribution of specific modules for each notion of computation.

Given this information, Fig. 10 depicts the detailed usage of each computation, in terms of its corresponding modules. The distributions are shown using stacked barcharts, keeping the same layout as in Fig. 9b.

In contrast to Fig. 9b where we used the indicator flags to count the usages, in Fig. 10 we count each import separately—even when the same package has two imports for the same computation. As a consequence, the total numbers in the latter figure may be larger than in the former.

We can see in Fig. 10 that for all computations the top-level module is indeed the most imported one. This supports the idea that most packages that import an `mtl`-package do so in order to use it as a library—that is, relying on its most general public-facing interface. On the other hand, we conjecture that packages directly importing the class modules are more likely to implement lower-level monadic operations. Indeed, this is reflected in the top-level modules themselves that must import the class-level ones.

In the case of the State, Reader and Writer computations we see that the strict variant is larger than the lazy one. However this is misleading, because the top-level modules also provide by default the lazy strategy. Thus, the lazy implementation strategies for these computations are indeed the most used ones in practice. This suggests that strict variants are only necessary, or even useful, in more specific scenarios, perhaps due to performance issues arising from using the lazy variants.

4.2.3. Most common monad combinations

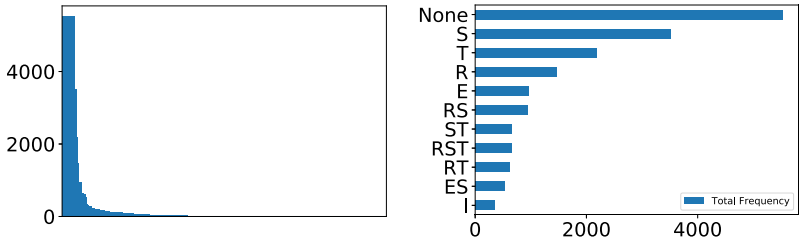
So far we have only considered individual notions of computation, either as a group of related modules, or as the specific modules that form part of a group. Now we are interested in the specific combinations of monads that are imported by the packages. To do this we compute a *usage vector*—a binary vector that combines all the indicator flags—and then we compute the frequency of all usage vectors in the dataset. To ease readability, we assign a one-letter code to the monadic computations, as described in Table 6. Therefore, we can refer to specific combinations as a string that concatenates the letters for each computation present in the combination. For instance, the *"CESTX"* combination contains the **C**ontinuation, **E**rror, **S**tate, **T**rans, and E**X**cept computations. We found a total of 244 monad combinations. As it is shown in Fig. 11a, the frequency of combinations is mostly uniform, except for a few outliers at the top that amount to the most frequent combinations. In Fig. 11b we show the top 11 combinations in the ranking, which amount to the 65% of the total usage vectors considered. At the top of the ranking there are around 6000 packages that have no computation at all. This is consistent with Fig. 9a and is discussed later in Section 6. The rest of combinations in the figure show the top-10 most used combinations, which we discuss next:

1. **S**: the State on its own is the most used combination of monads in `mtl`-packages. Of course this must be the case for consistency with the previous section, but we are a bit surprised to found that this computation is most of the time the *only* monad used in a package.
2. **T**: surprisingly for us, the Trans computation, that is the usage of the `Control.Monad.Trans` type class for defining monad transformers [8], is the second combination in the ranking—and it is also a single-element combination. We devote Section 5 for a deeper discussion regarding this situation.
3. **R**: the Reader computation, also on its own, is the third member of this ranking. Its usage is around half that of the **S** combination, and it indicates a more specialized use of state, as a read-only value.
4. **E**: the Error computation is also at the top on its own, meaning that its usefulness is not tightly coupled to any other computation. This shows the existence of many packages that can be functionally pure, but that directly address potential errors using a monadic approach.

5–9 **RS**, **ST**, **RST**, **RT**, **ES**: the last members of the ranking are just combinations of the previous ones. In particular the State computation is only missing in the 9th place, the **RT** combination, which still includes the specialized state-like Reader monad.

