Mining motivated trends of usage of Haskell libraries

Marc Juchli,* Lars Krombeen,*
Shashank Rao,* Chak Shun Yu*
Delft University of Technology
{M.B.Juchli, L.Krombeen, S.P.Rao, C.S.Yu}@student.tudelft.nl

Anand Ashok Sawant, Alberto Bacchelli
Delft University of Technology
{A.A.Sawant, A.Bacchelli}@tudelft.nl

***Abstract*—We propose an initial approach to mine the usage trends of libraries in Haskell, a popular functional programming language. We integrate it with a novel, initial method to automatically determine the reasons of clients for switching to different versions. Based on these, we conduct a preliminary investigation of trends of usage in Haskell libraries. Results suggest that trends are similar to those in the Java ecosystem and in line with Rogers theory on the diffusion of innovation. Our results also provide indication on Haskell libraries being all by and large stable.**

I. Introduction

Choosing the appropriate software library to use for a specific use case is not an easy task [1]. Even when this decision is made, it is still not trivial to choose what version should be used or what updating behavior should be followed [2], [3]. In fact, although newer versions of software libraries tend to introduce new functionalities, to remove obsolete features, and to ensure better security, upgrading to the latest version is not to be taken lightly. For instance, an update might deprecate a heavily used feature, might break existing functionality with unforeseen changes, and might change the protocol to interact with the provided components [2]. Even when adopting a previously unused API, clients do not necessarily adopt the latest version of the API, but put more thought into choosing the appropriate version to use given that introduction of a new API may be incompatible with existing dependencies. This behavior is observed in an existing database containing API usage [4]. Overall, making the right choice in which version of an API to use is hard.

Mileva *et al.* theorized that by using *wisdom of the crowd*, we can recommend which versions of an API should be used [2]. They proposed AKTARI, a tool for Java libraries, that provides such recommendation based on three metrics: *usage trends*, *current most popular version*, and *switch backs to earlier versions*. They provided initial evidence that these three metrics are a useful basis to help developers decide which version of a library to use.

In this paper, we make the first steps in expanding on this previous work by: (1) introducing an approach to mine library usage information for a functional programming language (*i.e.*, Haskell), (2) proposing a method to automatically infer the reasons *why* developers decide to switch a library version, and (3) conducting an initial exploration of the behavior of clients of Haskell libraries. With our effort, we strive to explore the angle of functional programming paradigm, which may present differences with respect to the object-oriented paradigm investigated in the initial approach by Mileva *et al.* [2]. In addition, we want to help developers base their decisions on the version of a library to use not only on trends, popularity, and switch backs, but also on *why* these occur.

*Marc, Lars, Shashank, and Chak contributed equally to the work and are to be considered all first authors. This work was developed as part of the Master course Mining Software Repositories at Delft University of Technology.

II. Mining Usage Information

We propose a method to mine usage information for libraries of a functional programming language. As target language, we select Haskell because it is a purely functional language and allow us to conveniently use Hackage [5] as our source of data. Hackage is the Haskell community's central programming library archive. It contains the published versions of each library, how many times these were downloaded, and the packages that each library depends on. Haskell library developers use Hackage to publish their libraries so that they can be used in another project as a dependency, similarly to how Java API developers use Maven central [6].

The projects on Hackage are APIs/libraries, which can be treated as regular Haskell projects with dependencies hosted on Hackage as well. One advantage of using Hackage based projects is that these projects use the building and packaging system Cabal [7]. The downside of this approach is that we focus on a specific type of Haskell projects (*i.e.*, libraries). A more comprehensive approach would use Haskell projects that can be found on platforms such as GitHub; however, there is no guarantee that they use Cabal, thus any data mined from them would lack in important version information. In this study, we decide to focus on Hackage based projects and leave a more comprehensive choice of projects and tackling the associated challenges to future work.

Similarly to Maven, which uses a POM file to explicitly declare a projects' dependencies, Cabal uses a file that declares all the build dependencies of the Haskell project, including the version of the API that is being used. One can, thus, determine the popularity of existing Haskell libraries. Moreover, since the content of Hackage is stored in a Git repository, one can get evolutionary data on the packages hosted on Hackage, their versions, and their cabal files.

Our approach parses all the cabal files for each hosted project on Hackage and all its versions, and collects their build dependencies. Subsequently, it computes how many times a package is being used in its various versions. Information on when a new release was made and when the cabal file changed is not available from the Git repository. Therefore, our approach crawls the Hackage website to find the webpage for each of the versions of a library and parse the release date.

