UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Dependencies between Haskell code fragments

Masterarbeit

zur Erlangung des Grades eines Master of Science
vorgelegt von

Philipp Schuster

Erstgutachter:	Prof. Dr. R. Lämmel Institut für Softwaretechnik
Zweitgutachter:	M. Sc. M. Leinberger Institut für Web Science und Technologien

Koblenz, im Januar 2015

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	□	□
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	□	□

....

(Ort, Datum) (Unterschrift)

Abstract

Code package managers like Cabal track dependencies between packages. But packages rarely use the functionality that their dependencies provide. This leads to unnecessary compilation of unused parts and to speculative conflicts between package versions where there are not conflicts. In two case studies we show how relevant these two problems are. We then describe how we could avoid them by tracking dependencies not between packages but between individual code fragments.

Zusammenfassung

Paketmanager für Quellcode wie zum Beispiel Cabal verwalten unter anderem die Abhängigkeiten zwischen Paketen. Softwareprojekte nutzen jedoch selten sämtliche Funktionalität, die ihre Abhängigkeiten bereitstellen. Das führt zur unnötigen Kompilation unbenutzer Code-Fragmente und zu vermeintlichen Versionskonflikten, wo gar keine Konflikte sind. In zwei Fallstudien zeigen wir wie relevant diese zwei Probleme sind. Danach beschreiben wir wie wir sie vermeiden können, indem wir Abhängigkeiten nicht zwischen ganzen Paketen, sondern zwischen einzelnen Code-Fragmenten feststellen.

Contents

List of Figures

Chapter 1

Introduction

Software development very often relies on the use of software libraries. Software libraries provide reusable functionality. Software maintainers bundle and distribute their libraries as packages. Every package corresponds to a version of the library. Libraries themselves very often reuse functionality from other libraries. These other libraries are their dependencies. During compilation all dependencies of a package have to be present. Package maintainers therefore explicitly state in every package its dependencies on other packages.

Packages very often only use parts of all the functionality another package provides. Specifying dependencies on entire packages instead of exactly on the used parts has at least two drawbacks that we explore in this thesis. First, installation of packages takes unnecessarily long because unused code has to be compiled all the same. Second, changes that are strictly in the unused parts of a package still lead to version conflicts.

If we knew exactly which parts of a package another package uses we could solve both problems: We could compile only those parts of a package that a package actually uses and we could find out if updating a package really affects another package. As far as we know nobody has examined if this is worthwhile and what implications this has on how distribution of code works.

Our research questions therefore are:

- How many code fragments do packages actually use?
- Can we automatically detect if an update affects a package?
- How could code distribution in units of code fragments work?

In this thesis we focus on Haskell and the related code distribution infrastructure. Other package-based code distribution infrastructures in principle suffer from the same problems. See section 2.1 for a comparison.

More concretely this thesis makes the following contributions:

- A formal model of Haskell code and packaging.
- An infrastructure to extract facts from existing Haskell packages according to that model.
- A case study finding how many declarations Haskell packages actually use.
- A case study finding unnecessarily prohibited update scenarios in a set of Haskell packages.
- A prototype implementation of a tool that compiles only those declarations a Haskell program actually uses.

The code for the tool that extracts the facts [1] and a script that glues everything together [2] as well as the code that imports the data into the database [3] and one that runs the queries [4] can be found online. Ongoing development of our tool that compiles only those declarations a Haskell program actually uses is under [5].

The rest of this thesis is organized as follows: We classify and review related work in chapter 2. We motivate and define key terms in chapter 3. We describe the methodology, implementation and results of the two case studies in chapter 4. We show the tool that avoids compilation of unused declarations in chapter 5. Finally we summarize and discuss our main contributions and results again, but this time from a more internal standpoint in chapter 6.

Chapter 2

Related work

We conduct empirical research on a Haskell package repository that includes change impact analysis. We review relevant work others have done in these areas and describe how it differs from our work.

2.1 Code package managers

This thesis is in the general context of code package managers and repositories. We focus on the package infrastructure for Haskell where cabal-install is the package manager and Hackage is the Package repository. Examples of very similar infrastructures include but are not limited to: CPAN, Pip/PyPi, Cargo, Npm, OPAM, maven, Clojar and Composer. Most of them encourage a versioning policy like the Package Versioning Policy called Semantic Versioning [6]. Some of them allow a single package to use multiple different versions of another package at the same time. All of them do dependency resolution and have to deal with version conflicts. In [ACTZ12] the authors propose to reuse a common dependency resolution implementation. Package conflicts during upgrades are the subject of [ADC11] where the authors predict upgrade failures in package management, but on the coarser package level and not code-based like we do. An analysis of the different sources of package conflicts is in [ASDC+12].

2.2 Empirical software engineering

In our case studies we mine software repositories which is a method from empirical software engineering. See [KCM07] for a survey. An infrastructure for mining open source software repositories is presented in [BOL14]. While in our case study we mine Hackage for fine-grained changes and dependencies, [RDV13] presents a dataset with roughly the same information except for Java packages in the Maven software repository. Another example of mining the Maven repository is [MKL$^+$14] where the authors run a tool that statically analysis Java bytecode for bugs on packages. In [MDBZ09] the authors mine software repositories to find the popularity of a version of a library to guide developer choice. Others have mined Hackage before, for example [BJL13], but with the different goal of surveying the use of generic programming techniques.

2.3 Change impact analysis

We do change impact analysis by software slicing. See [LSLZ13] or [Tip94] for surveys. Another example of fine grained impact analysis which includes mining of an evolving software repository is reported in [CC06]. In [Boh02] the author extends change impact analysis to third-party dependencies, but not exactly packages from a code package manager like we do. An example where software changes are propagated to exactly those affected components is [Raj00]. The authors predict the impact of change requests in [CC05], but in contrast to us not based on code but based upon the request itself. In [KM07] static program slicing and change prediction based on version history are combined. There are also examples of slicing Haskell programs [RB06] but with the different goal of finding coherent units of code and of slicing Erlang programs to find fine-grained dependency graphs [MT11].

2.4 Haskell, Hackage, Cabal

Other orthogonal approaches to the solution of dependency problems with Haskell packages are [KDPJM14], the recent [7] project, bumper [8] and similar tools as well as Cabal features like sandboxes [9], private dependencies and the ability to have multiple package instances installed at the same time[10]. A meta-model in UML for Haskell is given in [Szl11]. Inspiration for the Package Versioning Policy is from [11]. A tool to compare the public API of different versions of Haskell packages is hackage-diff [12]. SourceGraph is a tool that can build and visualize among other things a call graph for a given Haskell package. The non-scientific community already proposes but does not implement the idea of tracking more fine-grained dependencies in [13] and the automatic detection of compatible updates in a very similar way to ours in [14].

Chapter 3

A model of Haskell packaging

A Haskell program is a set of modules [15]. Not all the modules have to be written by the same developer. Very often developers reuse sets of modules written by other developers. The purpose of the code package manager Cabal and the central package repository Hackage is to make distribution of reusable sets of Haskell modules easier. They distribute and manage Haskell modules in units of packages. We motivate and define two fundamental problems with package-based code distribution.

3.1 An example dependency situation

Consider the two packages in figures 3.2 and 3.1. Package `application-0.1.0` is an executable that prints two strings to the console. The first string represents a color and the second one represents a number. Package `favorites-0.1.0` contains declarations for the two strings to print. Ignoring all the details, the important point here is that we have two packages and one uses the functionality the other one provides.

