Dependency behaviour of Haskell libraries
An extension to usage and switch-back tendencies in a different project environment

Marc Juchli, Lars Krombeen, Shashank Rao and Chak Shun Yu
Delft University of Technology

Abstract

Software developers face an enormous issue of upgrading or downgrading their library versions as the cost of migration is huge and can result in introduction of bugs into the project. In addition to the previously obtained Java usage trends, our research aims at studying the usage trends of Haskell libraries and determine if the popularity of a library is dependent on the programming language. We then extend our analysis to study the reason behind switching of different libraries by the developers. By analysing commit messages, we categorise upgrades and downgrades of library versions. Consequently, our research provides a technique to indicate which library version to use in a software project.

Keywords
datamining, library usage trends, version, API, switchbacks.

1. Introduction

Choosing which software library to use for your project is problematic for every programming team. Even when this decision has been made, it is still not trivial what version should be used [1], [2]. Newer versions of software libraries tend to introduce new functionality, which oftentimes is the main reason for using a library. It is also likely that an important security leak is fixed and therefore a new version will be released. But what if, after integrating the newer version into the project, an unforeseen bug occurs and the developers want to switch back. Thus, different choices for switching between newer and older versions of libraries can prove to be quite difficult.

Therefore, Mileva et al. investigated the usage trends of libraries among 250 APACHE Java projects that integrated MAVEN as their project management tool. They illustrate how these projects switched between library versions and constructed a technique that is based on the popular vote of the majority, the so called wisdom of the crowd. Further more, they provide numbers for the usage of versions of libraries and for the percentage of projects that switch between versions of libraries. For the developers of those libraries, this is valuable feedback as it gives insight about which versions of their work are highly appreciated and used, and which are less.

Given the trend of popularity for Java projects, it is uncertain whether the same results hold for projects written in other programming languages such as Haskell. In this paper we take upon this uncertainty and investigate whether the same result holds for different project environments. By looking at usage trends of other languages and comparing the results with the insight obtained by Mileva et al., we aim to make a more general statement about popularity.

In addition to the study of version usage trends, we also expand on the reasons why developers decide to switch from libraries. Mileva et al. did provide reasoning on why trends in library usages were observed, but they did not make this an actual part of their research to cover all of the projects. In this paper, we will try to cover these reasons more in depth by categorising switches between versions. Therefore we proceed keyword extraction and text analysis of the commit messages.

Combining the two gained insights, version popularity and the reasoning of version switches, our research shall assist users in understanding the reason behind popularity in certain versions of libraries. Thus, we want to support the users in finding the most suitable library version for their projects, instead of
blindly following the majority.

The structure of this paper is as follows: First we will describe the problem that we tackle in our research (Section 2). Then, we will provide background relevant to the problem that we defined (Section 3). This will consist of a information about the research of Mileva et al. and a short description about Hackage. Thereafter, we will describe in detail the approach we used to collect the data necessary for answering our research questions (Section 4). The approaches will be described separately for both research questions that we use to solve the problem defined. After formally formulating our research questions in Section 4, we will show the results of our research and provide analysis for their interpretation in sections 5, 6 and 7. Finally, we will evaluate our research and reflect upon parts that could be improved and limitations in our research in Section 8.

2. Problem Definition

The problem that we address in this paper is that it is problematic to choose proper library versions for your project and when to change the version of the library. When a new version of a used library is launched it can be considered to change to the newer version because the newer version fixes issues in the current one, or you could decide to stick with the current version because you do not require the update since everything is working fine. Another decision you could make is to wait a period of time to make sure there are no issues with the library based on other users. Looking at usage trends can give library users a recommendation which version to use or avoid which will speed up decision making and prevents possible issues with library that are used speeding up the development process. We also target to find out why libraries are abandoned or why newer versions are not used. This can help library developers by improving their application. For example, if a lot of users switch back from a certain version due to a bug, the library developers can quickly address the problem.

Consider that your company is building an application and wants to use a library for the image processing for an arbitrary reason. But which one should you use? Would you take the latest stable release or the latest release which could contain bugs but introduced a functionality that could come in handy? Additionally, consider that you just released V2.1.0 of your library, but it is not used a lot. Instead, people stick with V2.0.7 which is the previous release. What could be the reason behind it? Is there a bug that you are unaware of. Are the new features not as appealing as you originally thought. And what should you focus on for the next release to make sure that it will be used?

3. Background

This section will discuss work that has previously been done in the same research field with regards to library usage and version switches. In addition, Hackage and Cabal, two important tools to an arbitrary Haskell programmer, are mentioned.