Table 6
One-letter codes to describe combinations of monads imported at the same time by packages. For the RWS computation we use letter **Z**, as all other letters are already used.

Computation	One-Letter Code
Continuation	C
Error	E
Identity	I
List	L
Reader	R
State	S
Trans	T
Writer	W
Except	X
RWS	**Z**



(a) Distribution of monad combinations

(b) Packages that import a specific monad combination

Fig. 11. Top 11 most used combinations of monads, comprising 65% of all monad combinations. Following the available data, the most common combination is to have no monads at all. Then the standalone State, Trans, Reader and Error computations area the next most frequent ones.

10. **I**: the last one is the Identity computation on its own. There are around 350 packages that only import this monad. Upon manual inspection we observe this happens due to one or more of the following situations:
    - using another library that requires a monadic setting, thus using the Identity constructor to lift pure values,
    - using the `runIdentity` function provided by this module to unwrap computations obtained from other libraries, or
    - using the Identity class as the base for the declaration of custom typeclasses.

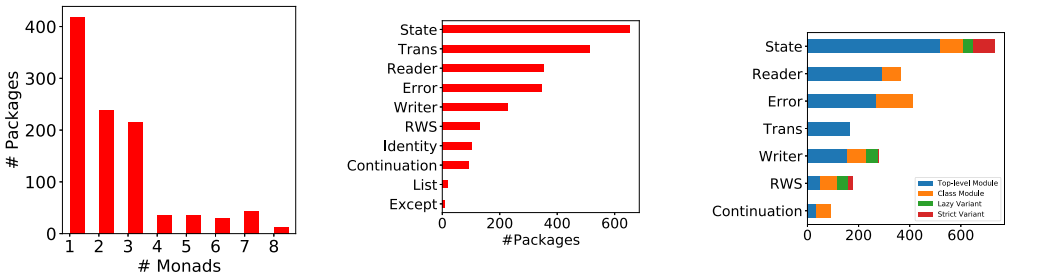
Going in more detail, the popularity of the State effect is such that there are only 80 combinations without this component—that is, almost two thirds of combinations include State. On the other hand, for the Trans and Reader monads, there are respectively 125 and 108 combinations that do not feature them. Finally, we explored whether there were significant *association rules* between monads in a combination. However our results show only significant relations between the Writer, State, Trans, Reader and Error monads, forming rules with a single antecedent and a single consequent. In other words, besides a strong bias towards state-related effects, we found no other significant correlations.

4.3. Regarding RQ3

Now we deal with the situation of packages that do not depend on `mtl` but that appear as using one or more monads provided by the library. This happens because the packages are using other libraries that provide a similar programming interface—*i.e.* the module names, such as `Control.Monad.State` are the same. Consequently, this section replicates all the analyses done for `mtl`-packages in Section 4.2. For brevity the results are shown in a summarized form, to then present the proper discussion at the end of the section.

Usage distribution. We found a total of 58936 non-`mtl` packages, whose information regarding packages and usage is shown in Fig. 12a and Fig. 12b. We found that there are 1028 packages that are using at least one monad. Fig. 12a shows that the per-monad usage distribution is very similar to that of `mtl`-packages (Fig. 9b). Note that for scaling reasons the bars in Fig. 12a do not show the 57908 packages that do not import any `mtl` monad at all.

Packages using alternatives to `mtl`. The `mtl` package has two major versions: 1.x and 2.x. In the 1.x series, the package defines all necessary types, type-classes and related elements, based on two non-standard—*i.e.* non-Haskell98—extensions: *multiparameter typeclasses*, and *functional dependencies*. Later on, around 2010 when `mtl` 2.x was in development, the original package was split, leaving the `transformers` package as a core dependency which shares the minimum common

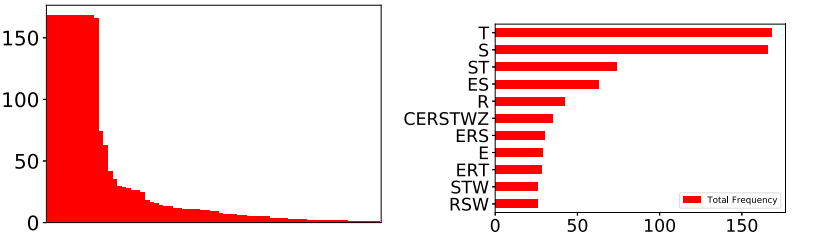


(a) Packages per quantity of monads.

