A preliminary assessment of how monads are used in Haskell

Ismael Figueroa∗
Escuela de Ingeniería Informática
Pontificia Universidad Católica de Valparaíso
Valparaíso, Chile
ismael.figueroa@pucv.cl

ABSTRACT
Monads are a denotational approach to embed and reason 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 to 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 8 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 how developers use monads. In this paper we present preliminary results of an empirical study that quantitatively describe how monads are used in a sample of the Hackage repository. Our results show that around 25% of sampled modules depend on the mtl package, whereas only 1% depend on alternative, yet compatible implementations. Nevertheless, usage patterns for each specific monad remain similar both for mtl and alternatives. Regarding usage, the state monad is by far the most used one, although all of them are used. We also report on the distribution of packages that use mtl, regarding their category and stability level.

CCS CONCEPTS
• Software and its engineering → Language features; Software libraries and repositories;

KEYWORDS
monads, empirical study, use of monads, Haskell, Hackage, mining software repositories

ACM Reference format:
DOI: 10.1145/3125374.3125385

∗Partially funded by CONICYT/FONDECYT Postdoctoral Project 3150672.

1 INTRODUCTION
The Haskell language [9] is perhaps the most well-known and representative member of the family 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. 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 [12, 16] 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. Arguably, Haskell is the language with the best support for general-purpose, practical monadic programming. Haskell provides a standardized interface for monadic programming, in the form of the monad transformers library—simply known as 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. In this paper we present preliminary results on an empirical investigation of the Hackage repository, the de-facto repository for open-source Haskell software. More specifically, we 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 stability or categories?

RQ2. What is the usage distribution of mtl’s monads in packages that directly depend on this library?

RQ3. What is the situation of alternative implementations to the mtl, and their usage on existing packages?

Although we could also 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 help us trim the otherwise huge number of packages that indirectly would depend on the mtl.

By understanding which monads are used and how, 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. 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. As a starting point, due to scalability and correctness limitations, mainly related to the large number packages and the difficulty of properly parsing files that rely on the C preprocessor, we focus on the latest version of all packages in the Hackage repository. Then, by parsing and analysing 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 [6].

As a reader, you only need a passing familiarity with functional programming, as well with the general concept of a monad.

2 BACKGROUND

Monadic Programming. In practical terms, monads are used in Haskell as a denotational approach to embed and reason about computational effects, such as mutable state or I/O. The approach was first suggested by Moggi [12] and Wadler [16], and then it was extended by Liang et al. [11] 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) 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 with mutable access to a provided collection of values.
- **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 kind of computation.
- **Continuations**: represents computations that can be suspended, passed around and resumed, based on the construction and manipulation of continuations.

Hackage Repository. Hackage [7] is the de-facto repository for open source software written in Haskell. Currently, it features over 11000 packages written both by researchers and practitioners. In Hackage, developers can upload several versions of a package, alongside its metadata, following the conventions of the Cabal build system [8]. 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.

Empirical research on Hackage. There is some background on empirical studies using the Hackage repository. Morris [13] analyzed Hackage to assess the usage frequency of a GHC extension named OverlappingInstances, in order to guide the design of the Habit1 language. Another study was performed by Bezirgiannis et al. [1] 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 gpaht tool, that automatizes the analysis performed in their work. This tool is similar to our own, although it is developed entirely in Haskell.

3 METHODOLOGY

We follow a simplified version of the standard pattern used in the mining software repositories (MSR) research [10], which is depicted in Figure 1. Our pipeline features two broad stages: generation of initial package data, and then, a more costly step, is 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. Due to scalability and parsing limitations, we have decided to give preliminary answers to our research questions by considering only the latest version of every package in Hackage, up until April 2017. In addition to the large quantity of packages and versions, it turns out that properly parsing Haskell files is surprisingly difficult; we are not aware of a standard mechanism to do it, besides using HSE. All programs and datasets used in this work are available in the companion website [6].

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 analysing monad usage. Computing imported modules amounts to parsing and analysing the main and provided modules of a package. To do this we download the package source and feed it to another Haskell program, which leverages HSE. With this, it is simple to tag and count all usages of monad modules.