A. Determining the used library version

Resolving the exact version of the library that is being used by a project is not trivial. In fact, the way in which one can specify range of versions and wild card characters allows an undetermined number of library versions to be valid dependencies; as in the following three valid cabal package definitions: (1) '`pkgname >= v`', (2) '`pkgname >= v1 && < v2`', and (3) '`pkgname == v.*`'. With the following notation, we describe the ways in which a package can be defined:

```
GT(v)  ;  GETandLT(v1, v2)  ;  EQ(v, wc)
```

When we are parsing the version information from each cabal file, we attempt to identify the category of package definition that is being used in the cabal file. Once we can identify the type of package definition being used (one of `GT`, `GETandLT` or `EQ`), the next step is resolving the absolute version this definition corresponds to. However, in most cases the version definition in the cabal file corresponds to one of the `GT` or`GETandLT` types, while absolute version definitions of the type `EQ` are rarely to be seen. In fact, even in the case that there is the usage of the `EQ` type of version definition, we see that a wild card is used.

We describe two approaches to resolve the version of the used API and overcome the issues with the various complexities of the version definition mechanism that cabal provides.

With the first approach, one could reconstruct the version resolving process as it must have happened at the time of committing the cabal file. Therefore, the publication date of the release of the library and the dependency we are investigating serves as the commit date for the version definition. For example, if project `A-1.0` was released on date `D` and depends on library `L` with version definition `V` (e.g. `L >= 1 && < 2`), one would look up our available data and see which was the latest possible version of library `L` that was published at date `D`. However, with this approach we would see more version upgrades than those that actually performed, since it would not be clear whether the developer is the one who decided to perform an upgrade of the version being used, given the version definition is left unchanged.

With the second approach, given a version definition `D`, the absolute corresponding version will not be resolved. Instead, one relies completely on the version definition – in a slightly modified way. For the `EQ` version definition wild-cards will be neglected, for `GETandLT`, we only consider the upper bound of the definition and for `GT` we consider the lower bound of the definition. Thus, the resolver can be described as follows:

```
GT(v) → v  ;  GETandLT(v1, v2) → v2  ;
                EQ(v, wc) → v
```

By always selecting the boundary cases, we ensure that we get an accurate count of the version upgrade that is performed. As the boundary case changes to reflect that newer versions of the library are suitable for this project, we can increment the version that the project is using. Given that the second solution does not suffer from the same limitations as the first, we choose the second one to use in our approach to mine usage information of Haskell libraries.

III. Inferring reasons behind switching

Mileva *et al.* analyzed the trend of version usage of a library over time. They looked at whether the adoption of a new version of a library was impacted by a bug in that version and whether this lead to rollbacks of the version being used. With this paper, we propose a language independent approach to also analyze the rationale behind the client making a change to the version of the library being used.

We use Hackage as a source of clients of Haskell APIs.

To infer the reasons behind the change of a library version, our approach analyzes the commit messages. One caveat here is that a commit message on its own might not reveal the reason behind the change in a Cabal file, however, it is the best resource available to us in this study. In the future it would be prudent to investigate additional reasons behind the change in a Cabal file by analyzing resources such as issue tracker, API documentation and code comments. It starts with a pre-processing phase to (1) remove the non-alphabetical characters and stop-words and (2) stem the remaining words. Stemming simplifies how messages can be group together based on their content, *e.g.*, "fixed" and "fix" should be grouped together. Subsequently, our approach classifies the commits using keywords [8], [9]. We consider the keywords by Mauczka *et al.*, who created an algorithm that develops a weighted dictionary of keywords to classify commits based on their commit messages [10]. The dictionary that they created using their algorithm had a classification rate of 80.34% based on 8 open source projects, which all had at least $30,000$ commits [10]. We use the final dictionary created by the algorithm of Mauzcka *et al.* to classify commits into 3 categories [10], [11]:

Corrective. Commits that fix errors, failures and bugs concerning performance or the implementation.

Adaptive. Commits that add/change functionalities.

Perfective. Commits that increase performance, decrease redundancy and inefficiency, increase maintainability, or improve the layout and code style.

We apply this classification mechanism only to commits interesting to our purpose: Those that make a change to the build management system configuration. In the case of Hackage as a source of clients of Haskell APIs, interesting commits are those that modify the cabal files, *e.g.*, to add a library or to remove/change the version of a used library.

Based on the type of modification *i.e.*, whether a new dependency definition was added or whether one was removed,

we see as to what the change pattern is. For instance a new version definition could be added and the commit message could indicate that the modification is of a corrective type, thus combining these two facts our approach suggests that this new version of the library fixes some bug that was previously present probably due to an error in the previous version of the library. Table I shows a summary.