3.1. Previous work

Mileva et al. is a previous study done in 2009 that was in the same research field [1]. They tried to provide a solution for choosing a specific library version based on usage trends. This was done by mining hundreds of Apache projects to determine global trends in library usage. With the information a developer could faster and easier determine which library version to use, speeding up the development process.

By looking at the pom.xml files within these projects they could gather information when a library version was changed or abandoned. Given the change and the date they could gather information about which library version was used and how many times it was used at a given time stamp. This could easily be converted to a usage trend.

3.2. Cabal

Cabal is a system used for the programming language Haskell to build and package libraries and programs[1]. The Cabal can be used in projects to make sure that the developers have the same libraries and dependencies installed on their system.

3.3. Hackage

Hackage is Haskells community’s central package archive[2]. It contains various information about projects

1. https://www.haskell.org/cabal/
2. https://hackage.haskell.org/
such as the published versions, the amount of downloads, the dependencies of other libraries for every published version etc.

Hackage is typically used for authors to publish their software as packages, ready to be used in other project as dependency. It is also used for other Haskell programmers as a central place for library documentation/usage.

4. Approach

The goal of the research was to find out if the research performed by Mileva et al. on library usage trends for Java-based software projects hold for Haskell. We also went a step further and looked why people choose to switch versions or abandon a library. The research questions we would like to solve using the approach described in this section are as follows:

- **Research Question 1**: How do usage trends of APIs in other programming languages compare to those in Java?
- **Research Question 2**: What are the reasons that software developers switch between different libraries and versions?

4.1. Popularity of libraries in arbitrary programming languages

Given this trend of popularity for Java projects described by Mileva et al., it is uncertain whether the same results hold for projects written in other languages. In order to investigate this, we will extract version changes from our gained results in form of library usage. The term usage in this case refers to the situation when a software library is used by another software project.

**Usage**: library is considered as a dependency in external software project.

Thus, each time the library is used as a dependency, the usage increases by one. In a second step, we confront the number of times a switch-back arises to the amount of switch-back’s given in a MAVEN based project. Hereby, a switch-back arises when a software project depends on an older version of a library, whereas in a previous release a newer version was already used.

**Switch-Back**: dependent version was decreased over time by an external software project.

In order to proceed this investigation, it is certainly beneficial to rely as closely as possible to the same approach of analysing library usage as Mileva et al. However, as we have chosen to confront the results with those of another programming language, namely Haskell, the process towards finding the final results might be different from the paper we defined as our suspect of investigation. Nevertheless, once our findings in usages and switch-backs are prepared, we are positive that those can be compared to the ones from from Mileva et al. Consequently, we can provide a statement in regards to whether it is or is not a wise choice of which library version should be used, based on usage trends.

The following part of this section will explain in detail the approaches we followed in order to collect the which was data required to answer the first research question.

4.1.1. Data Collection.

Selecting most used libraries

Hackage provides public insight into the top downloaded (in the past 30 days) Haskell projects. We found this a valid starting point for collecting library information that is significant to an arbitrary Haskell user. As we describe in Section 5, this hypothesis was confirmed with a snapshot containing all library downloads since Hackage was launched in 2007. As a result of this first step, a list of library names is available:

```
[Library]
```

Mine versions

Fortunately, the content of Hackage is stored in a Git repository³, which makes it handy to determine the available versions of the mentioned libraries in the previous step. Using Python as our language for the data mining tasks in the project, the second element of the function walk from the os library provides the names of elements within a folder. In this case, the elements are sub-folders named after the version numbers available for the particular library. As a results, we have now a list of libraries with all their released versions for the time being:

```
[Library[Version]]
```

3. https://github.com/commercialhaskell/all-cabal-files
Publication date

As the single source of a publication date for a library we could find was on Hackage itself. We were, at this point, forced to scrape the website of Hackage. Fortunately, every package, defining a specific library and version, holds its own API endpoint on the Hackage website:

/package/:package/

Therefore, an XPath (XML Path Language) query was used to identify the location of the publication date stated on the website and thus, read its value. After this step, a list of libraries with their versions and publication dates was available to us:

[Library[(Version, Published)]]

Resolve dependencies

This step is critical, as dependencies are the relevant suspects of our investigation. Since Hackage provides a distinctive website for the dependencies of a package, it was convenient to continue with the approach of using XPath. Thus, for every version above all libraries, we scraped the dependencies using the following API:

/package/:package/dependencies

Finally, the data set contained all the information for proceeding with the analysis part:

[Library[((Version, Published, [Dependency])]]