For the sake of example the implementation for `color` in figure 3.2 does an unnecessary string concatenation. The package author improves the implementation of `color` and releases `favorites-0.1.1` shown in figure 3.3. It is safe and beneficial to replace every use of the declaration for `color` from `favorites-0.1.0` with the declaration from `favorites-0.1.1` because the latter exhibits the same behavior but might be more efficient. In cases like this the

```
-- application.cabal
name:         application
version:      0.1.0

executable application
  main-is:        Main.hs
  build-depends: base >=4.7.0.0 && <4.8,
                 favorites >=0.1.0 && <0.2
```

```haskell
-- Main.hs
module Main where

import Favorites (color,number)

main :: IO ()
main = do
    putStrLn color
    putStrLn number
```

Figure 3.1: Package `application-0.1.0`

```
-- favorites.cabal
name:         favorites
version:      0.1.0

library
  exposed-modules: Favorites
  build-depends:   base >=4.7.0.0 && <4.8
```

```haskell
-- Favorites.hs
module Favorites where

color :: String
color = "light" ++ "blue"

number :: String
number = "four"
```

Figure 3.2: Package `favorites-0.1.0`

```
-- favorites.cabal
name:         favorites
version:      0.1.1

library
  exposed-modules: Favorites
  build-depends:   base >=4.7.0.0 && <4.8
```

```haskell
-- Favorites.hs
module Favorites where

color :: String
color = "lightblue"

number :: String
number = "four"
```

Figure 3.3: Package `favorites-0.1.1`

replacement should happen automatically, so that users benefit from the advantages of the new implementation without effort.

Now the author decides to further improve the `favorites` library. In package `favorites-0.1.1` she changes `number` to have type `Integer` instead of `String`. She then releases the new version as `favorites-0.2.0`. But most code that uses the declaration for `number` from either `favorites-0.1.0` or `favorites-0.1.1` can not use the declaration from `favorites-0.2.0`. In most cases this would result in a type error at compile time. This time no automatic replacement should happen. Instead developers have to decide on a case to case basis if they use the new package, because very often they would have to adjust their code to work with the new version of the library.

This concludes the example of the life cycle of a Haskell package. We observe that there are two kinds of updates: compatible and breaking. Package users want to benefit from compatible updates while they do not want their code to break because of a breaking update. The Haskell packaging infrastructure makes this possible.

```
-- favorites.cabal
name:         favorites
version:      0.2.0

library
  exposed-modules: Favorites
  build-depends:   base >=4.7.0.0 && <4.8
```

```haskell
-- Favorites.hs
module Favorites where

color :: String
color = "lightblue"

number :: Integer
number = 4
```

Figure 3.4: Package `favorites-0.2.0`

3.2 Modules, packages, dependencies

Every Haskell package comes with a Cabal [16] file ending in `.cabal`. It contains information about the package such as its name and its version number. In the example we have four packages. The one in figure 3.1 has name `application` and version number `0.1.0`. The three in figures 3.2, 3.3 and 3.4 all have name `favorites`. Their version numbers are `0.1.0`, `0.1.1` and `0.2.0` respectively.

Package `application-0.1.0` is a package with a section that describes an executable called `application`. It depends on two other packages `base` and `favorites`. Every executable Haskell program has to have a module called `Main`. In this case it is in file `Main.hs`. Package `favorites-0.1.0` in figure 3.2 is a package with a section that describes a library. The library exposes a module `Favorites`. It depends on the `base` package. If a library exposes a module there has to be a corresponding Haskell source file in a location determined by convention. In this case the `Favorites` module is in file `Favorites.hs` also shown in figure 3.2.

A Haskell module consists roughly of a name, a list of imports and a list of declarations. Every declaration defines a (possibly empty) list of symbols. It also mentions symbols bound by other declarations. The mentioned symbols can be bound in the same module or imported from other modules as controlled by the list of imports. In other literature [15, 17] what we call a symbol is called an entity.

The file `Main.hs` shown in figure 3.1 is a Haskell module. The first line specifies the module's name to be `Main`. It imports symbols `color` and `number` from module `Favorites`. It also contains two declarations. The first one is a type signature. It does not bind any symbols but mentions the symbols `main` and `IO`. Every Haskell module implicitly imports many commonly used symbols, among them the `IO` symbol. The second declaration binds the `main` symbol. Furthermore this declaration mentions the implicitly imported `putStrLn` symbol. It also mentions the symbols `color` and `name`. The module explicitly imports those two symbols from module `Favorites`.

The `Favorites` module in file `Favorites.hs` is shown in figure 3.2. It does not contain any imports but does contain four declarations. Two of them are type signatures that all mention the implicitly imported `String` type. Each of them also mentions another symbol that the other declarations in this same module bind. Those two symbols are `color` and `number` respectively. The declaration that binds `number` does not mention any symbols. The declaration that binds `color` mentions the implicitly imported string concatenation operator `(++)`.

Module `Favorites` in figure 3.3 is like `Favorites` in figure 3.2 except for the declaration that binds `color`. Instead of using the string concatenation operator `(++)` it consists of a literal of the already concatenated string. In module `Favorites` in figure 3.4 the declaration that binds `number` as well as its signature are different. The type of `number` is `Integer` when previously it was `String`. The declaration that binds `number` is a number literal instead of a string literal.

All implicitly imported symbols are from module `Prelude` from package `base-4.7.0.0`. All example packages list the `base` package as a dependency. This package is usually tightly integrated into any Haskell compiler. Therefore Haskell compilers usually come with this package already installed.

Resolution of imports and exports of Haskell modules is rather involved. For a formalization of the Haskell module system see [DJH02]. We can uniquely identify a symbol by its name, the name of module where it is bound and the namespace it is in. We have to distinguish value and type namespaces because it is possible that a type-level and a value-level symbol share exactly the same name.

For example it is common to bind a data type as well as a constructor with exactly the same name in the same module.

3.3 The Package Versioning Policy

We can successfully compile package `application-0.1.0` together with package `favorites-0.1.1`. We can not compile package `application-0.1.0` together with package `favorites-0.2.0`. Moreover we argue that package `favorites-0.1.1` is preferable to package `favorites-0.1.0`. The Package Versioning Policy (henceforth PVP) ensures that a fresh installation of package `application-0.1.0` compiles it together with package `favorites-0.1.1` and not with package `favorties-0.1.0` (because it is outdated) nor with package `favorties-0.2.0` (because it is incompatible). The PVP suggests how developers choose version numbers of new packages and how they constrain dependencies on other packages. All packages in section 3.1 follow the PVP. One goal of the PVP is that if a package compiled at one point in time it should be possible to compile it at any later point [11]. The other goal is to have users get all compatible updates on every fresh installation of a package.

Cabal packages have version numbers. A version number is a list of numbers separated by dots. The first two numbers are called the major part of the version number, the rest is called the minor part of the version number. In the example the major part of both version numbers is `0.1` while the minor part of both is the single digit `0`. If the version number would be `4.6.0.3` the major part would be `4.6` and the minor part would be `0.3`. The PVP says that if a library author releases a new version and it introduces breaking changes the new version is required to have an increased major version part as compared to the previous version. If the new version cannot possibly break any package that depends on it the PVP only requires an increase in the minor version part of the version number. The version number therefore reflects backwards compatibility.

The PVP furthermore requires that package authors constrain the versions of every dependency by a lower and an upper bound. In figure 3.1 the package `application-0.1.0` has a dependency on `favorites`. This dependency has a lower bound of and including `0.1.0` and an upper bound of and excluding `0.2`. The lower bound must be accurate, which means it has to include the earliest package that is known to work and not more. The upper bound must exclude packages whose version number has a major part that is larger than the latest version known to work.

A package does not necessarily expose exactly the symbols that are bound in it. It can expose more symbols by reexporting and it can exposer fewer symbols by hiding entire modules or individual symbols. The PVP is only concerned with exposed symbols. It defines breaking changes to be:

- Removal of an exposed symbol
- Change of an exposed value symbol's type
- Change of an exposed type symbol's definition
- Addition of an orphan type class instance
- Removal of a type class instance

It is at the package authors discretion to decide if semantic changes are breaking or not. Note that while the addition of orphan instances can break code because of clashes, instance clashes are impossible for non-orphan instances. This list does not cover all possible sources of breakage for example through name clashes. Nevertheless it works pretty well in practice.

3.4 Dependency resolution

In the example in section 3.1 we can compile package `application-0.1.0` together with either package `favorites-0.1.0` or with `favorites-0.1.1`. This means there are two possible installations of package `application-0.1.0`. An installation for a package is a list of other packages. In Cabal an installation has to meet two criteria: It must satisfy all dependency constraints and every package name must be unique.