TABLE I: Reasons behind the change in dependency based on the type of source code change and the commit category.

Change	Category	Reason
Addition	Corrective	Compatible with the project
Addition	Adaptive	Dependency needed for new patch or feature
Addition	Perfective	Improvement of project
Deletion	Corrective	Removed for bugfix
Deletion	Adaptive	Compatibility issues
Deletion	Perfective	Not relevant / Unused

a) Addition and Corrective (ADD-COR). The addition of a library in a corrective commit suggests that the library fixes some failure or error in the project. When this is combined with the deletion of another library, it may indicate that this library may be better than other similar libraries.

b) Addition and Adaptive (ADD-ADP). The addition of a library in an adaptive commit suggests that the library is required for correctly implementing a certain patch or feature. Thus, indicating that the library or the library version added may be beneficial for the project and aid in its evolution.

c) Addition and Perfective (ADD-PER). A library being added in a perfective commit could be due to a several scenarios. For example, a library was added to increase performance or maintainability, to decrease inefficiency, or to perform stylistic changes. Since these possible scenarios differ so much, we conservatively say only that this combination is a improvement to the project.

d) Deletion and Corrective (DEL-COR). A library deleted in a commit responsible for fixing errors, faults, or bugs (Corrective) suggests that the usage of the library introduced unforeseen and undesired consequences into the project. Therefore, the library dependency was dropped to fix the bug(s).

e) Deletion and Adaptive (DEL-ADP). The deletion of a library in a commit that is responsible for adding or changing functionalities (adaptive) can be down to three potential scenarios: (1) the removed library did not provide the desired functionality, (2) the library was not compatible, (3) another library is better suited for the project. Overall, the library and project are not compatible with each other.

f) Deletion and Perfective (DEL-PER). As we consider perfective commits to be past the points of actual implementation, no major changes would be made in perfective commits involving changes in library dependencies. Only minor changes will be made like cleaning up or beautifying the source code. In these cases, the deletion of a library dependency will most likely be the removal of it due to it not being used or it could have been replaced by another.

Our approach conducts this classification for every library and its version as specified in the cabal file. After the whole process of data collection, we obtain for every library with its version the number of occurrences for each of the combinations specified above. A limitation to the validity of our approach is that we do not conduct a manual validation of the commits to establish the accuracy of our classification technique; we propose this validation as future work.

IV. Initial evaluation

Having implemented the approach, we conduct an initial exploration of the behavior of clients of Haskell libraries.

A. Popularity of versions over time

To find trends among library versions, we compared the results computed by our approach on the commonly used Haskell libraries. We distinguish between versions of an API that have been popularly adopted and those that are unpopular. As a preliminary threshold for this initial exploration, we define as popular a version that is used by at least 30%.

Our results show that certain versions are seldom used. It often happens that these versions are used by some projects for a long period of time and these projects never upgrade to a newer version. This situation may indicate that the projects are fully satisfied with the version.

From our data, we notice two trends in terms of popularity when a new version of a library is launched: (1) When a new version is launched, there are a lot of adopters, hence the version gains popularity so quickly that we can term it as a popular version; (2) the new version is barely adopted by projects, thereby becoming an unpopular version of the library.

The versions that we term as popular are adopted by many clients. Despite the release of newer versions of the library, these versions are never completely abandoned. For example, despite 2 years having elapsed since the release of a popular version and many newer versions being released, this popular version is continued to be used. This is similar to the trend observed in Java projects by Sawant and Bacchelli [12].

We observe that for all versions that become popular, the number of initial adopters is small. Then quite suddenly there is a large number of adopters of that version. These results indicate that developers are likely to follow the behavior of other developers, which is similar to the findings of Mileva *et al.* [2] and in line with Rogers theory of Diffusion of innovation [13].

For all the Haskell libraries under investigation we see similar results. The sole difference being the number of popular versions that we observe at the same time. For example, the library 'directory' often had two versions that were popular at the same time, whereas it was possible for the 'bytestring' to have three versions that were simultaneously popular. Similarly to the findings of Sawant and Bacchelli [12], libraries with several versions released over time have more than one popular version at a given time.

B. Reasons behind the switching of libraries

We look at the reasons behind the changing of a library for $1,250$ projects. We obtained $2,220$ unique library dependencies from the dependency files of the projects we target. We

wanted to see how each of these dependencies change over time and what the reasons are behind a change being made.

