Declaration-level Change and Dependency Analysis of Hackage Packages

Extended Abstract

Philipp Schuster and Ralf Lämmel

Software Languages Team, Department of Computer Science, University of Koblenz-Landau, Germany

Abstract

Version numbers for Haskell packages on Hackage communicate when an update is possibly breaking and prevent installation of such updates. However, version numbers say nothing about which declarations actually changed. Similarly, version bounds on dependencies do not take into account which declarations a package actually uses. This leads to cases where installation of a package is unnecessarily prohibited. We describe a methodology and an supporting infrastructure, HackPackUp, for determining if and how an update affects a package. In a sample of 1,578 packages, we find 5,404 scenarios where an update is prohibited even though a package uses none of the changed declarations.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features; D.2.7 [*SOFTWARE ENGINEERING*]: Distribution, Maintenance, and Enhancement; F.3.2 [*LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages

Keywords Haskell. Hackage. Package Update. Version Bound. Program Analysis. Mining Software Repositories. HackPackUp.

1. Motivation

Imagine a hypothetical package *favorites* of version 0.2.0. Haskell or Hackage's *Package Versioning Policy* (PVP)[1] defines the major part of a version number to be the first two digits (in our example: 0.2) and the minor part to be the rest (in our example: 0). Version numbers are ordered lexicographically. It also specifies that all dependencies on *favorites* should be constrained with a lower and an upper bound. The lower bound excludes versions the package was not tested with. The upper bound includes all future minor versions but excludes all future major versions. So if for example some package was tested with *favorites* version 0.1.1 and 0.2.0 the version bounds should be $favorites >= 0.1.1 \&\& < 0.3$).

Further imagine, there is a function declaration *color* in a module in *favorites*. If a new version of *favorites* with an improved but backwards compatible implementation for *color* is released, its version number would only be increased in the minor part (in our example: 0.2.1). In this way, future installations of packages depending on *favorites* could use the improved version. If however, in a new version of *favorites*, the declaration for *color* would be removed, then the change would be backwards incompatible and the version number would have to be increased in the major part (in our example: 0.3.0). This means no existing package depending on *favorites* can be installed with the new version. They all have to be checked manually for compatibility and updated accordingly. If a package did not even use *color*, then the update would not affect the package, thereby prohibiting installation unnecessarily. We want to know if such cases exist in practice and how significant of a problem this is. To this end, we need information about the changes and the dependencies at the level of individual declarations for packages on Hackage.

The research reported in this extended abstract is an instance of 'mining software repositories'; see [4] for a survey. Such mining has also been researched in a Haskell/Hackage context with an objective different from ours; see the analysis of generic programming on Hackage [1]. The analysis of declaration-level changes and dependencies, as it is central to our research, also relates to change impact analysis, which is an established subject specifically for imperative languages; see [5] for a survey. For instance, a fine grained impact analysis which includes mining for an evolving software repository is reported in [3]. On the Haskell front, there exist tools to check what symbols a Hackage package update changes, e.g., *hackage-diff*,[2] but the analysis of changes at a declaration-level and their impact in terms of dependencies has not been the subject of research.

2. Research question

Our initial research question is this: *Do unnecessarily prohibited update scenarios exist on Hackage?* An update scenario is characterized by a package, a dependency of that package that satisfies the dependency constraints and any later version of that dependency. A prohibited update scenario is one where the later version of the dependency does not satisfy the constraints. An unnecessarily prohibited update scenario is one that could be permitted based on the criterion that the package is not 'affected' by revised or deleted declarations in the newer version of the dependency. Just like the PVP, we do not consider additions to be breaking because while added

[1] PVP: `http://www.haskell.org/haskellwiki/Package_versioning_policy` Explanation of the idea behind the PVP: `http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue2/EternalCompatibilityInTheory`