1 http://hasp.cs.pdx.edu
2 https://hackage.haskell.org/package/haskell-src-exts
Package data description. To clarify the data generated in the processing pipeline we briefly describe the fields that are computed for each sampled package:

- **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 the Control.Monad.Trans, related to monad transformers in the mtl.
4 EMPIRICAL RESULTS AND DISCUSSION

The following results were obtained after processing 11171 packages from Hackage, which amount to the analysis of 451231 provided modules. The total size of downloaded packages is around 3GB. This is plenty of data, considering that we are only analysing the latest version of each package. Before developing the answers to our research questions, in Figure 2 we depict descriptive statistics regarding the quantity of imported/provided modules, and the number of dependencies of the analysed packages. The boxplots show that for 75% of all sampled packages, the declared dependencies and the provided modules is a quantity between 1 and 10. For imported modules, the quantity varies between 1 and 20. Given the exploratory nature of this work, this overview gives us a general overview of how a “regular” package looks like, and may be useful to gauge the weight or impact of mt1-related dependencies.

4.1 Regarding RQ1

From the total of 11171 analysed packages, we found that 2803 directly depend on the mt1 package—we denote such packages as mt1-packages. In other words, the 25.1% of analysed packages directly import the mt1, while the 74.9% do not import it. However, given that there are several packages that belong to more than 1 category, we consider a total number of 3670 unique mt1-package/category combinations, thus addressing the multiplicity of categories. We use the 3670 figure as the baseline for the pie charts in Figure 3. Regarding their distribution, Figure 3a depicts the distribution of mt1-packages by category, while Figure 3b shows the distribution by stability level. We established a 5% threshold to determine relevant categories, and a 3% for relevant; those labels are shown in the plots. Other items below the threshold are merged into the “Others” label.

By categories. In the analysed data there are 538 categories, of which 308 feature mt1-packages and 230 do not. Regarding the distribution of mt1 packages, the web category is by far the one with the largest amount of mt1-packages, with a 14%, followed by language, data, network, control, development, text, graphics, database, and system. Other than the web category, all relevant categories have around 4-7% of packages. When taken together, all categories above the 3% threshold contain 67.9% of mt1-packages.

By stability. There are 49 declared stability levels, of which 23 feature mt1-packages and 26 do not. In their distribution, the 83% of mt1-packages do not have any assigned level. In Figure 3b 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 mt1 is mostly used as a dependency in categories of software that is not so stable, such as: experimental, alpha, or provisional; combined, these categories represent around 40% of mt1-packages. Only a 4% of mt1-packages are declared into the stable category. For now we can only conclude that a mt1-package is most likely to be an experimental package, or to not have any declaration of its stability at all.

Stability of relevant categories. In Figure 3c we focus on the relevant categories and describe their composition with respect to
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. Each computation however 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.

Packages per quantity of monads. As a first analysis we want to see how many different monads are imported in mtl-packages. Following the package information described before (Section 3), we count the number of indicator flags related to each notion of computation. That is, we count whether the monad is imported in at least one module of the corresponding mtl package. This result is shown in Figure 4a, only for mtl-packages. As a first remark, we see a decreasing trend, i.e., there are not many packages that import many monads at once. Indeed, descriptive statistics show that 75% of mtl packages import between 0 and 2 different monads. Finally, there is somewhat strange data in the figure, as there are 726 mtl-packages that do not import any monad at all. We discuss this situation in Section 5.

Packages per individual monads. Regarding the usage of each specific monad, Figure 4b 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. The rest have similar usage levels.