The important aspect at this point, in order to understand the data, is that a dependency is nothing else than yet again a library and version description. In this case, one could describe the available data as follows:

[Library[((Version, Published, ([Library, Version])]]

4.1.2. Determine Usage. Usage information is not yet processed, but as stated in the paragraph above, all the required data is given at this point. What is left to do, is to find the common dependencies above all libraries and their versions. It is convenient to represent this information in a dictionary with the library and version tuple as a key and the number of usages on a certain date as value, again stored as a dictionary:

{ (Library, Version): { Date: Integer } }

Using an iterative approach, we looped through every library, its versions and their dependencies (see structure in paragraph above) and incremented the integer value by one for each occurrence of a library version tuple. As we know the publication date of the library-version using the library-version of suspect as a dependency, it is possible to mark the date when the usage happened. Eventually, this dictionary represented the usage of all available library versions, including a very significant information, the date of that usage.

4.1.3. Determine Switch-backs. A switch-back was defined as the decrease of a version of a particular dependency. Decreasing a dependency version is one possible outcome of a developers decision in regards to dependency changes. The following are all the possible actions a developer of a library is in control of, in regards to the previous version/release of the project he is working on:

- New: A dependency, which was not used in the previous release, was added.
- Switch-back / Down grade: Resolved version of dependency is older than the version used in the previous release.
- Upgrade: Resolved version of the dependency is greater than the version used in the previous release.
- Stay: Resolved version is equal to the version being used in the previous release.
- Remove: A dependency which was used in previous release was removed.

The last three items of this list lead us to the conclusion that the implementation of a comparator might be suitable function to indicate which case is true, given two dependency versions. In order to define such a comparator, the following functionality must be given. First, one must be able to parse the version definition and second, one must be able to resolve the absolute version being used given the version definition.

Version definition parser

Parsing is very much bound to the specification of a version definition in cabal:

\[
pkgname >= n \\
pkgname >= n && < m \\
pkgname == n. *
\]

Thus, we defined three corresponding classes, each of which can take the version numbers as its arguments.

Tokenization of the version numbers is not required as Python can compare versions given as string correctly by default. Given a version definition, the parser would then return the appropriate class.

\[
\begin{align*}
&\text{GT}(n) \\
&\text{GETandLT}(n, m) \\
&\text{EQ}(n, wc)
\end{align*}
\]

A possible wild-card post-fix for EQ definitions are indicated with a Boolean argument.

**Resolve absolute version based on version definition**

After it is known what type of version definition we are observing (GT, GETandLT or EQ), the next step will lead to resolving the absolute version this definition corresponds to. This is not a trivial task, as a version definition in cabal may not relate to an absolute version number directly. In fact, most of the version definition found in projects uploaded to Hackage contain either version ranges (GT, GETandLT) or a wild-card within the EQ definition. At this point, one can follow two possible approaches.

Firstly, one could reconstruct the version resolving process as it must have happened back in time. Therefore, the publication date of the release of the library its particular dependency we are investigating serves as the due date for the version definition. For example, if library A-1.0 was release on date T and depends on library B with version definition D (e.g. B >= 1 && < 2), one would look up our available data from 4.1.1 and see which was the latest possible version published at date T. However, this approach has one significant drawback as it is not clear whether the developer did or did not decide on its own to proceed an upgrade, given the version definition was not changed. Thus, a version upgrade would be counted much more often than probably decided by a developers in the past.

The second approach, a much more naive one, works as follows. Given a version definition D, the absolute corresponding version wont be resolved at all. Instead, we rely completely on the version definition – in a slightly modified way. For the EQ version definition wild-cards will be neglected, and for GETandLT, we only consider the m of the definition. Thus, the resolver can be described as follows:

\[
\begin{align*}
&\text{GT}(n) \rightarrow n \\
&\text{GETandLT}(n, m) \rightarrow m \\
&\text{EQ}(n, wc) \rightarrow n
\end{align*}
\]

Our results are based on the second approach due to its simplicity and the mentioned drawbacks of the first approach. However, a more accurate approach would be a combination of both as mentioned in Section 8.

**Defining the comparator**

Last but not least we can define the comparator very easily give the functionality described above:

```python
def compare(a, b):
    av = parseVersionDefinition(a)
    bv = parseVersionDefinition(b)
    if resolveVersionNaive(av) > resolveVersionNaive(bv):
        return 1
    elif resolveVersionNaive(av) < resolveVersionNaive(bv):
        return -1
    else:
        return 0
```

First, we parse both version definitions a and b. Then we resolve the absolute versions in a naive way before we compare those with each other.