[2] `http://hackage.haskell.org/package/hackage-diff`

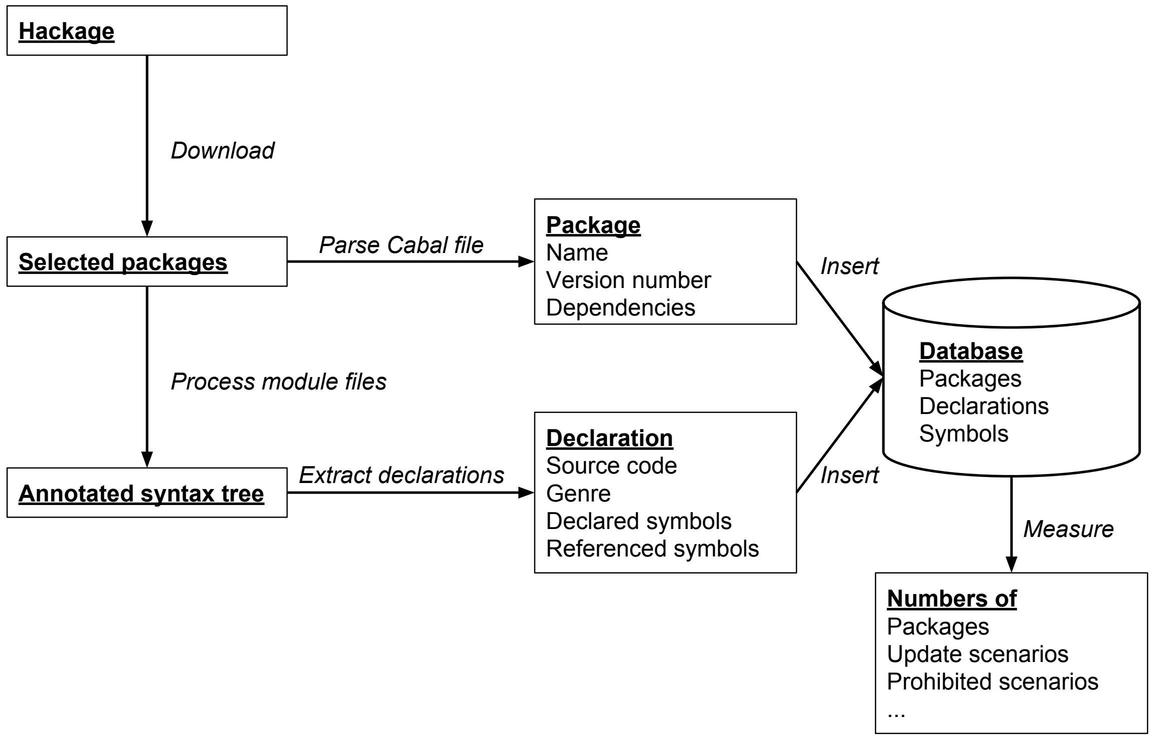

Figure 1. Overview of the fact extraction and measurement process.

declarations could cause name clashes these can be prevented with explicit import lists.

In fact, we would like to find out whether the problem of unnecessarily prohibited update scenarios is a significant one. Thus, we complement our research question as follows: *How many unnecessarily prohibited update scenarios exist on Hackage?* Accordingly, we perform (roughly) the following measurements by analyzing (a sample of) Hackage packages:

- How many update scenarios are there?
- How many of those are prohibited?
- How many of those prohibited are 'unnecessary'?

3. Fact extraction model

Fact extraction is applied to a set of packages (i.e., all of Hackage or a sample thereof). A *package* is uniquely identified by its *name* and its *version number*. A package p *depends* on package p' if the name of p' is listed as a dependency in the package description of p and the version number of p' satisfies the constraints in the package description. An *update* is a pair of packages with the same name and where the version number of the first package is smaller than the version number of the second package. The version numbers do not have to be consecutive, an *update* may skip several versions. A *major update* is where the major parts of the version numbers are different; otherwise we speak of a *minor update*.

A package declares a set of declarations, possibly subdivided into several modules. Every declaration has these properties: a *genre* (i.e., *function*, *type*, *class*, *instance*), sets of *declared* and *referenced symbols*, and the *source code* or *AST* underlying the declaration. Symbols are to be qualified by module names and genres (because of separate namespaces).

Using these basic facts, we can compute what symbols of a dependency a package requires and in what way an update changes a symbol. This allows us to decide if an update affects a package. We are inspired by other work on classifying software changes [2]. By collecting all such information and general package descriptions for Hackage packages (all of Hackage or a sample thereof), we can compute the measurements of §2.

4. Methodology

We address the research question through the following methodology. Figure 1 gives an overview.