(b) Packages that import an specific monad.

(c) Top-level vs Class modules.

Fig. 12. Distribution of monad usage for non-`mtl` packages, considering monads per package, per-monad usage, and the composition of monads regarding top-level and class modules.



(a) Distribution of monad combinations

(b) Packages that import a specific monad combination

Fig. 13. Top 11 most used combinations of monads. Following the available data, the most common combination is to have no monads at all. Then the standalone State, Trans, Reader and Error computations area the next most frequent ones.

definitions that are Haskell98 compatible, leaving the door open to the development of monad libraries based on other extensions, in addition to functional dependencies.[8] In addition, some other alternatives have been developed. In the following list we consider alternative implementations that explicitly aim to be as compatible as possible with the `mtl` programming interface:

- `transformers`: a core dependency of `mtl` itself, we consider it as it may be used directly in packages in lieu of `mtl`.
- `monads-fd`: a deprecated package, that implements monads based on functional dependencies. It is meant to be equivalent to `mtl` itself. Indeed, it currently re-exports the `mtl` package.
- `monads-tf`: implementation of monads using type families, rather than functional dependencies.
- `mtl-tf`: a deprecated package, precursor to `monads-tf`.
- `mmtl`: a package providing *modular monads transformers*, in a generalization of `mtl`.
- `mtlx`: a library of *indexed* monad transformers, that allows the creation of several instances of the same transformer, with different labels or indexes, rather than relying on the redefinition of monadic types and all associated boilerplate.

Considering these alternatives, we found that 680 packages, out the 1028 packages mentioned before, depend on at least one of the alternative implementations of monads. Hence, the 66.0% of no-`mtl` packages that actually use monads rely on the aforementioned alternative implementations. The remaining 348 packages, amongst which we found both `mmtl` and `mtl-tf`, seem to mostly provide their own implementation of the monads used. For instance, the `cabal-install-bundle` package purposely includes all its dependencies.

Usage of monad combinations. Similar to Section 4.2.3, Fig. 13a depicts the distribution of monad combinations present in non-`mtl` packages (Fig. 13a) and the top-10 most used combinations (Fig. 13b). In contrast to our previous analysis, for non-`mtl` packages we consider only the packages with at least one monad, because the empty combination distorts the relevant information that we explore at this point. Again, we see a pattern very similar to that of `mtl` packages. The most noticeable difference is the ranking of the top-11 combinations:

- Trans takes the top spot, displacing State to a very close second place

[8] https://mail.haskell.org/pipermail/libraries/2010-September/014281.html.

- Although the State, Trans, Reader, and Error monads are still in the top 10 as standalone monads, all the other combinations are on average longer than those for `mtl` packages
- We see the RWS and Writer monads in the top-10, in contrast to the situation for `mtl` packages

Overall, we see the same general pattern as before: the State monad dominates the ranking, as it appears in 7 out of 10 combinations, while Trans and Error tie at the second place with 4 instances. Also, most of the non-`mtl` packages that use monads are covered in the top-10, which covers around 50% of the total package frequency, confirming the distribution shown in Fig. 13a.