**Gaining insights into switch-back behaviour**

Given the version comparator as a tool which can be applied to the data we collected earlier (Section 4.1.1) will finally give insight into the upgrade- and downgrade behaviour of the Haskell libraries. Observing the behaviour of upgrades and respectively downgrades can be done in two different ways, each of providing a slightly different insight. The first approach, which is also described in Mileva et al., explains how often a library, used as a dependency, was downgraded by other libraries. In the paper mentioned, as well as in this paper, we call this behaviour switch-back. As a result, we can produce, for an arbitrary library we are interested in, a list consisting of the information of how often an action was proceeded for a given version of this library.

\[
\text{[(VersionDowngraded, VersionUpgraded, VersionStayed, VersionRemoved)]}
\]

The second approach, which we introduce in this paper as an extension to Mileva et al., concerns the proceeded upgrade- and downgrades by a certain library (or respectively its developers). For an arbitrary library we are interested in, we can conveniently summarise our findings into a quadruple containing the number of times certain actions were proceeded over the course of all available versions of this library.

\[
\text{[(DependencyDowngraded, DependencyUpgraded, DependencyStayed, DependencyRemoved)]}
\]
4.2. Reasoning about switching library versions

Where the research of Mileva et al. stops after gathering the information and combining them to discover usage trends in libraries and their versions, we are of the opinion that more relevant data can be extracted from this information if a little bit more of research was to be done. To be able to discover the specific reasons why software developers switch back to an older version of the library or decide to abandon a library in its whole for another library, we will dig deeper into the commits when the change in library happened. Information that we will try to obtain from these commits are the reasons behind the decision to change a library (version).

The following part of this section will explain in detail how we used existing research for the purpose of categorising commits into reasons. Based on the combination of these different categories, we will determine a specific reason behind the commit. Using these methods, we will obtain the data needed to answer the second question.

4.2.1. Data Collection.

Collecting data.

For the data we want to look at, we take the top downloaded and highly used libraries available at Hackage. In the end, libraries are just a project as well and it complements the data used in our first research question. For every of those libraries, we then consider all the commits that were stored in their version control system. Our data set consists of all the commits of in total 1250 Haskell libraries.

Determining interesting commits.

After collecting all the commits in the previous step, we go through all of them and filter them based on whether they are interesting for our research question. The commits that are interesting for us are the commits in which a dependency was changed. Whether it was adding a library, removing a library or changing a library version. They way in which we determine whether a commit has changed a dependency is by looking through the files changed in the version control system of the project and checking if the dependency manager file is included in these. In Hackage projects, this is the .cabal file at the root of the project. If no change was recorded in the .cabal file for a certain commit, we discard the commit and move on. In total, at this stage we are left with 659 interesting libraries and 37148 interesting commits.

Pre-processing commits messages.

Before we continue with our analysis of the messages involved in the commits, we have to pre-process the messages to obtain useful information out of them. First, we remove the non-characters using regular expressions. Then, we remove the stop words because they are frequent appearing words that are not interesting for us in any way. Lastly, we stem all the remaining words to be able to group them together. For example, "fixed" and "fix" should be matched together. To obtain this, we have to stem all the remaining words. The word "fixed" will then be reduced to its root "fix". These last two steps are performed using NTLK[^5]. All this together, is called normalisation.

Determining category of the commit.

After determining which commits are interesting for our research question, we classify them using a bag of keywords [3], [4]. The bag of keywords that we used is extracted from research done by Mauuczka et al. In their research, they created an algorithm that trains and develops a weighted dictionary of keywords on itself for the purpose of classifying commits based on their commit messages. 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 [5]. In this step, we use the final dictionary created by Mauuczka et al.'s algorithm to classify all the interesting commits into 3 categories: Corrective commits, Adaptive commits and Perfective commits [5], [6].

- **Corrective Commits.** Commits that contain changes for the purpose of fixing errors, failures and bugs in performance or implementation.
- **Adaptive Commits.** Commits that contain changes for the purpose of adding or changing functionalities.

[^5]: http://www.nltk.org/
• **Perfective Commits.** Commits that contain changes for the purpose of increasing performance, decreasing redundancy and inefficiency, increasing maintainability and changes to layout or style. We consider this past the point where features are already implemented and running.