The process of finding an installation for a given package is called dependency resolution. It is non-trivial [ACTZ12, TLO10], because the number of candidate installations quickly grows with the number of dependencies. In the Haskell packaging infrastructure the tool cabal-install does dependency resolution. On the one hand it is very frustrating for users if cabal-install does not find an installation. Even more so if a working installation exists but version constraints prohibit it. On the other hand if we ignore version constraints it is equally frustrating if cabal-install tries a failing installation even though a working one exists.

In Haskell folklore the term for what we call an installation is the term package instance or just the term instance. But the term instance is easily confused with the term type class instance. We therefore prefer the term installation.

3.5 The data model

We formalize the concepts that we motivate in the previous sections and show how they relate. The language of choice for this formalization is Prolog because we describe primitive entities as well as their relations. We also define new relations based on the primitive ones. An entity relationship diagram of the data model is given in figure 3.5. Boxes correspond to entities. Their basic properties are listed in the boxes. Arrows correspond to relationships between entities.

First, we have packages. We uniquely identify packages by their name and version number. Packages have a next version relation. If there is a package with the same name and a higher version number the next version is the package with the lowest version number that is still higher. We also have a relation between packages that satisfy the dependency constraints of other packages. Finally packages can have a relation to an installation. An installation is a pair of the package and a list of other packages.

Every package contains a set of declarations. The only property of a declaration is its source code. Every declaration relates to the symbols it binds. It also relates to the symbols it mentions. We uniquely identify symbols by their name, the module where they are originally bound in and their namespace (to disambiguate constructors, types and type classes). This allows us to relate declarations that bind a symbol with declarations that mention it. In contrast to packages and symbols we do not uniquely identify declarations by their properties i.e. their source code.

In figure 3.6 we show relations for the packages that we introduce in section 3.1. There are four packages in the example in figures 3.1, 3.2, 3.3 and 3.4. We ignore package `base-4.7.0.0` for readability. A version of `base` comes with every Haskell compiler and almost all Haskell packages depend on it. The next version of package `favorites-0.1.0` is package `favorites-0.1.1` and the next version of package `favorites-0.1.1` is package `favorites-0.2.0`.

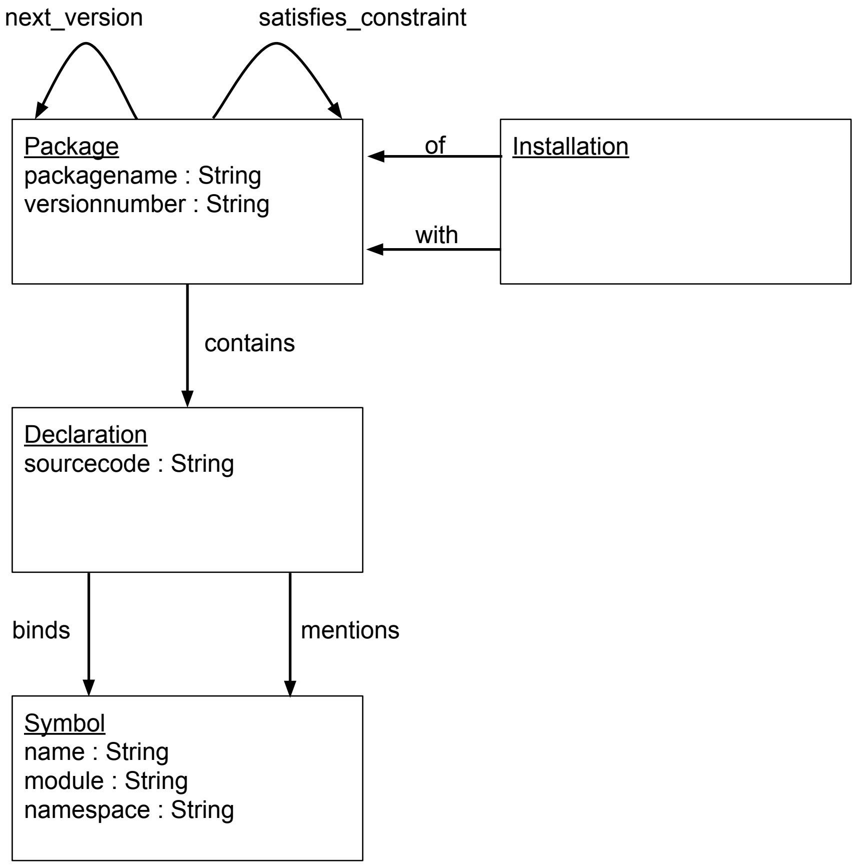

Figure 3.5: Data model for Haskell packaging.

```
package(package(’application’, ’0.1.0’)).
package(package(’favorites’, ’0.1.0’)).
package(package(’favorites’, ’0.1.1’)).
package(package(’favorites’, ’0.2.0’)).

next_version(package(’favorites’, ’0.1.0’),
             package(’favorites’, ’0.1.1’)).
next_version(package(’favorites’, ’0.1.1’),
             package(’favorites’, ’0.2.0’)).

satisfies_constraint(package(’application’, ’0.1.0’),
                     package(’favorites’, ’0.1.0’)).
satisfies_constraint(package(’application’, ’0.1.0’),
                     package(’favorites’, ’0.1.1’)).

installation(installation(package(’application’, ’0.1.0’),
                          [package(’favorites’, ’0.1.0’)])).
installation(installation(package(’application’, ’0.1.0’),
                          [package(’favorites’, ’0.1.1’)])).
installation(installation(package(’application’, ’0.1.0’),
                          [package(’favorites’, ’0.2.0’)])).
```

Figure 3.6: Relations for example packages.

The two packages `favorites-0.1.0` and `favorites-0.1.1` both satisfy the version constraint on the `favorites` dependency of `application-0.1.0` so there are relations to reflect that fact. Package `favorites-0.2.0` does not satisfy the version constraint so there is no `satisfies_constraint` relation for it. This example also lists three different installations of `application-0.1.0`: one with package `favorites-0.1.0`, one with package `favorites-0.1.1`, and one with package `favorites-0.2.0`.

There are many declarations in the examples in section 3.1. In figure 3.7 we show relations for the declaration contained in package `application-0.1.0` that binds the `main` symbol. It mentions three symbols: `puStrLn` originally bound in module `System.IO` as well as `color` and `number` originally bound in module `Favorites`. Note that while the symbol `putStrLn` is imported from `Prelude` it is originally bound in `System.IO`. Module `Prelude` reexports it.

This is exactly the kind of data that we need for the case studies in chapter 4.

```
contains(package(’application’,’0.1.0’), D).

sourcecode(D, ’main = do
    putStrLn color
    putStrLn number’).

binds(D,symbol(’main’, ’Main’, ’value’)).
mentions(D,symbol(’putStrLn’, ’System.IO’, ’value’)).
mentions(D,symbol(’color’, ’Favorites’, ’value’)).
mentions(D,symbol(’number’, ’Favorites’, ’value’)).
```

Figure 3.7: Relations for an example declaration.

3.6 Installation of unused declarations

Packages rarely use all declarations their dependencies contain. The example package `some-printer-0.1.0` in figure 3.8 contains a section for an executable that simply prints a string. Package `many-things-0.1.0` in figure 3.9 contains among others a declaration for this string. The `main` declaration in package `some-printer-0.1.0` uses the declaration for `thing1` contained in package `many-things-0.1.0`. We can relate the two declarations by use because the former mentions the symbol `thing1` that the latter binds.

If we want to compile `some-printer-0.1.0` we have to compile its dependencies first. In other words, we have to find and compile an installation for `some-printer-0.1.0`. In this example the installation is obvious: we compile package `some-printer-0.1.0` together with package `many-things-0.1.0`. But in this example we could compile `main` without compiling the declarations for `thing2` and `thing3`. Package `some-printer-0.1.0` does not actually use them.