5. Understanding the usage of `Control.Monad.Trans`

At this point it is clear that the State effect, in all of its varieties and implementations, is the most prevalent one in all Hackage packages (Section 4.2). However, surprisingly for us, the second place in our ranking goes to the module responsible for monad transformers: the `Control.Monad.Trans` module. Throughout this paper we have referred to this as the "Trans" computation. This module not only provides several essential definitions for the mechanism of *monad transformers* [8], but also provides a bridge between *impure* side-effecting computations via the `IO` monad, to the monadic setting of the `mtl`. More specifically, this module provides the following main components:

- The `MonadTrans` type class [12] which is used to register all instances of monad transformers that can be compatible with the design of `mtl` and other existing monads and transformers. The only method in this typeclass is the `lift` operation that must satisfy certain laws when applied on top of an arbitrary monad `m`. Thus, any type `t` that is an instance of `MonadTrans` must guarantee that, for any monad `m`, the combination `t m` is also a valid monad.
- The `MonadIO` type class that allows the embedding of the `IO` monad at the bottom of the monad stack. Any monad `m` that is instance of `MonadIO` can invoke the `liftIO` method provided by this class in order to access the underlying native I/O in the `IO` monad.

Now we would like to know *why this module is so widely imported in `mtl`-packages.* Without any empirical information, a reasonable Haskell developer would probably conjecture three usage scenarios that require the importing of this module:

1. Programmers import `Control.Monad.Trans` mainly to use the `MonadIO` type class and/or the `liftIO` operation, because the operation of the package involves I/O at some point. Note that this is provided by the `Control.Monad.IO.Class` module—which can be imported directly—but for some reason is re-exported by the module for monad transformers.
2. Programmers implement their own monad transformers, and then declare them as an instance of `MonadTrans` to integrate their work with the rest of the `mtl`. For instance as in [13,14].
3. Programmers develop higher-order functions that, for some reason, take an arbitrary monad transformer as argument; or alternatively, declare new instances of type classes, based on existing monad transformers. Both cases require having `MonadTrans` in the type signature. This also happens in [13,14].

Unfortunately, the data obtained in our processing pipeline (Section 3) is not precise enough to give a categorical answer or support to these scenarios. Indeed, doing so would require a much more finer-grained approach to the parsing and analysis of packages, which goes beyond the scope of this exploratory study. For instance, confirmation for Scenario 1 would require detailed information on which methods, and of what type classes, are used in the packages. Similarly, for Scenario 2 we would need to catalog all type class instances defined in all packages. For Scenario 3 the required information grows even more, as we would need the type signature of all functions defined in all packages. Indeed, working towards a richer semantic model for mining package information is an interesting venue for future work. Indeed, this is actually an excellent technical justification for using GHC-as-a-library, rather than HSE, as this tool has rich support for querying type definitions and types of exported functions.

Despite the aforementioned limitations, we can try to give support to the first two scenarios by further exploiting the data that we do have available. In the following we argue that the above intuition could be very close to the actual usage of the Trans computation. Indeed, let us recall from Section 4.2 that there are 2185 "T-only" packages that feature only the Trans computation. From this we can see the following:

Regarding scenario 1. We conjecture that packages importing Trans to access `MonadIO` or `liftIO` are likely to *also import other modules related to I/O*; hence we would like to know this quantity. A small complication arises because the Haskell Prelude, a default set of available values and functions, provide several elements of the I/O system such as the `IO` data type, and several functions for opening/reading/closing files. We consider as IO-related modules the following list of modules provided by the `base` package, which is available in all Haskell installations:

- `System.IO, System.IO.Error, System.IO.Unsafe`: these modules declare the core data types and functions for the operation of `IO` computations

- `Control.Monad.IO.Class`: this module declares the `MonadIO` type class
- `Data.IORef`: this provides the facility for a mutable variable inside the `IO` monad.

By filtering the dataset of T-only packages with respect to the other imported modules we found that there are 1175 packages—more than 50%—in this situation. We believe this strongly suggest the validity of our first conjecture.