### Classification in reasons.

After determining whether the commit we are currently considering is corrective, adaptive or perfective, we look into the changes made in the `.cabal` file. More specifically, we look into the addition and deletion of lines that concern the dependencies of the current project. We can do this because the `.cabal` file is responsible for dependency managing and every dependency is specified in exactly one line. Based on whether a library was involved in an addition or deletion and the category of the commit in which these changes happened, we classify them into another set of different categories. These categories are what we think is most logically implied by the combination of the factors we just mentioned. Here, we will elaborate more on our reasoning behind the combinations and their implication to our specified categories. In Table 1 there will be a summary of all these combinations and their derived categories.

<table>
<thead>
<tr>
<th>Change</th>
<th>Category</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>Corrective</td>
<td>Compatible with the project</td>
</tr>
<tr>
<td>Addition</td>
<td>Adaptive</td>
<td>Dependency needed for new patch or feature</td>
</tr>
<tr>
<td>Addition</td>
<td>Perfective</td>
<td>Improvement of project</td>
</tr>
<tr>
<td>Deletion</td>
<td>Corrective</td>
<td>Removed for bugfix</td>
</tr>
<tr>
<td>Deletion</td>
<td>Adaptive</td>
<td>Compatibility issues</td>
</tr>
<tr>
<td>Deletion</td>
<td>Perfective</td>
<td>Not relevant / Unused</td>
</tr>
</tbody>
</table>

Table 1. Reasons behind the change in dependency based on the type of source code change and the commit category.

**Addition and Corrective.** This combination is probably the best useful feedback a library developer can receive. The addition of a library in a corrective commit first of all means that the library fixes some failure or error in the project of the developer and thus is doing something good. Secondly, if this is combined with the deletion of another library, it most likely also means that for this particular developer and project the library is better compared to other similar libraries. In the end, the similarity in these reasons is the fact the library added is compatible with the project. This is in two ways: either another library introduces a bug or error which this library fixes, which means that this library is more suited for the project, or the library fixes some issue in the project, which means it is compatible with the project. *Compatible.*

**Addition and Adaptive.** This combination is probably the most straightforward combination, in our opinion. The addition of a library in an adaptive commit means that the library is required for correctly implementing a certain patch or feature. Thus meaning that the library added is beneficial for the project and aids in the implementation of a feature in the project in a suitable way. *Feature.*

**Addition and Perfective.** For the combination of a library being added in a perfective commit, there are a wide range of scenarios imaginable. Either...
a library was added to increase performance or maintainability, decrease inefficiency or stylistically changes. These possible scenarios differ so much from each other that they are hard to group together. The only thing we can guarantee, is that this combination is an improvement of the project. **Improvement.**

**Deletion and Corrective.** When a library is deleted in a commit responsible for fixing errors, faults and bugs (Corrective), it is very likely that the the usage of the library introduced unforeseen and undesired consequences into the project. Therefore, the library dependency has to be dropped in order to remove or fix the bug(s). **Bug.**

**Deletion and Adaptive.** The deletion of a library in a commit that is responsible for adding or changing functionalities (adaptive) normally has two possibilities. Either the library that is removed from the project is not what the developer wanted from the library, or it was not compatible with the project, or the developer found another library that is better suited for his project in functionalities and/or compatibility. All in all, the main similarity in all these reasons is that the library and project are not compatible with each other, in ways of actual compatibility and how suitable the library is for the project. **Non-Compatible.**

**Deletion and Perfective.** 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 beautify the source code. In these cases, the deletion of a library dependency will most likely be the removal of due to it not being used or wanted anymore because it was replaced by another. **Replaced.**

This classification will be done for every library and its version as specified in the `.cabal` file. After the whole process of data collection we will obtain for every library with its version the amount of occurrences for each of the combinations specified above. The format of the data obtained is as follow:

```plaintext
[ Library ,
  #ADD-COR, #ADD-ADP, #ADD-PER ,
  #DEL-COR, #DEL-ADP, #DEL-PER ]
```

Parts of this data will be showed in Section 7 and we will also try to derive trends based on this data and also evaluate onto these derivations there.

## 5. Library Usage Trends

To find trends amongst library versions we compared the mined results of common libraries. One of the results can be seen in figure 2 which shows a normalised stacked bar chart. The chosen libraries were commonly used libraries: dictionary, bytestring, base, lens, cabal, time and containers because most projects rely on these libraries, making them interesting to look at.

To find patterns we looked at the following:

- Trends of the dominating library version. A version is considered to be dominating if it holds more than 0.3 of the total shares.
- Total duration that a library was used frequently. A version is considered to be in use if it has a value of 0.2 or above. Everything below this threshold is not taken into consideration since they will not have a statistical meaning on major trends.
- Shape of the bars of a specific library version. So for example, do when see values like 0.2, 0.3, 0.4, 0.3, 0.2 it indicates that a version is slowly becoming popular. Another possibility is that we see values like 0.3, 0.3, 0.3, 0.2 which could be an indication that a version reached its highest popularity rapidly.