- Download Hackage packages. We may need to trade scalability for completeness; see the actual selection of packages in the case study (§5).
- For each package's Cabal file, save these properties in the database:
  - The package's name and version number.
  - Which of all the packages under investigation it is allowed to depend on according to the dependency constraints. Usually, multiple different minor and major versions are allowed.

Packages	1,578
Declarations	332,663
Symbols	36,603
Update scenarios	212,176
Minor	113,030
Allowed	113,000
Prohibited	30
Major	99,146
Allowed	77,418
Affected	40,479
Unaffected	36,939
Prohibited	21,728
Affected	16,354
Unaffected	5,374

Figure 2. Measurement results of the case study.

  - An immediate next version, if it exists, together with the status whether it differs in the major version part. From the immediate next versions we can generate all updates. We are primarily interested in major updates, but the minor updates are interesting for validating the methodology.
- Attempt to install every package.
- Run the HackPackUp processor instead of the regular compiler:
  - Preprocess and parse to get the abstract syntax tree (AST).
  - Resolve names to annotate every symbol occurrence in the AST with its origin.
  - Save properties of each declaration in a database:
    - Source code
    - Genre
    - Declared symbols
    - Referenced symbols
- Run measurements (§2) against that database.

5. Case study

While the initial goal was to analyze all of Hackage, for scalability reasons, we had to choose some segment of Hackage. Without such selection, we would have to exercise too many dependencies and validate too many results. We looked for a list of packages that share many dependencies to minimize the number of packages we have to process. Stackage is a Haskell package repository with fewer packages than Hackage has. It is also complete in the sense that all dependencies of all its packages are included in it. We got the list of packages from Stackage and took all versions of all those packages from Hackage.

Table 2 lists number of entities resulting from fact extraction and storage in the database (§3 and §4).

We computed 212,176 update scenarios from our data. An update scenario consists of three parts: A package, a dependency of that package that satisfies the constraints and any later version of that dependency. In 113,030 of these update scenarios the update does not involve a major version change which means they should not be prohibited and indeed only 30 are. The other 99,146 update scenarios involve a major update. In 21,728 of these the version of the later dependency does not satisfy the constraints anymore. This means they are prohibited by an upper version bound.

We find that in 5,404 of the prohibited update scenarios the update does not affect the package and are therefore unnecessarily prohibited. We conclude this because none of the symbols the package requires from the first version of its dependency are absent or different in the second. If on the other hand we look at the 77,418 major update scenarios that are allowed we find that in 40,479 of those the update does indeed remove or alter at least one of the symbols required by the package. This could mean that the upper version bounds are wrong or missing. This could also mean that there are backwards compatible changes to parts of a package and backwards incompatible changes to another part requiring the update to be classified as *major*. We plan to investigate this problem in future work.

Besides the fact that we have only taken a sample of Hackage there are other threats to validity. Not all packages of the sample can be processed with HackPackUp. We use *haskell-src-exts*[3] for parsing and *haskell-names*[4] for name resolution. Most packages are only tested to work with GHC and make implicit assumptions about preprocessor flags, language extensions and builtin modules and therefore are not immediately parseable or some symbols fail to be resolved. We do not consider the *base* package as a dependency because it is a dependency of almost every package and for simplicity we only have one version and no information about the declarations of *base* available. We currently ignore type class instances because their resolution is out of scope of our tool.

At the time of writing, we are working on validating our classification of the update scenarios as unnecessarily prohibited. To this end, we need to install the packages with the varying dependencies. This proves difficult because of the number of installs and the time they take to build, especially in the view of a clean state needed for each test install.

In summary we can say that we indeed have found a significant number of cases where an update scenario is prohibited while the update would not affect the package. We have found, to our surprise, that in many update scenarios a major update does affect the package while it is still allowed. This observation calls for future work.

References

[1] N. Bezirgiannis, J. Jeuring, and S. Leather. Usage of generic programming on hackage: Experience report. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, pages 47–52. ACM, 2013.

[2] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, Sept. 2005.

[3] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 105–111. ACM, 2006.

[4] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, Mar. 2007.

[5] B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *Softw. Test., Verif. Reliab.*, 23(8):613–646, 2013.

[3] `http://hackage.haskell.org/package/haskell-src-exts`

[4] `http://hackage.haskell.org/package/haskell-names`