Scenario 2: Custom Monad Transformers. To obtain an educated guess regarding the second scenario, we conjecture that packages that implement their own monads or monad transformers are *more likely to provide modules in the* `*Monad*` *namespace*. Hence, again we filtered the set of T-only packages selecting those that export at least one module with the substring "Monad" in its name. This yields a total of 153 packages. Notice that there are only 64 packages that also match the filtering of Scenario 1. We performed a manual inspection on these packages to determine whether they: (i) declare a new instance of `MonadIO`, (ii) use the `liftIO` operation, and (iii) declare a new custom monad transformer, that is, a new instance of `MonadTrans`. We found that:

- There were only 15 false positives out of the 153 filtered packages. Still, all these packages are closely related to monadic programming, for instance providing new functions or plain monads that are not constructed with monad transformers.
- On the contrary, 62 packages satisfy all three criteria, that is, they define instances of `MonadIO`, use `liftIO` and declare instances of `MonadTrans`.
- Of the remaining packages:
  - 20 only define a new monad transformer
  - 28 only use `liftIO`
  - there is only 1 package that defines instances of `MonadIO`, declares a monad transformer, but does not uses `liftIO`. Although probably the operation is defined implicitly by some extension of GHC such as *Generalized Newtype Deriving*.
  - 27 packages declare instances of `MonadIO` and use `liftIO`

The manual inspection was quite instructive because we found strange situations such as packages that explicitly import `Control.Monad.Trans` only for the I/O related classes. That is code, similar to:

```
import Control.Monad.Trans(MonadIO(..), liftIO).
```

This is strange because it would be more straightforward to import the module `Control.Monad.IO.Class`, which is the one that actually defines the entities related to I/O. Even more strange, some packages had *two imports* for `MonadIO`, one from `Control.Monad.Trans` and the other from `Control.Monad.IO.Class`—this even happened on the same module file. We also observed many definitions that would support Scenario 3, however, as we explained above, this is quite difficult to quantify with our current methodology.

6. Threats to validity

Our work presents some important limitations and threats to its validity. A first limitation is the use of indicator flags to signal whether a package uses a monad or not. The consequence of this is that we are not really weighting the proportional impact of each monad in the package. However, doing so would require more complex parsing and analysis of source code. Nevertheless, the more problematic limitation is that our processing pipeline is not able yet to fully parse every module in every package, because—surprisingly—there is no standardized solution to parsing and analyzing Haskell packages and files. Major difficulties arise from the use of the C preprocessor, and from compiler-specific extensions. The consequences of this are more severe:

- There are packages with conditional dependencies, which are parsed as having no dependencies at all. This probably affects the number of `mtl` and non-`mtl` packages.
- There are modules that cannot be parsed because they rely on external files, such as C headers, or that cannot be parsed for other reasons. As a consequence we may be missing monads that are used/imported in such packages.

A concrete situation that arises from these limitations is that in Fig. 9 there are 5534 mtl-packages that use 0 monads. The situation is explained as follows:

- There are 3585 with parsing errors, which might or might not use monads at all.
- There are 1085 packages that only import the `Control.Monad` module—which is not counted in any of the computations. This module provides several typeclasses, definitions and functions to work over monadic values. This module is imported to work with existing libraries that pass around monadic values.
- There are 864 packages that import the `mtl` and actually do not import any monad at all.

Still, we consider that our methodology and results are coherent and properly address our research questions.

7. Related work

Our work lies in the field of Mining Software Repositories—a vast area of research, featuring its own international conference and other venues—with a very specific focus on the study of programming language features. As mentioned in Section 3, there are very few empirical studies regarding Haskell and its features, amongst which we highlight the work of Morris [9], which assesses the use of *OverlappingInstances* to guide the design of the Habit language, as well as the work of Bezirgiannis et al. [10]. Similar empirical studies on language features include the work of Callaú et al. [15], regarding how developer uses the dynamic features of Smalltalk, and also Callaú et al. [16], regarding the use of *type predicates*, that is, methods that specifically query the type of an object, such as a Java's `instanceof` method. Other studies, such as Nagappan et al. [17] consider the use of `goto` and whether or not it is really harmful, as posed originally by Dijkstra [18]. Similar research is that of Casalnuovo et al. [19] regarding the use of assertions in GitHub projects. Finally, there is some research by Robbes et al. [20] on whether objects meet their promises regarding modularity and reuse.

