A Package Manager for Curry

Jonas Oberschweiber

Master-Thesis
eingereicht im September 2016

Christian-Albrechts-Universität zu Kiel
Programmiersprachen und Übersetzerkonstruktion

Betreut durch: Prof. Dr. Michael Hanus und M.Sc. Björn Peemöller

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

_____

Contents

1

Introduction

Modern software systems typically rely on many external libraries, reusing functionality that can be shared between programs instead of reimplementing it for each new project. In principle this reuse of functionality is, of course, desirable: less time is spent reinventing the wheel and existing libraries are more likely to have seen real-world use and the bug fixes that come along with it. Heavy use of libraries does, however, come with its own set of challenges.

First, a user has to *find* a library that offers the functionality they need. Then they have to *acquire* the library and *integrate* it into their project, i.e., alter the compiler's or interpreter's load path, customize the `Makefile` or whatever else their programming environment might require. Next, they have to ensure that any libraries required by the original library are also present in the correct versions. If some library is required by two other libraries, they have to ensure that its version is compatible to both – if such a version even exists. When the next programmer is added to the project and sets up their development environment, they have to repeat many of these steps manually.

To improve discoverability of new libraries and automate some of the steps surrounding installation and integration, *package management systems* have been developed for many modern programming languages. Package management systems are usually able to install packages from a centralized package repository and ensure that any dependencies – other packages that the original package needs to function – are installed as well. They are integrated into the programming language ecosystem and can automatically make installed packages available to the compiler or interpreter.

The goal of this thesis is the development of a package management system for the integrated functional logic programming language Curry. This system should be able to install packages, resolve dependencies, and feature a simple user interface. Additionally, it should itself be written in Curry and be compatible to the two major Curry compilers, PAKCS and KiCS2.

In the next chapter, we give a cursory introduction to the Curry programming language. Chapter 3 discusses package management systems and resolution of dependencies in general and gives a brief overview of three modern package management systems. In Chapter 4, we discuss the design of the Curry package manager and Chapter 5 describes the implementation of that design. Chapter 6 evaluates how two of the system's major features work in practice and gives some performance figures. Finally, we present conclusions and possible improvements in Chapter 7.

2

The Curry Programming Language

Curry is an integrated functional logic language that combines the functional and logic programming paradigms into one language. Syntactically and in its functional features, it is similar to Haskell [Pey03], most notably lacking support for type classes. We assume the reader to be familiar with basic Haskell syntax and semantics. A complete description of Curry is given in the Curry Report [Han+16a].

There are two major compilers for Curry, the Portland Aachen Kiel Curry System, PAKCS[1], which compiles Curry programs to Prolog programs, and the Kiel Curry System 2, KiCS2 [Bra+11], which compiles to Haskell.

Throughout this chapter, we will give a brief introduction to Curry's logic features before turning our attention to *AbstractCurry*, a representation of Curry programs as Curry data types that can be used for metaprogramming. Afterwards, we will briefly discuss the Curry compiler ecosystem.

2.1 Curry's Logic Features

Since Curry's functional programming features are largely similar to Haskell's, we will only give an introduction to its logic programming constructs, namely nondeterminism and free variables.

Operations in Curry can be non-deterministic, i.e., they can return multiple results

[1] `https://www.informatik.uni-kiel.de/~pakcs/`

for one input. The most basic non-deterministic operation is Curry's built-in operator "?", which returns both of its arguments non-deterministically. The following and all further examples are evaluated using KiCS2 version 0.5.1. Lines beginning with > are inputs given to the interactive Curry environment, lines without such a prefix are the results given by the system.

```
> 0 ? 1
0
1
> "a" ? "b"
"a"
"b"
```

While Haskell tries the rules of a function in the order they are specified and stops at the first match, Curry will try all rules, evaluate all that match and return the results of those evaluations. The function `addBoth` defined below results in both `0` and `10` if called on `(0, 0)`.

```
addBoth :: (Int, Int) -> Int
addBoth (x, _) = x
addBoth (_, x) = x + 10

> addBoth (0, 0)
0
10
```

In addition to non-determinism, Curry includes support for free variables. Free variables are not bound to a value until the expression is evaluated. The Curry system then generates possible values for free variables, binds them to those values and evaluates the expression. Since multiple values can be generated, expressions containing free variables are non-deterministic. In the following example, the variable `xs` is declared as free using `where xs free`. Each generated value of `xs` is printed in braces with the result of the original expression printed afterwards.

```
> [] ++ xs == [1] where xs free
{xs = []} False
{xs = ((-_x3):_x4)} False
{xs = (0:_x4)} False
{xs = [1]} True
{xs = ((2 * _x5):_x4)} False
```

```
{xs = (1:_x6:_x7)} False
{xs = ((2 * _x5 + 1):_x4)} False
```

Note that `[1]` is the only generated value for `xs` which makes the expression true. If constraints on free variables are used in guards, then the guarded rule will only be evaluated for bindings of the free variables that make the guard expression true. We can then use the now-bound variables inside the function body. As an example, we define a function `prefix` that will return all possible prefixes of a list using free variables:

```
prefix :: [a] -> [a]
prefix xs | ys ++ zs == xs = ys where ys, zs free

> prefix [1,2,3]
[]
[1]
[1,2]
[1,2,3]
```

2.2 *Abstract Curry*

Curry's has powerful built-in metaprogramming capabilities via *AbstractCurry*. AbstractCurry is a representation of Curry programs as Curry data types. Functions are included for reading Curry programs into AbstractCurry form and for pretty printing AbstractCurry terms into Curry programs. Additionally, many helper functions are provided for selecting specific parts of an AbstractCurry program, for building AbstractCurry programs, and for transforming AbstractCurry programs. In this section, we will give a brief overview of the main data types that make up an AbstractCurry program. All of these types are defined in the `AbstractCurry.Types` module.