The Trans module. Figure 4b shows the Control.Monad.Trans module, also provided by mtl, which defines the essential mechanism of monad transformers [11]—the MonadTrans typeclass [17]. It is interesting that imports of this typeclass are almost on par with others such as a state, reader or writer monads. We conjecture three usage scenarios for this typeclass:

- Programmers import Control.Monad.Trans mainly to use the MonadIO typeclass and/or the liftIO operation, because the operation of the package involves I/O at some point. For instance, packages AGI and z3 only use these features.
- Programmers make their own monad transformers an instance of MonadTrans to integrate with the mtl. For instance, the Consumer package defines a new Consumer monad and ConsumerT monad transformer.
- Programmers develop higher-order functions that, for some reason, take an arbitrary monad transformer as argument; or alternatively, declare new instances of typeclasses, based on existing monad transformers. Both cases require having MonadTrans in the type signature. For instance, the HAppS-State defines instances of the Monad typeclass, based on any arbitrary monad transformer.3

---

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 provided a similar programming interface—i.e. the module names, such as Control.Monad. State are the same. Consequently, when considering all 8368 non-< mt1 > packages, we show respectively in Figure 5a and Figure 5b the usage distribution regarding the quantity of monads imported per-package, as well as the usage of each specific monad. We found that there are 104 packages that are using at least one monad. Figure 5a shows that the per-monad usage distribution is very similar to that of mtl-packages (Figure 4b). Note that for scaling reasons the bars in Figure 5a do not show the 8264 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 type-classes, 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 definitions that are Haskell98 compatible, leaving the door open to the development of monad libraries based on other extensions, in addition to functional dependencies.4 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.
- mtlx: a library providing modular monads transformers, in a generalization of mtl.
- mt1x: 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 78 packages, out the 104 packages mentioned before, depend on at least one of the alternative implementations of monads. Hence, the 75.0 % of no-< mt1 > packages rely on the aforementioned alternative implementations. The remaining 26 packages, amongst which we found both mtl 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.

5 THREATS TO VALIDITY

Our work presents some important limitations and threats to its validity. The first limitation is intentional: we consider only the latest version of each package in the index, accounting to a grand total of 451231 modules, thus limiting the general applicability of our results. We believe that, considering the exploratory nature of this work, this limitation does not invalidate this study. Another design consideration 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.

4https://mail.haskell.org/pipermail/libraries/2010-September/014281.html
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 analysing 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 \texttt{mtl} and non-\texttt{mtl} packages.
- There are modules that cannot be parsed because they rely on external files, such as a 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. For instance, in Figure 3 there are 726 mtl-packages that use 0 monads. Of those, 438 are packages affected by at least one parsing error, while the remaining 288 show no apparent errors in their processing.

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

6 RELATED WORK

Our work lies in the field of Mining Software Repositories, with a specific focus on the study of programming language features. As mentioned in Section 3, there is very few empirical studies regarding Haskell and its features, amongst which we highlight the work of Morris [13], which assesses the use of \texttt{OverlappingInstances} to guide the design of the Habit language, as well as the work of Bezirgiannis et al. [1]. Similar empirical studies on language features include the work of Callau et al. [2], regarding how developer use the dynamic features of Smalltalk, and also Callau et al. [3], regarding the use of type predicates, that is, methods that specifically query the type of an object, such as a Java’s \texttt{instanceof} method. Other studies, such as Nagappan et al. [14] consider the use of \texttt{goto} and whether or not it is really harmful, as posed originally by Dijkstra [5]. Similar research is that of Casalnuovo et al. [4] regarding the use of assertions in GitHub projects. Finally, there is some research by Robbes et al. [15] on whether objects meet their promises regarding modularity and reuse.

7 CONCLUSIONS AND FUTURE WORK

We have described an empirical study to determine \texttt{how developers use monads in Haskell}. By collecting information directly from the Hackage package repository we have established that: (i) around 25% of packages depend on the \texttt{mtl} library; (ii) the state monad is the most used one; (iii) that the monad transformer module is widely used, although it is not clear yet why; and (iv) that a tiny fraction of sampled packages, around 1% use alternative monad libraries, thus showing the prevalence of \texttt{mtl} in real practice. We also explore package metadata, but could not conclude regarding any influence on the use of monads.

\textbf{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.

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 \texttt{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 evolve over different package versions? Are there any inherent limitations of the \texttt{mtl}, if so, which ones?

\section*{REFERENCES}