8. Conclusions and future work

We have described an empirical study to determine *which monads do Haskell developers use*. By collecting information directly from the Hackage package repository we have established that: (i) 30.8% of packages depend on the `mtl` library (RQ1); (ii) the state monad is the most used one and that the monad transformer module is widely used, being second only to state (RQ2); and (iii) that a tiny fraction of packages, 1.2%, uses alternative monad libraries, thus showing the prevalence of `mtl` in real practice (RQ3). We also explore package metadata, finding that they are not strong predictors of monad usage, as most packages are clumped in a few categories or stability levels. Apart from research questions, we can infer and discuss the following additional findings:

Alternatives to the `mtl` package library. The empirical evidence shows a small percentage of packages uses alternative monad libraries. Due to the exploratory scope of this work it is not clear *why* this situation happens, although we can envision some possibilities. On the one hand, perhaps the code resulting from using the standard library is too complex, or maybe the software requires a novel typechecking mechanism not provided by the `mtl`. On the other hand, it could be the case that the alternative packages are developed as a product of research, and the packages that use it are meant as a proof of concept that is not continuously developed alongside the rest of the Haskell ecosystem.

Potential benefits for language designers. A core motivation of this work was to provide language designers with empirical information for the design and development of novel or refined mechanisms related to the use of monads. In this regard, we believe this work provides at least three key insights:

1. There are 3 computations that *do not appear* in the top-11 most used monad combinations: List, Continuation, and RWS. Indeed, the quantity of packages using these notions of computations is significantly lower than for the other (Section 4.2.3). This suggests an opportunity for improvement in terms of software engineering: the development of a new *core*-`mtl` package library focused on the most used monads, while still providing the less used ones in external non-core packages. This way, major improvements could be done to the core package, such as certified development, testing, etc. This is particularly interesting, given that, at least conceptually, the List and Continuation monads are rather complex, which could difficult the improvement of the other parts of the `mtl`.
2. Clearly, the usage of State monad dominates all other cases. This could provide insights for simpler mechanisms to use it, or for additional syntax sugar. Ideally, this should be designed by user-centered studies. Additionally, and considering the current trends for using functional programming concepts—such as purity and immutability—in libraries and languages such as Javascript, Scala, and others, it would be beneficial to provide simple ways to use state-like monads or constructs, given that it is also likely that the need for mutable state arises in these situations. Interestingly, this latter insight goes beyond Haskell itself, backed by the current evidence shown in this study.
3. There is strong evidence suggesting the implementation of the Trans module is unnecessarily conflated with the typeclasses related to monadic IO (Section 5). This could be alleviated by providing a stricter separation between those two interfaces.
4. Going beyond languages on their own, the designers of software repositories, be it in Haskell or any other language, should be careful when providing unstructured fields for metadata, such as stability or category. The evidence shown in this study suggests that, in the absence of a clear protocol for assigning such metadata, these fields could be biased, being clumped in a few items, hampering any clear correlational analysis.

Challenges for future work. Our first goal for future work is to address the limitations and threats to validity faced on this preliminary study. The most pressing issue is to obtain a more precise method for parsing source files, even though they may depend on external C dependencies such as header files. On the second place we want to develop a scalable analysis infrastructure, to manage the huge number of packages in Hackage. For this end, we want to develop distributed analysis infrastructure, following the same processing pipeline described here. Such infrastructure may be eventually leveraged for

other empirical analysis of Haskell code in the Hackage repository. By leveraging this infrastructure, we would like to replicate and further extend this study on Stackage, a curated repository for stable Haskell packages, to know how monads are used in a setting purposefully geared towards commercial software development in Haskell. Going beyond the technical aspects, we aim to focus on the more qualitative aspects of this research: Why are specific monads being imported? How are they used? How are monad transformers developed and how is the integration with the `mtl`? In which cases developers prefer to define their own monads instead of using the standard ones? Why are customly-defined monads created? How does the usage of monads evolves over different package versions? Are there any inherent limitations of the `mtl`, if so, which ones?-

CRediT authorship contribution statement

Ismael Figueroa: Conceptualization, Methodology, Software, Writing - original draft. **Paul Leger:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing. **Hiroaki Fukuda:** Conceptualization, Investigation, Validation, Visualization, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is partially funded by FONDECYT Postdoctoral Project 3150672. We thank all the anonymous reviewers for their insightful comments.