In figure 3.10 we use the data model from section 3.5 to define what it means for a declaration to either directly or transitively use another declaration. A declaration directly uses another declaration if it mentions a symbol the other declaration binds. It transitively uses a declaration either if it uses the other declaration directly or if it directly uses a declaration that in turn uses the other declaration transitively.

```
-- some-printer.cabal
name:        some-printer
version:     0.1.0

executable output-app
  main-is:       Main.hs
  build-depends: base >=4.7.0.0 && <4.8,
                 many-things >=0.1.0 && <0.2
```

```haskell
-- Main.hs
module Main where

import ManyThings (thing1)

main :: IO ()
main = putStrLn thing1
```

Figure 3.8: Package `some-printer-0.1.0`

```
-- many-things.cabal
name:        many-things
version:     0.1.0

library
  exposed-modules: ManyThings
  build-depends:   base >=4.7.0.0 && <4.8
```

```haskell
-- ManyThings.hs
module ManyThings where

thing1 :: String
thing1 = "thing1"

thing2 :: String
thing2 = "thing2"

thing3 :: String
thing3 = "thing3"
```

Figure 3.9: Package `many-things-0.1.0`

```
uses_directly(Using_Declaration, Used_Declaration) :-
  mentions(Using_Declaration, Symbol),
  binds(Used_Declaration, Symbol).
```

```
uses_transitively(Using_Declaration, Used_Declaration) :-
  uses_directly(Using_Declaration, Used_Declaration).
uses_transitively(Using_Declaration, Used_Declaration) :-
  uses_directly(Using_Declaration, Another_Declaration),
  uses_transitively(Another_Declaration, Used_Declaration).
```

```
installs_declaration(Installed_Dependencies, Declaration) :-
  member(Installed_Dependency, Installed_Dependencies),
  contains(Installed_Dependency, Declaration).
```

```
actually_uses(installation(Package, Installed_Dependencies),
              Used_Declaration) :-
  installs_declaration(Installed_Dependencies,
                       Used_Declaration),
  contains(Package,
           Declaration),
  uses_transitively(Declaration,
                    Used_Declaration).
```

Figure 3.10: Definition of installed and actually used declarations.

We also define which declarations an installation installs versus which declarations a package actually uses. The installs_declaration relation relates an installation with every declaration in one of the installed dependencies. The actually_uses relation relates an installation of a package with the subset of those installed declarations that at least one contained declaration transitively uses. In both cases we do not count declarations in the package itself but only those from other packages.

In real world packages the problem of compiling declarations that are not actually used is even more pronounced. We provide evidence for that in section 4.2. We demonstrate a tool that avoids compilation of declarations that are not actually used in chapter 5.

```
package(package(’some-printer’, ’0.1.0’)).
package(package(’many-things’, ’0.1.0’)).

satisfies_constraint(package(’some-printer’, ’0.1.0’),
                     package(’many-things’, ’0.1.0’)).

installation(
    installation(package(’some-printer’, ’0.1.0’),
                [package(’many-things’, ’0.1.0’)])).

actually_uses(
    installation(package(’some-printer’, ’0.1.0’),
                [package(’many-things’, ’0.1.0’)]),
    D).

sourcecode(D,’thing1 = "thing1"’).
```

Figure 3.11: Example for actually used declarations.

If we set up the data for the example packages from this section as described in section 3.5 we can deduce the relations shown in figure 3.11. The only declaration that the installation of `some-printer-0.1.0` actually uses is the one for `thing1`. We could avoid the unnecessary compilation of the other declarations.

3.7 Updates not affecting packages

In section 3.1 we introduce among others packages `aplication-0.1.0` and `favorites-0.1.1`. We explain why we can not install `application-0.1.0` with the newly released package `favorites-0.2.0`. The reason is that package `aplication-0.1.0` requires the symbol `number` but the update from package `favorites-0.1.1` to package `favorites-0.2.0` breaks it.

In figure 3.12 we show another package `color-output-0.1.0` that also depends on `favorites`. Its `main` declaration outputs a color imported from module `Favorites`. But in contrast to `application-0.1.0` it does not actually use the declaration for `number`. Therefore we can successfully compile `color-output-0.1.0` with `favorites-0.2.0`. But an upper bound on the dependency on `favorites` prohibits the installation just as the PVP dictates.

```
-- color-output.cabal
name:       color-output
version:    0.1.0

executable color-output
  main-is:       Main.hs
  build-depends: base >=4.7.0.0 && <4.8,
                 favorites >=0.1.0 && <0.2
```

```haskell
-- Main.hs
module Main where

import Favorites (color)

main :: IO ()
main = putStrLn color
```

Figure 3.12: Package `color-output-0.1.0`

Knowing `favorites-0.2.0` we can see that in this particular case the update from package `favorites-0.1.1` to package `favorites-0.2.0` does not affect the package `color-output-0.1.0`. The update for this package is unnecessarily prohibited.

In figure 3.13 we define, based on the model from section 3.5, when an update affects a package. An update is a pair of packages that have the same name but might have different versions. The second version does not have to be the immediately next version of that package. An update can jump several versions.

An update affects a package if the package requires a symbol that the update breaks. A package requires a symbol if it contains any declaration that mentions the symbol. An update breaks a symbol either if it removes it or if it alters it. An update removes a symbol if the first package provides it, but the second does not. A package provides a symbol if it contains any declaration that binds it. An update alters a symbol if there is a declaration in the first package that binds the symbol and there is a declaration in the second package that binds the symbol but the two declarations are different. Two declarations are different if their source code is different.

```
affects(Update, Package) :-
    requires(Package, Symbol),
    breaks(Update, Symbol).

requires(Package, Symbol) :-
    binds(Package, Declaration),
    mentions(Declaration, Symbol).

breaks(Update, Symbol) :-
    removes(Update, Symbol).
breaks(Update, Symbol) :-
    alters(Update, Symbol).

removes(update(Package1, Package2), Symbol) :-
    provides(Package1, Symbol),
    not(provides(Package2, Symbol)).

provides(Package, Symbol) :-
    contains(Package, Declaration),
    binds(Declaration, Symbol).

alters(update(Package1, Package2), Symbol) :-
    contains(Package1, Declaration1),
    contains(Package2, Declaration2),
    binds(Declaration1, Symbol),
    binds(Declaration2, Symbol),
    different(Declaration1, Declaration2).

different(Declaration1, Declaration2) :-
    sourcecode(Declaration1, SourceCode1),
    sourcecode(Declaration2, SourceCode2),
    not(SourceCode1 = SourceCode2).
```

Figure 3.13: Definition of an update affecting a package.

Two declarations being different could be defined in at least two other ways. We could define difference based on the type signature or we could define difference based on all transitively used declarations. In section 4.4 we discuss why we do not do this.

We conservatively assume that every change of the source code of a declaration is breaking. There is no general way to decide if a code change is breaking or not. We can imagine approximating the decision with tests, but this is not part of this thesis.

We call an update together with a using package an update scenario. In our case study we generate all update scenarios from our data and check if the update affects the using package. We only look at update scenarios where the first package in the update satisfies the version constraint of the using package. We assume that this means that the using package works with the first package in the update. Figure 3.14 makes the definition of update scenario more precise.

There are two interesting properties an update scenario may have. First, if not only the first but also the second package in the update satisfies the version constraint of the using package we say that the update scenario is allowed, if it does not we say that the update scenario is prohibited. Second, we say that an update scenario is major if the update involves a change in the major version. It is minor otherwise.

In the example in this section the update scenario is an update from package `favorites-0.1.1` to `favorites-0.2.0` for package `color-output-0.1.0`. It is prohibited because the version number of `favorites-0.2.0` does not satisfy the version constraint in package `color-output-0.1.0` on dependency `favorites` of `>=0.1.0 && <0.2`. It is major because the version numbers of `favorites-0.1.1` and `favorites-0.2.0` differ in the major part.