During the evaluation we will ignore small results because they are not of interest when finding major trends. These usages are most likely from projects that cling to a specific version for which everything works and do not consider switching. We can see similar trends between different libraries which we will discuss next. The differences will be also be briefly discussed.

When looking at the similarities we first see that when looking at dominating versions it can be observed that a version will quickly grow in use to be a dominating version once launched or be a version that is never used frequently. This can also be seen in figure 2 where four dominated versions are highlighted with white in the centre. It is also noticeable that the usage trends grown almost immediately after the first occurrence of the version after which they gradually...
Figure 2. Stacked bar chart of the contributions of the versions to the total usage of the Haskell library time. Bars that are white in the centre indicate some of the dominating versions which means that they contribute to more than 0.3 for some amount of time.

decrease in usage and make place for a next dominating version. Versions are taken in use quickly and discarded slowly but almost never discarded entirely.

When looking at the duration of used libraries we observe that the usage of a version is decreasing when a new version is released that will grow into a dominating version. When this is not the case the changes in version usage are minimal. So the introduction of a new version does only influence the other version usages if it will be successful. It is noticeable that this is the only influence on major changes in usages of different versions. The results show that developers are likely to follow the behaviour of other developers which was expected as Mileva et al. found similar results for projects in Java.

There are few differences between the results. The difference that is most present in the results are the amount of versions that are being used at the same time. For example, the library directory would most of the times have 2 dominating versions, whereas bytestring has 3 or even 4 dominating versions simultaneously. It is observed that libraries that have more available versions have more simultaneous dominating versions.

6. Switch-backs and Upgrades of Library Versions

This section will go in depth about the trends for library upgrades and switch-backs. Resulting from our approaches introduced in Section 4.1.3, we present switch-back behaviour from two different point of views. First, from the perspective of a library executing switch-backs on its dependencies. The second perspective puts the library in the perspective of a dependency being switched-back (downgraded) by other libraries.

6.1. Insights from a library perspective

The data collected in regards of the number of switch-backs and upgrades proceeded by a library is represented by popular libraries like lens and cabal in the following paragraphs.

We compared the results with the function

\[ \text{Rating} = \log(\frac{\#\text{upgrades} + 1}{\#\text{switchbacks} + 1}) \]

We add one in the fraction to prevent divisions by zero. Also, this results in a neutral line of \( \text{Rating} = 1 \) where 1/10 should be as far from the neutral line as
10/1. This is why the log is taken. The result is a line where negative values indicate more switch-backs then upgrades and the reverse for positive values. The results of lens and Cabal can be seen in figure 3.

The results could be used to find behaviour in switch-backs and upgrading. Firstly, a period where many downgraded occurred could give an indication that developers switched too early to newer versions. Secondly, if many stays are observed the version could be considered a sustainable.

The results show both similarities and dissimilarities. Firstly, we observe no similarities when people have a tendency to switch back versions. We see that switch-backs mostly occur after a period of time of many upgrades. However, some switchbacks happen after 1 year of the last upgrades and others after 1 month so this behaviour shows no obvious pattern when being compared.

Unfortunately, similarities are trivial. The amount of upgrades is always higher than the amount of switch-backs which should be the case. The other observation is that a switch-back will never occur before an upgrade which also is logical behaviour.

Concluding, the behaviour for switching back or upgrading the library version is strongly dependent on the library. If a new version is released which causes compatibility issues it is more likely that more switch-backs will occur. Therefore, it is difficult finding a pattern for upgrading and switching back versions without knowing the real reason behind it (e.g. the library contained a bug) which is a semantic problem. This was also researched and the results can be found in the following section.

6.2. Insights from a dependency perspective

The second perspective gives information about how often a library, being used as a dependency, was switched back or upgraded. The results clearly show that one can read a trend of versions being switch-backed over time. As an example, Table 2 contains the most interesting versions of the base library. Interesting meaning, that either switch-backs, upgrades or stays makes more than 10% of their corresponding total given all available versions. In total this means for the particular case of base:

| Total downgrades: 137.0 | Total upgrades: 162.0 | Total stays: 2225.0 | Total removes: 20.0 |

One can easily extract that, version resolving under the version definition (> 1&&< 1) are switched by more than 10% of all the libraries using base as a dependency, whereas for (== 4.) it is slightly more than 5%. Interesting to see is that people are starting to switch to base version 5, indicating by the slight switch-back rate of (>= 4&&< 4.8).