We observe that over 50% of the library dependency changes were categorized as either "**ADD-ADP**" or "**DEL-ADP**". This indicates that over 50% of the dependencies were changed (added or deleted) to either introduce a new feature or address the compatibility of a feature that might have been introduced in an earlier version. Furthermore, we see that only around 500 dependencies were removed due to the introduction of bugs or were indirectly associated with them. That number is small when compared to the total number of changed dependencies. This suggests that Haskell clients may not be adversely affected by API evolution on a large scale.

Furthermore, the developers have mostly either added or deleted a dependency due to corrective or adaptive measures while perfective measures were of less concern. This may suggest two scenarios: 1) when Haskell APIs evolve they do not make any major improvements to existing features (improvements such as efficiency of execution of a feature) or 2) clients of libraries do not care much about the minor improvements afforded by an API.

To further support our reasoning about library changes, we look at the relation between the categorization of the library change commits with the statistics from its switchbacks. For this analysis, we have selected one of the most downloaded and used library dependencies from Hackage, namely the `lens` package. We observed that `lens` had the most number of commits (515 commits) that modified its dependency file. Dependencies of `lens` have been upgraded 30% of the time to address or introduce a feature. It also indicates that a library has been upgraded over 60% of the time to maintain stability of the `lens` package.

In conclusion, the trend that we observed from our mined data was that majority of the libraries were upgraded to introduce new features and address the compatibility of other newly added dependencies. Most of the popular libraries showed equal number of addition and deletion of their library versions and simultaneously being categorized into corrective and adaptive metrics; this suggests that these libraries are quite stable and are upgraded to avoid bugs or incompatibility issues that may occur due to introduction of newer libraries. Interestingly, the libraries that we studied were not much affected by perfective measures like errors in documentation, refactoring or increasing efficiency and maintenance. This might be an indication that the developers in the Haskell community have already ensured good level of stability and efficiency. However, it could also mean that not much importance is given to these kind of measures as compared to other measures like preventing bugs and adding features. We believe that by mining and analyzing more commit messages, we could possibly state better inferences about the perfective measures. In this initial exploratory investigation, we did not investigate the trends we found further, but studies can be designed and carried out using and extending our approach to determine whether these trends are confirmed and establish their causes.

V. Conclusion

In this paper, we have presented a technique to identify the version of a library that a Haskell based project might be using, we also show how this data can be mined from Hackage on a large scale. Furthermore, we presented a language independent technique to infer the reasons that behind a project changing the library or version of the library it uses.

Based on our techniques, we analyzed the popularity of library versions, showing that a new version of a library is either popular or completely unused. We found that the popularity of a version grows all of a sudden. This is in line to what was seen in previous studies with the Java ecosystem, thus suggesting a behavior that transcends the programming paradigm and that brings more evidence to Rogers' theory on the diffusion of innovation. We observed that the majority of the libraries were upgraded to introduce new features and address the compatibility of other newly added dependencies. Only few changes made to dependencies were due to the fact that an upgrade would have been improved existing functionality. We also see that there are few corrective or adaptive changes, which suggests that the libraries may be all by and large stable.

References

[1] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN OOPSLA*. ACM, 2006, pp. 681–682.

[2] Y. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. ACM, 2009, pp. 57–62.

[3] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," *ECOOP 2009–Object-Oriented Programming*, pp. 318–343, 2009.

[4] A. A. Sawant and A. Bacchelli, "A dataset for api usage," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 506–509.

[5] J. G. Morris, "Experience report: Using hackage to inform language design," in *ACM Sigplan Notices*, vol. 45, no. 11. ACM, 2010, pp. 61–66.

[6] S. Raemaekers, A. v. Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 221–224.

[7] I. Jones, "The haskell cabal, a common architecture for building applications and libraries," 2005.

[8] B. Ray, V. Hellendoorn, Z. Tu, C. Nguyen, S. Godhane, A. Bacchelli, and P. Devanbu, "On the naturalness of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 428–439.

[9] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on*. IEEE, 2000, pp. 120–130.

[10] A. Mauczka, M. Huber, C. Schanes, W. Schramm, M. Bernhart, and T. Grechenig, "Tracing your maintenance work–a cross-project validation of an automated classification dictionary for commit messages," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 301–315.

[11] A. E. Hassan, "Automated classification of change messages in open source projects," in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 837–841.

[12] A. A. Sawant and A. Bacchelli, "fine-grape: fine-grained api usage extractor–an approach and dataset to investigate api usage," *Empirical Software Engineering*, pp. 1–24, 2016.

[13] E. M. Rogers, *Diffusion of Innovations*, 5th, Ed. Free Press, 2003.