In figure 3.15 we list a few examples for the relations defined in this section. According to our definition the update from package `favorites-0.1.1` to package `favorites-0.2.0` does not affect package `color-output-0.1.0`. But the update from package `favorites-0.1.0` to package `favorites-0.1.1` does. In the first case the update alters `color` while in the second case the update alters `number`. The package `color-output-0.1.0` requires only `color`, it does not require `number`, so only the first update affects it. We as developers know that it affects the package in a beneficial way and therefore should be

```
update_scenario(
    update(Used_Package1, Used_Package2),
    Using_Package) :-
        update(update(Used_Package1, Used_Package2)),
        satisfies_constraint(Using_Package, Used_Package1).

update(update(Package1, Package2)) :-
    next_version(Package1, Package2).
update(update(Package1, Package2)) :-
    next_version(Package1, PackageInBetween),
    update(update(PackageInBetween, Package2)).
```

Figure 3.14: Definition of an update scenario.

allowed. It it not our concern to define or decide what beneficial means in this case.

In section 4.3 we find real world update scenarios that are prohibited but where the update does not affect the package. In chapter 5 we propose a code management architecture that sidesteps the problem.

```
not(affects(update(package(’favorites’,’0.1.1’),
                   package(’favorites’,’0.2.0’)),
            package(’color-output’,’0.1.0’))).
affects(update(package(’favorites’,’0.1.0’),
               package(’favorites’,’0.1.1’)),
            package(’color-output’,’0.1.0’)).

alters(update(package(’favorites’,’0.1.0’),
              package(’favorites’,’0.1.1’)),
       symbol(’color’,’Favorites’,’value’)).
alters(update(package(’favorites’,’0.1.1’),
              package(’favorites’,’0.2.0’)),
       symbol(’number’,’Favorites’,’value’)).

requires(package(’color-output’,’0.1.0’),
         symbol(’color’,’Favorites’,’value’)).
not(requires(package(’color-output’,’0.1.0’),
             symbol(’number’,’Favorites’,’value’))).
```

Figure 3.15: Example relations for an update affecting a package.

Chapter 4

A case study on Hackage

We want to know if and to what extent the problems described in sections 3.6 and 3.7 exist in real world Haskell code. To this end we do a corpus based empirical analysis. We fill a database with the facts introduced in section 3.5. We extract those facts from a selection of packages from Hackage. Hackage is a large Haskell package repository. We translate the definitions from sections 3.6 and 3.7 into queries against that database and interpret the results.

4.1 Methodology

A diagrammatic overview over our methodology is given in figure 4.1.

At the time of writing there are 49989 packages on Hackage. For reasons of scalability we are not able to analyze all of them and have to pick a sample as our corpus. We start by picking 10 packages at random. We then keep adding all versions of all dependencies of all these packages until convergence. The number of packages chosen this way is given in figure 4.2 together with other numbers. We want to maximize the number of update scenarios that we can generate from our dataset. We therefore make our package selection so that there are many versions of the same package and many dependencies between packages.

We extract and store meta-data about each selected package: Its name, its version, which packages satisfy its dependency constraints and its immediate next version if it has one. This is the first part of the data we describe in 3.5. To get the other part we install all selected packages with cabal-install. But instead of running an ordinary Haskell compiler we instruct cabal-install to run our custom

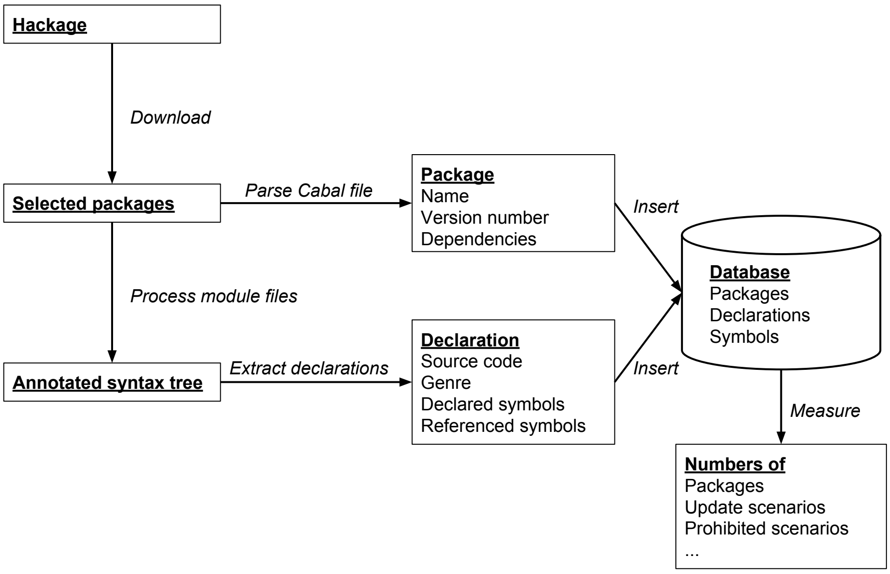

Figure 4.1: Overview of the fact extraction and measurement process.

tool on the module files in that package. The tool saves two files per module into a folder named after the package: A list of all declarations in that module and a list of all symbols the module exports. The lists of declarations is the data we need and the exported symbols are for name resolution in packages that depend on this one. Additionally, the case study in section 4.2 needs a list of installations. For each selected package we take the installation cabal-install chooses.

We do not start every run of cabal-install from a clean state but keep installed packages between runs to save run time. cabal-install sometimes does not want to reinstall a package because it is already present. We explicitly force those reinstalls because we sometimes need two instances of the same package differing only in their concrete dependencies.

We process each module file in the following way. We explicitly set includes and defines for the C preprocessor that many module files need. The includes and defines are usually implicitly set by GHC [18], the most widespread Haskell compiler. We parse the module file with the parser from `haskell-src-exts` [19]. Most module files contain annotations that they use language extesions. Some files use language extensions but do not explicitly say so, so we parse with those language extensions globally enabled. We then do name resolution with `haskell-names` [20] to annotate every symbol occurrence in the parsed abstract syntax tree with the origin of that symbol. We store a list of declarations. For each declaration we store its source code, the symbols it mentions and the symbols it binds. The source code of a declarations is a pretty printing of the abstract syntax tree of it. We use `haskell-src-exts` for pretty printing. Each symbol has a name, an original module and a tag identifying what the symbol refers to: for example `value`, `type`, `class` and so on. Strictly speaking this is more information than we need to disambiguate constructors and types, but `haskell-names` already provides exactly this information.

We insert the package meta-data and declarations that we extract from each package into a neo4j graph database [21]. Packages, declarations and symbols correspond to nodes. Packages have the properties of a package name and a version number. Declarations have their source code as their single property. Symbols have a name, an origin module and their name space as properties. Relationships are `contains`, `binds` and `mentions` and `installed_with`. For every declaration in a package there is a declaration node and a `contains` relationship from the package node to the declaration node. Every declaration node

Packages	728
Declarations	129093
Symbols	17291

Figure 4.2: Numbers of entities in our database

also has relationships `binds` and `mentions` to the corresponding symbol nodes. Additionally every package has an `installed_with` relationship to each package it was installed with. It is important that the package nodes are unique per package name and version number. It is also important that symbol nodes are unique per their name, origin module and name space so that we can establish connections between bound and mentioned symbols.

We store these facts in a database. Our measurements are queries against this database. We implement the queries in Java using the neo4j API. The facts and the queries are translations from those in sections 3.5, 3.6 and 3.7. Table 4.2 has a summary of the numbers of different nodes and relationships in our database at the time of writing.

4.2 Installation of unused declarations

In section 3.6 we motivate how an installation of a package may compile unnecessarily many declarations. We now look at the installations of the packages in our database and compare how many declarations the installation contains versus how many declarations the package actually uses. We translate the definitions of `installs_declaration` and `actually_uses` from section 3.6 to queries on our database.

We have 728 packages in our database. We look at one installation for each package. In figure 4.3 every dot corresponds to an installation for a package. On the horizontal axis we have the number of installed declarations while on the vertical axis we have the number of actually used declarations. It is consistently much lower.