<table>
<thead>
<tr>
<th>Switch-backs (%)</th>
<th>Upgrades (%)</th>
<th>Stays (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>('base', ' (==4.*)')</td>
<td>5.56</td>
<td>0.00</td>
</tr>
<tr>
<td>('base', ' (&gt;1 &amp;&amp;&lt; 1)')</td>
<td>10.49</td>
<td>0.00</td>
</tr>
<tr>
<td>('base', ' (&gt;=4 &amp;&amp;&lt; 4.8)')</td>
<td>0.62</td>
<td>10.22</td>
</tr>
<tr>
<td>('base', ' (&gt;=4.3 &amp;&amp;&lt; 5)')</td>
<td>0.00</td>
<td>25.35</td>
</tr>
</tbody>
</table>

Table 2. Insight into the 10% most interesting versions of base library in regards of switch-backs, upgrades or stays.

7. Reasons behind Library Changes

In this section, we study the reason behind the library changes using the mined data as described in
Section 4.2. We obtained 2220 unique library dependencies from the dependency files of the top downloaded libraries of Hackage. Each of these dependencies were checked for the categories as explained in "Classification in reasons" of Section 4.2.1. From this data, we aim to obtain the trends and draw inferences over the possible reasons for change in libraries.

For studying the trends in the library changes, the relative statistics of the categories were of more relevance than their respective individual statistics. The statistics were divided into 4 bins of relative occurrence: category occurrences from 0 - 0.25, 0.25 - 0.50, 0.50 - 0.75 and 0.75 - 1.0. Figure 4 shows that almost over 1421 libraries had a maximum of 25% of "Compatible" feature. This indicates that not many libraries were changed due to compatibility issues. Additionally, figure 5 shows that over 50% of the library dependencies were categorised as either "ADD-ADP" or "DEL-ADP". As described in Section 4.2, this categorisation 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, figure 6 shows that only around 500 dependencies were removed as they introduced "Bugs" or were indirectly associated with them. That number is less when compared to the total number of dependencies that were changed.

If we refer to the figure 5, it is quite evident that the developers have mostly either added or deleted a dependency due to corrective or adaptive measures while perfective measures where of much less concern.

To further support our reasoning about library changes, we correlate the categorisation of the library with the stats from its switchbacks. We have selected one of the most downloaded and used library dependency from the Hackage, namely, *lens* package. We observed that *lens* had the most number of commits (515 commits) that modified it’s dependency file. Figure 7 gives some interesting yet consistent trends. *Lens* has been upgraded 30% of the time to address or introduce a Feature. It also indicates that the library has been upgraded over 60% of the time to maintain stability of the package. This trend can be validated with respect to the switchbacks and upgrades of its version as described in Section 7 and figure 3.

In conclusion, the trend that we observed from our mined data was that majority of the libraries were upgraded to introduces 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 categorised into corrective and adaptive metrics, thus indicating that these libraries are quite stable and are upgraded to avoid bugs or incompatibility issues creeping 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 analysing more commit messages in future, we could possibly state better inferences about the "perfective" measures.

8. Threats to Validity

We discuss some limitations that must be considered when interpreting the outcomes of the research. The following points are threats to validity:

Results might now hold for projects using different programming languages
The decision to compare the results of Mileva et al. with Haskell was because Haskell is a functional programming language in contrast to Java. Mileva et al. have claimed that "the libraries used in a project depend on the scope of the project and not on its management tools". We make the same claim and append it that by comparing it with a different type of programming language we have a more scattered sample set making the results less biased and thus more reliable.

Sample set may have biased the outcomes
By taking the top downloaded Haskell projects we took a sample set that is large enough to make conclusions about and simultaneously provide a sample set which is relevant because they are up-to-date and commonly used. Less popular projects do not contribute to the results. However, this filters out projects that are not relevant to the research, such as small personal projects. We believe that the less popular relevant projects are represented in the large sample set.

Version definition resolver might be too naive
As described in 4.1.3, the approach used to resolve the absolute version given a definition of a version is naive in the way that only the definition itself is considered. However, a more appropriate approach might be to actually reconstruct the resolving process as was in the past by considering the publication date of the release of the library that uses the dependency being our suspect. Then combine those two approaches in order to retrieve most accurate results.