References

[1] haskell org, The Haskell Language, http://www.haskell.org, 2017.
[2] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1) (1991) 55–92.
[3] P. Wadler, The essence of functional programming, in: Proceedings of the 19th ACM Symposium on Principles of Programming Languages, POPL 92, ACM Press, Albuquerque, New Mexico, USA, 1992, pp. 1–14.
[4] haskell.org, Hackage, http://hackage.haskell.org, 2017.
[5] I. Figueroa, A preliminary assessment of how monads are used in Haskell, in: Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP 2017, ACM Press, New York, NY, USA, 2017.
[6] haskell.org, Haskell Cabal, http://www.haskell.org/cabal, 2017.
[7] I. Figueroa, Online resources: which monads Haskell developers use, http://zeus.inf.ucv.cl/~ifigueroa/doku.php/research/empirical-monads, 2017.
[8] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: Proceedings of the 22nd ACM Symposium on Principles of Programming Languages, POPL 95, ACM Press, San Francisco, California, USA, 1995, pp. 333–343.
[9] J.G. Morris, Experience report: using Hackage to inform language design, in: Proceedings of the Third ACM Haskell Symposium on Haskell, Haskell '10, ISBN 978-1-4503-0252-4, 2010, pp. 61–66.
[10] N. Bezirgiannis, J. Jeuring, S. Leather, Usage of generic programming on Hackage: experience report, in: Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming, WGP '13, ISBN 978-1-4503-2389-5, 2013, pp. 47–52.
[11] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, M.W. Godfrey, The MSR cookbook: mining a decade of research, in: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, ISBN 978-1-4673-2936-1, 2013, pp. 343–352.
[12] P. Wadler, S. Blott, How to make ad-hoc polymorphism less ad hoc, in: Proceedings of the 16th ACM Symposium on Principles of Programming Languages, POPL 89, ACM Press, Austin, TX, USA, 1989, pp. 60–76.
[13] T. Schrijvers, B.C. Oliveira, Monads, zippers and views: virtualizing the monad stack, in: Proceedings of the 16th ACM SIGPLAN Conference on Functional Programming, ICFP 2011, ACM Press, Tokyo, Japan, 2011, pp. 32–44.
[14] I. Figueroa, N. Tabareau, É. Tanter, Effect capabilities for Haskell: taming effect interference in monadic programming, Sci. Comput. Program. 119 (2016) 3–30.
[15] O. Callaú, R. Robbes, É. Tanter, D. Röthlisberger, How (and why) developers use the dynamic features of programming languages: the case of Smalltalk, Empir. Softw. Eng. 18 (6) (2013) 1156–1194.
[16] O. Callaú, R. Robbes, É. Tanter, D. Röthlisberger, A. Bergel, On the use of type predicates in object-oriented software: the case of Smalltalk, in: Proceedings of the 10th ACM Dynamic Languages Symposium, DLS 2014, in: ACM SIGPLAN Notices, vol. 50, ACM Press, Portland, OR, USA, 2014, pp. 135–146, 2.
[17] M. Nagappan, R. Robbes, Y. Kamei, É. Tanter, S. Mcintosh, A. Mockus, A.E. Hassan, An empirical study of GOTO in C code from GitHub, Repositories (2015) 404–414.
[18] E.W. Dijkstra, Go To statement considered harmful, Commun. ACM 11 (3) (1968) 147–148.
[19] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, B. Ray, Assert use in GitHub projects, in: Proceedings of the 37th International Conference on Software Engineering – Volume 1, ICSE '15, ISBN 978-1-4799-1934-5, 2015, pp. 755–766.
[20] R. Robbes, D. Röthlisberger, É. Tanter, Extensions during software evolution: do objects meet their promise?, in: J. Noble (Ed.), Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP 2012, in: Lecture Notes in Computer Science, vol. 7313, Springer-Verlag, Beijing, China, 2012, pp. 28–52.