In figure 4.4 we contrast the numbers of contained, installed and actually used declarations for a few chosen example packages. We see that the number of compiled declarations is significantly higher than the number of actually used ones.

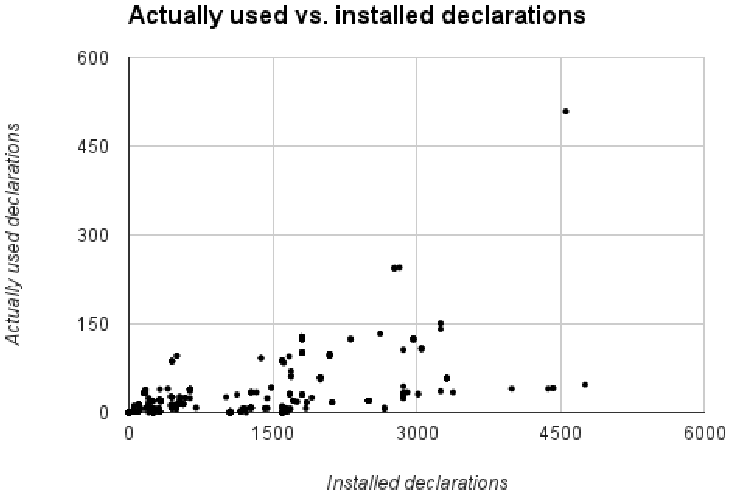

Figure 4.3: Installed declarations versus actually used declarations.

We conclude that installation of a package where we compile only the relevant declarations is much faster. But see section 4.4 for the caveats.

4.3 Updates not affecting packages

In section 3.7 we motivate and define update scenarios and when an update does not affect a package. In this case study we go through all update scenarios that we can generate on our database and classify them according to if the update affects the package or if it doesn't. We do this separately for update scenarios that are minor or major and for update scenarios that are prohibited or allowed.

The numbers of update scenarios grouped by properties are given in figure 4.5. We generate a total of 74,606 update scenarios. Of these 41,258 are minor. If packages follow the PVP minor update scenarios should never be prohibited. Indeed, we do not have any prohibited minor update scenarios in our database.

We divide the other 33,348 major update scenarios into 29,407 allowed ones and 3,941 prohibited ones. In 14,403 of the allowed scenarios the update affects the package and in the other 15,004 update scenarios it does not. In 1,896 of the

Package	Contained	Installed	Actually used
BiobaseDotP-0.1.0.0	15	433	27
bytestring-0.10.0.1	1369	41	1
ListLike-3.1.7	83	1600	87
ListLike-4.0.1	111	2768	244
statistics-0.10.5.2	599	173	38
parallel-1.1.0.1	125	1807	30
parallel-3.2.0.3	183	1641	7
aeson-0.3.0.0	194	3052	108
aeson-0.3.2.7	210	2967	124
binary-0.7.2.0	313	1807	101
utf8-string-0.3.7	208	0	0
utf8-string-0.3.8	208	1444	24
cereal-0.4.0.0	357	1807	128
parsec-3.1.5	332	2666	8
http-types-0.8.4	314	1193	6
hostname-1.0	4	0	0
network-2.3.1.0	602	2500	20
network-2.4.1.2	655	2500	20
network-2.5.0.0	657	2500	20
xml-1.3.9	286	0	0
zlib-0.4.0.4	145	0	0

Figure 4.4: Contained, installed, and actually used declarations.

Update scenarios				74,606
	Minor			41,258
		Allowed		41,258
			Affected	14,403
			Unaffected	15,004
		Prohibited		0
	Major			33,348
		Allowed		29,407
			Affected	14,403
			Unaffected	15,004
		Prohibited		3,941
			Affected	1,896
			Unaffected	2,045

Figure 4.5: Classification and count of update scenarios.

prohibited update scenarios the update affects the package and in 2,045 it does not.

We expected the number of update scenarios where the update does not affect the package to be much higher. It is roughly half the total number of update scenarios. This seems to be regardless of whether the update scenario is prohibited or allowed.

4.4 Threats to validity

The results of both case studies might be invalid for a number of reasons.

Some of the packages we initially select do not end up in our database. Either the solver built into cabal-install fails to find an install plan or our tool is unable to process a module because the package is only tested with GHC and makes implicit assumptions about preprocessor flags, language extensions or builtin modules. This causes preprocessing, parsing or name resolution to fail or be incomplete. If any of these fails we reject the entire package.

We were not able to validate the results. To validate the results from section 4.2 we would have to try and install each package together with pseudo-packages where we remove all unused declarations. To validate the results from section 4.3 we would have to try and install each package with the updated dependency. Both prove difficult because of the number of dependencies in each installation and the time they take to install, especially in the view of a clean state needed for each test install. We could go even further than testing if a package builds and run associated tests if there are any. But this clearly is future work.

There is always the possibility of an implementation error. We can not run the queries as described in chapter 3 directly but have to implement them in Java against the neo4j API. As this is error prone it is possible that we have not done this correctly.

Moreover the conclusions we draw may be false because of the following.

The selected packages might not be representative of Haskell packages. One reason could be that we only consider the library sections of packages and ignore executables. It is entirely possible that executables exhibit drastically different usage patterns of other packages. For example it might be that developers of library packages avoid too many dependencies.

On the one hand we assume that every code change is a breaking one, which is overly conservative. Instead of comparing the source code we might instead compare the type signatures of those declarations that have one. This would also not be fully satisfactory because type signatures being textually different does not imply that the types are different. Moreover a type could be more general and still compatible.

On the other hand we are overly liberal, because we do not compare declarations based on their transitive closure of other used declarations. A declaration can experience a breaking change because one of the declarations it uses experienced a breaking change, even if the source for the declaration stays the same. It is difficult to trace transitive uses if the declaration uses a declaration from another package because until dependency resolution fixes an installation it is unclear what the other package is.

The use of type class methods creates implicit dependencies on type class instances. It is only clear after instance resolution or in some cases even at runtime what the actually used instances are. This is out of scope of this thesis. We could conservatively assume that the use of a method means use of all instances for the methods class. This is left to future work.

Different installations of the same package might differ. Some packages have flags controlling different features of packages. The code in a package can differ depending on the package versions that dependency resolution picks. Haskell packages also come in different variants for different platforms. If we took into account all these different package installations we would have an increased number of almost identical packages. We disregard that fact and consider only one installation per package.

Chapter 5

Towards fragment-based code management

In code package managers a package is the unit of distribution. This means that developers download, compile and update whole packages. In chapter 3 we motivate problems that package-based code management has. In chapter 4 we analyze real world packages to find out how relevant these problems are.

In this chapter we propose an alternative to package-based code distribution that avoids the compilation of unused code fragments. The key design decision is to track dependencies between the smallest fragments of code possible. In this way it is possible to compile only necessary fragments. It should also be possible to statically and trivially know which fragments an update affects.

We want to distribute individual code fragments, but a code fragment alone is useless. We need all other code fragments it transitively uses as well. The used code fragments are its dependencies. We call a code fragment together with all code fragments it transitively uses a slice. Our use of the term slice is related to but not the same as the usual meaning from software slicing. The main difference is that usually slicing is done on the expression level, while we do it for larger code fragments.

Code slices are available in a repository. Developers use them in their code. They have to download and compile only those slices that they actually use. Figure 5.1 gives an overview over the proposed architecture. In its current form, our architecture does not support many common use cases that code package managers do support. Our long term goal is to create a fragment-based code manager.