Extend classification library change
For research question 2, we combined the addition or deletion of a library in the dependency manager file with the category of the commit based on the commit message. This resulted in six new categories that we used to identify different reasons for certain changes in library dependencies. However, in these six categories we only looked into the simple cases of either an addition or either a deletion. More information could be extracted based on the addition or deletion of another library in the same commit. In our analysis of the category combining an addition with a corrective commit, we already mentioned such a case. If a certain library X was added to the dependencies in a corrective commit whereas another library Y was deleted in the same commit, we have more information and we can make more specific conclusions. Instead of library X being suitable and compatible for the project, we can also derive that X is most likely more suitable and compatible to the project than library Y. This could provide better feedback for library developers to receive feedback on what their users want and what might be better in other competitors' libraries. Instead of six new categories, the classification could be extended to twelve cases and most likely provide more in-depth reasons.

Time of usage
Another factor that was not taken into consid-
eration in the analysis of research question 2, which could be interesting, is the time period over which libraries are used. Whereas the time that a library is kept into a project might not be that interesting, the time between adding a library and then removing it might be. This time-span could provide more information about the reason behind the deletion of the library. If the library was removed shortly after it was added, it was most likely due to not containing the features that the developer desired or was not compatible with the project. Whereas if the library was used for a longer period of time before being removed, it could tell something about the compatibility issues with other libraries, maintainability or scalability. The influence of time can be sizable, but has to be researched before any statement can be made concerning it.

9. Related Work

There has been quite some research in relation to API usage trends. Steven et al have performed similar studies previously. Their study primarily focuses on determining the stability of an API and eventually, helping the developer in deciding whether to switch altogether to another similar library or not. According to their studies, they based their results by computing four metrics: weighted number of removed methods, amount of change in existing methods, the ratio of change in new to old methods and the percentage of methods. Using these metrics they determine whether a change in method has caused a certain instability in the library. This study could help us in our analysis of switchbacks of library versions by the developers [7].

Cedric et al have studied about the various patterns of the library migration. They studied the dependency of one library on another and used the library usages as the weights for their metrics. Their result is quite promising as their study helps the developer over the decision of migration by showing them the graphs of migrating transition of the library. Although, this research was based on individual java based library belonging to limited categories, we aim to perform similar analysis on other languages as the methodology involves mining of the dependency files and import statements. Their study could also help us in improving our analysis over version switchbacks as it would require mining of individual library versions rather than the individual libraries themselves. Regarding the categorisation of software development based on the commit messages and the source code, Mockus and Votta [4] have performed similar analysis. Their categorisation methodology was based on keyword-detection. They categorised the commit messages then into 3 categories. Our method for determining reasons behind switching of libraries is also based and inspired by their methodology. To our best knowledge, there are quite less researches being performed on programming languages other than Java (Maven-based projects). Most of these studies show the developers to migrate to previous versions or other libraries but do not give the reason behind the switchback. We believe that our study could help in these aspects [8].

Zhong et al have built a tool which can recommend the use of an API based on the usages of other project. This indicates if the programmer is using the right sequence of API calls to prevent bugs [2] or give an indication to the programmer how to use the API to avoid searching how to use it properly.

10. Conclusion

This paper tried to find patterns in usage trends for library usage by looking at a various amount of projects written in the programming language Haskell.

This paper presented an approach on how to find patterns in library usage behaviour. Techniques were developed to find the general trends in the usage and to find patterns in switchbacks and upgrades of a library version. Another technique was developed to gain an understanding in the reasons behind the trends. Based on existing research and information provided by the version control system, we provided a technique to categorise commits into six different reasons for the change in library dependency.

The results showed that developers are likely to be attracted to similar library versions and that a new version is either successful or almost never used. The overall trend in the usage of such successful versions is a period where the usage rapidly increases after which the usage slowly declines. It was also noticed these successful versions are almost never discarded entirely. This proves that the results of Mileva et al. also hold for Haskell since developers are likely to follow others behaviour.

When looking at the qualitative results of switchbacks and upgrades we found that it is difficult to make statements about them since they are different for every library. No trends can be found when looking at
numbers only. There must be looked at the underlying reason behind the switch-back or upgrade to make valid statements.

From the results for the second research question, we observed that the 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 numbers of corrective and adaptive changes, which is a positive indication for the stability and usage of the libraries. Not a lot of libraries were affected by perfective measures. This could be due to factors like stability and efficiency of the libraries, but based on the data we have at hand this can not be concluded with certainty.

Acknowledgment

We would like to thank the Delft University of Technology as well as the Hackage Community for providing necessary resources. We also would like to especially thank Alberto Bacchelli and Anand Sawant for guiding us in the field of Mining Software Repositories and providing counsel and feedback.

References