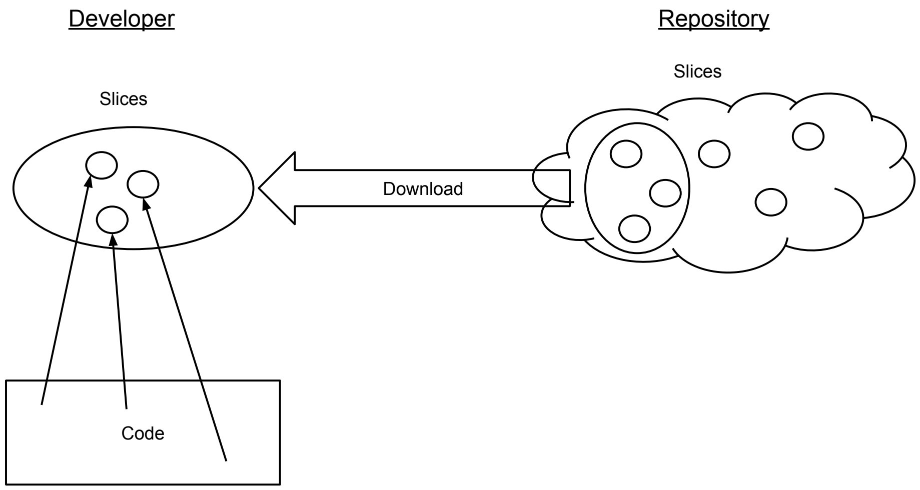

Figure 5.1: Architecture of fragment-based code management.

Two things that we imagine but that are missing from this picture are environments and updates. An environment is a set of modules that reexport symbols from a set of slices. You can think of them as virtual packages that reuse a common ground. We can imagine generating a package for a certain platform that does not contain declarations that do not build on that platform. In our architecture usages and therefore dependencies are fixed. This means we avoid dependency resolution. This also means that all updates have to be explicit. We envision an update as a set of replacements of a slice with another slice. Updates could be more fine-grained which makes backporting more pleasant or even unnecessary. Instead of using CPP we could model variants for different platforms as updates.

However, we present the implementation of a tool that provides an important part of the proposed architecture. We call the tool `fragnix` because it works on code fragments and is inspired by the Nix package manager [Dol06]. Right now our tool can take a set of modules and generate all contained slices. It can also compile a slice that binds a `main` symbol to an executable. It demonstrates how

```haskell
-- Main.hs
module Main where

import Greet (putHello)

main :: IO ()
main = putHello "Fragnix!"
```

```haskell
-- Greet.hs
module Greet where

putHello :: String -> IO ()
putHello x = putStrLn ("Hello " ++ x)

putHi :: String -> IO ()
putHi x = putStrLn ("Hi " ++ x)
```

Figure 5.2: Example application.

we can avoid compiling any unused code. Section 5.4 talks about future work and describes interesting development directions.

5.1 A small example program

We want to first extract all slices and then compile the small Haskell program from figure 5.2. There are two modules: `Main` and `Greet`. The `main` function uses the `putHello` function from the `Greet` module to print the string `"Hello␣Fragnix!"` to the console. It does not use the `putHi` declaration which is also declared in module `Greet`.

If we invoke fragnix on the example files `Main.hs` and `Greet.hs` it produces three slices and an executable called `main`. But it entirely avoids compilation of the `putHi` declaration. Even if we change the `putHi` declaration and invoke fragnix again no recompilation is necessary. Changes to `putHi` do not affect the executable.

The output of an invocation of fragnix on the example modules is shown in figure 5.3. To get the executable we invoke a Haskell compiler on two generated Haskell modules that respectively contain the two relevant code fragments for

```
>fragnix Main.hs Greet.hs
[1 of 2] Compiling F3521215606480787135 [...]
[2 of 2] Compiling F922946688791680081 [...]
Linking main ...
```

Figure 5.3: Example invocation of fragnix tool.

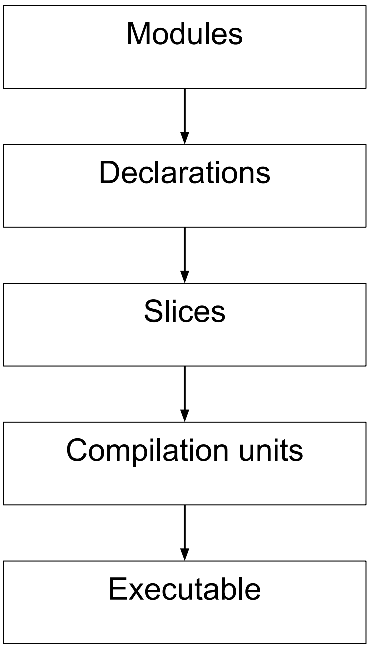

Figure 5.4: Data flow diagram for fragnix tool.

main and putHello. Those generated modules import exactly those symbols necessary for compilation and not more. An overview of the data flow is in figure 5.4.

5.2 Extracting slices from modules

We extract and store slices from modules. The slices should be shared in a central repository. For the example in figure 5.2 fragnix finds three slices: one for main, one for putHello and one for putHi. A json serialization of the one for main

```json
{
  "sliceID": 9229466887916080081,
  "fragment": [
    "main :: IO ()",
    "main = putHello \"Fragnix!\""
  ],
  "usages": [
    {
      "origin": {
        "otherSlice": 3521215606480787135
      },
      "usedName": {
        "valueIdentifier": "putHello"
      }
    },
    {
      "origin": {
        "originalModule": "System.IO"
      },
      "usedName": {
        "typeIdentifier": "IO"
      }
    }
  ]
}
```

Figure 5.5: Example slice.

is shown in figure 5.5. The important properties of a slice are the code fragment and a list of usages.

We give a unique ID to each slice. The ID is a hash of its code fragment and its usages. Because usages refer to other slices by their hash the hash includes all other code fragments that the slice transitively uses. In this case the hash is `9229466887916080081`. The code fragment of this example slice consists of two declarations: The type signature for `main` and its declaration. The slice has two usages: it uses the value `putHello` from another slice that we refer to by ID and the type `IO` that is builtin and comes from builtin module `System.IO`.

We extract all slices in a given set of modules in two steps. First, we extract all declarations from the given modules. A declaration here consists of the same information as in chapter 3: the source code, all bound symbols and all mentioned symbols. Second, we build up a graph where the nodes are declarations and the edges are usages. We also add pseudo usage edges from a declaration to its type signature if it has one. This graph might have cycles. Cycles are a problem because the hash of a slice recursively includes the hashes of other slices it uses.

We therefore eliminate all cycles by finding strongly connected components in this graph. Every strongly connected component consist of several nodes and therefore of several declarations. A strongly connected component uses another strongly connected component if any of its declarations uses any of the other components declarations. Now we create a slice from each strongly connected component where the strongly connected component's declarations form the slice's code fragment and the strongly connected component's usage edges are the slice's usages. We compute a hash for each slice in a bottom-up manner. This gives us slices that we serialize for further processing and possibly sharing.

5.3 Compiling slices

We compile the slice that binds the `main` symbol to an executable. This means that we also have to compile all other slices that the `main` slice transitively uses. We call these the relevant slices. We do not have to compile slices that are not relevant. We forge an ordinary Haskell module from each relevant slice. We call each such module a compilation unit. In the example there are two relevant slices. The corresponding compilation units are shown in figure 5.6. Both hide the implicitly imported `Prelude` and explicitly import only exactly those symbols that the included code fragment mentions. We do not generate modules for slices that are not relevant. We then invoke the Glasgow Haskell Compiler [18] on the generated modules.

The generated module for a slice has as its name the slice's ID prefixed with the letter F. The prefix is necessary because numbers can not be module names in Haskell. The generated module always uses the `NoImplicitPrelude` pragma to prevent the implicit import of the `Prelude` module. This prevents name clashes and serves as a check that we explicitly include all relevant usages. For each usage in the slice the generated module has one import statement. If the usage refers to another slice the import statement imports from the generated module of the other slice. If on the other hand the usage refers to a builtin symbol the

```
-- F922946688791680081.hs
{-# LANGUAGE NoImplicitPrelude #-}
module F922946688791680081 where
import F3521215606480787135 (putHello)
import System.IO (IO)

main :: IO ()
main = putHello "Fragnix!"

-- F3521215606480787135.hs
{-# LANGUAGE NoImplicitPrelude #-}
module F3521215606480787135 where
import System.IO (putStrLn)
import Prelude ((++))
import Prelude (String)
import System.IO (IO)

putHello :: String -> IO ()
putHello x = putStrLn ("Hello " ++ x)
```

Figure 5.6: Example compilation units.

import statement imports from the module that originally binds the mentioned symbol. The actual code in the generated module is exactly the code fragment of the slice.

We invoke GHC on the generated module that contains the fragment that binds `main`. We also tell GHC to use the folder that contains the generated modules as a search directory. GHC chases module imports and finds the corresponding generated module files. There are no unused imports in the generated modules. We do not compile any unused code fragments.

5.4 Future work

Our vision goes beyond the architecture that we present in this chapter. Our ultimate goal is to make dependencies lightweight enough that there is no second thought in reusing other people's code. It should also be easier to find and contribute individual declarations than cloning. This means we want not only faster installation and recompilation but also robustness against change. We want to systematically analyze use cases for code package managers. Based on this we want to design ways of user interaction.

Even if we do not use fragnix for code management we can use the existing infrastructure to statically analyze Haskell code. Because we know all transitive dependencies we can find code fragments that transitively use unsafe or deprecated features. We could also find real world examples of use to help learning about a package.

Finally we are also interested in how the idea extends from Haskell to other functional and non-functional languages.

Chapter 6

Conclusion

We have discussed two problems that are inherent in package-based code distribution. The first problem is the unnecessary compilation of code fragments that are not actually used. The second problem is that the Package Versioning Policy prohibits updates that do not even affect a package. We have conducted two case studies to evaluate the relevance of these problems.

Although our results are subject to quite a few threats to validity, we believe that we have shown that compilation time could drastically be reduced by early elimination of unused code fragments.

In about half of the prohibited updated scenarios the update does not affect the package. This means it is probably worthwhile to do more fine-grained dependency and update tracking and to automatically adjust the upper version bounds of packages.

We have explored how an alternative to package-based code distribution could look like. Instead of having packages as the unit of distribution, as well as the unit of compilation, documentation and update, we propose a system with more flexibility. The fragnix tool already delivers on the reduced compilation time, but how code distribution and version control work under this different paradigm is unclear. Our exploration is only at its very beginning.

Bibliography

[ACTZ12] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: A separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228 – 2240, 2012. Automated Software Evolution.

[ADC11] P. Abate and R. Di Cosmo. Predicting upgrade failures using dependency analysis. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 145–150, April 2011.

[ASDC$^+$12] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Why do software packages conflict? In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 141–150, June 2012.

[BJL13] Nikolaos Bezirgiannis, Johan Jeuring, and Sean 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.

[Boh02] Shawn A. Bohner. Extending software change impact analysis into cots components. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, SEW '02, pages 175–, Washington, DC, USA, 2002. IEEE Computer Society.

[BOL14] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79(0):241 – 259, 2014. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[CC05] Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, METRICS '05, pages 29–, Washington, DC, USA, 2005. IEEE Computer Society.

[CC06] Gerardo Canfora and Luigi 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.

[DJH02] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A formal specification of the haskell 98 module system. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 17–28, New York, NY, USA, 2002. ACM.

[Dol06] Eelco Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Universiteit Utrecht, January 2006.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan 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, March 2007.

[KDPJM14] Scott Kilpatrick, Derek Dreyer, Simon Peyton Jones, and Simon Marlow. Backpack: Retrofitting haskell with interfaces. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 19–31, New York, NY, USA, 2014. ACM.

[KM07] Huzefa Kagdi and Jonathan I. Maletic. Combining single-version and evolutionary dependencies for software-change prediction. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 17–, Washington, DC, USA, 2007. IEEE Computer Society.

[LSLZ13] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Softw. Test., Verif. Reliab.*, 23(8):613–646, 2013.

[MDBZ09] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas 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*, IWPSE-Evol '09, pages 57–62, New York, NY, USA, 2009. ACM.

[MKL$^+$14] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. The bug catalog of the maven ecosystem. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 372–375, New York, NY, USA, 2014. ACM.

[MT11] István Bozó Melinda Tóth. Building dependency graph for slicing erlang programs. *Periodica Polytechnica Electrical Engineering*, 55(3-4):133–138, 2011.

[Raj00] Václav Rajlich. Modeling software evolution by evolving interoperation graphs. *Annals of Software Engineering*, 9(1-2):235–248, 2000.

[RB06] Nuno F. Rodrigues and Luís S. Barbosa. Component identification through program slicing. *Electronic Notes in Theoretical Computer Science*, 160(0):291 – 304, 2006. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005) Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[RDV13] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 221–224, Piscataway, NJ, USA, 2013. IEEE Press.

[Szl11] Marcin Szlenk. Metamodel and uml profile for functional programming languages. In Wojciech Zamojski, Janusz Kacprzyk, Jacek Mazurkiewicz, Jarosław Sugier, and Tomasz Walkowiak, editors, *Dependable Computer Systems*, volume 97 of *Advances in Intelligent and Soft Computing*, pages 233–242. Springer Berlin Heidelberg, 2011.

[Tip94] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[TLO10] Paulo Trezentos, Inês Lynce, and Arlindo L. Oliveira. Apt-pbo: Solving the software dependency problem using pseudo-boolean optimization. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 427–436, New York, NY, USA, 2010. ACM.

Technical Documentation

[1] Haskell declarations. https://github.com/phischu/haskell-declarations. Accessed: 2015-01-22.

[2] Master thesis script. https://github.com/phischu/master-thesis-script. Accessed: 2015-01-22.

[3] Declaration import. https://github.com/phischu/declarationimport. Accessed: 2015-01-22.

[4] Master report. https://github.com/phischu/masterreport. Accessed: 2015-01-22.

[5] Fragnix. https://github.com/phischu/fragnix. Accessed: 2015-01-22.

[6] Semantic versioning 2.0.0. http://semver.org/. Accessed: 2015-01-22.

[7] FP Complete. Stackage server. http://www.stackage.org/. Accessed: 2015-01-22.

[8] Silk. bumper: Automatically bump package versions, also transitively. https://hackage.haskell.org/package/bumper. Accessed: 2015-01-22.

[9] Developing with sandboxes. https://www.haskell.org/cabal/users-guide/installing-packages.html#developing-with-sandboxes. Accessed: 2015-01-22.

[10] Multi-instance packages. https://ghc.haskell.org/trac/ghc/wiki/Commentary/Packages/MultiInstances. Accessed: 2015-01-22.

[11] Eternal compatibility in theory. https://www.haskell.org/haskellwiki/The_Monad.Reader/Issue2/EternalCompatibilityInTheory. Accessed: 2015-01-22.

[12] Tim C. Schroeder. hackage-diff: Compare the public api of different versions of a hackage library. http://hackage.haskell.org/package/hackage-diff. Accessed: 2015-01-22.

[13] Thomas Schilling. Specifying dependencies on haskell code. https://www.haskell.org/pipermail/cabal-devel/2008-May/002799.html. Accessed: 2015-01-22.

[14] Thomas Schilling. Beyond package version policies. http://nominolo.blogspot.de/2012/08/beyond-package-version-policies.html. Accessed: 2015-01-22.

[15] Simon Marlow. Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010/. Accessed: 2015-01-22.

[16] The haskell cabal: Common architecture for building applications and libraries. https://www.haskell.org/cabal/. Accessed: 2015-01-22.

[17] Package versioning policy. https://www.haskell.org/haskellwiki/Package_versioning_policy. Accessed: 2015-01-22.

[18] The glasgow haskell compiler. https://www.haskell.org/ghc/. Accessed: 2015-01-22.

[19] Niklas Broberg. haskell-src-exts: Manipulating haskell source: abstract syntax, lexer, parser, and pretty-printer. https://hackage.haskell.org/package/haskell-src-exts. Accessed: 2015-01-22.

[20] Roman Cheplyaka. haskell-names: Name resolution library for haskell. https://hackage.haskell.org/package/haskell-names. Accessed: 2015-01-22.

[21] Inc. Neo Technology. Neo4j, the world's leading graph database. http://neo4j.com/. Accessed: 2015-01-22.