An AbstractCurry program is represented by the `CurryProg` type, which has one constructor of the same name taking a module name, a list of imported modules and lists of type, function and operator declarations.

```
type MName = String

data CurryProg = CurryProg MName [MName] [CTypeDecl] [CFuncDecl]
```

```
  [COpDecl]
```

A type declaration, `CTypeDecl`, is either a data type, a type synonym, or a `newtype`, which is a special case of a data type.

```
type QName = (String, String)

data CTypeDecl = CType    QName CVisibility [CTVarIName] [CConsDecl]
               | CTypeSyn QName CVisibility [CTVarIName] CTypeExpr
               | CNewType QName CVisibility [CTVarIName] CConsDecl

data CConsDecl = CCons   QName Visibility [CTypeExpr]
               | CRecord QName CVisibility [CFieldDecl]
```

All three constructors take a name, a visibility – whether or not the type is exported from the module – and a list of type variables used in the following constructors or type expressions. Additionally, type synonyms take the type expression they are a synonym for, while data types and newtypes take a list of constructors or a single constructor. A constructor can either be a simple constructor taking positional arguments (`CCons`) or a record constructor (`CRecord`).

A function declaration, `CFuncDecl`, consists of a name, the function's arity, its visibility, its type and a list of rules. There is an additional constructor, `CmtFunc`, that is similar except for an additional parameter for a comment.

```
data CFuncDecl = CFunc QName Arity CVisibility CTypeExpr [CRule]
```

While we have only given a very superficial introduction to AbstractCurry, it is sufficient for understanding the parts of this thesis that make use of AbstractCurry.

2.3 *The Compiler Ecosystem*

As mentioned in the introduction, the two major Curry compilers under active development are PAKCS and KiCS2. Both compilers share the same frontend written in Haskell and only differ in their backends which generate the target code. Additionally, many settings and command line parameters are the same in both systems. While they each provide their own executable to launch the compiler and interactive environment, the `pakcs` and `kics2` commands, both also include an alias called

`curry`. The `curry` command can be used to launch an interactive Curry environment when either compiler's `bin` directory is on the path.

Additionally, both compilers use the directories from the `CURRYPATH` environment variable to search for modules when an `import` is encountered, which will turn out to be very useful in Chapter 4.

Both compilers ship with the same set of standard libraries that cover a variety of areas: essential data types and operations on those data types, GUI programming, web programming, data structures and algorithms, database access, as well as libraries for metaprogramming such as AbstractCurry. Additionally, a set of tools for tasks such as generating documentation web sites from specially formed comments or executing unit and property tests is included in both distributions.

3

Package Management Systems

Depending on the context in which is it used, the term *package* can have many different legitimate uses when talking about software. In Java, a package is a language concept somewhat similar to hierarchical modules in Curry and Haskell. When talking about package management systems, we use the term *package* to refer to a distribution of a software component in binary or source code form and its accompanying metadata such as a package name, author and version. Packages are often distributed in the form of archive files (such as TAR or ZIP). We will use the term *package file* to refer to these archive files explicitly.

A package management system is a software system tasked with installing and updating packages in an environment. In the case of system-wide package managers such as Debian Linux's *apt*[1] or Android's *Google Play Store*[2], the environment is the operating system instance. For more specialized package managers, e.g. programming-language specific ones such as the examples discussed in the last few sections of this chapter, the environment can be a system-wide, per-user or even per-directory collection of packages, or a combination of the above.

Packages often need other packages to function correctly and these *dependencies* are usually listed in the package's metadata. For example, a package for manipulating image files might depend on a package supplying a decoder for the JPEG image format. Ensuring that a package's dependencies are met when it is installed and/or used and that there are no conflicts between the dependencies of different packages is part of a package management system's tasks. Package dependencies are

[1] `https://www.debian.org/doc/manuals/debian-reference/ch02.en.html`
[2] `https://play.google.com/store?hl=en`

described further in Section 3.2.

In the next two sections, we will discuss version numbers and their semantics as well as dependency management in general. Afterwards, we will turn our attention to three specific package management systems in common use today and give a brief overview of the design decisions made during their implementation and the effect these decisions have on those system's day-to-day usability.

3.1 *Semantic Versioning*

Many different schemes are used to version software releases, from simple increasing numbers to codenames. One of the more common ways to version software is using a hierarchy of numbers separated by *dot* characters, e.g. 1.0, 1.0.2 or 2.5.3.1002. The meaning of the different version components can vary from project to project and organization to organization. The Eclipse and Apache Commons open source projects, for example, each have their own guidelines on when to increment which component of the version number[3][4].

Semantic Versioning aims to be a technology-agnostic standard on the structure and semantic meaning of software version numbers. It has seen adoption in different programming communities, such as the Ruby and JavaScript ecosystems. In this section, we will give a brief overview of the most important aspects of version 2.0.0 of the standard, which is the current version as of this writing. The full specification can be found in [Pre16].

A semantic versioning version number is made up of three integers separated by *dot* characters and, optionally, additional specifiers for pre-release versions (e.g. alpha or beta versions) and build metadata. The first, second and third components are called the *major*, *minor* and *patch* versions. Semantic versioning is meant to be applied to software systems that can be used by other software systems programatically. Thus, all descriptions of semantic meaning of a version number change in the semantic versioning standard refer to the public application programming interface (API) of the versioned software system.

If the major version is set to zero, e.g. 0.5.2, then the system is considered to be in the initial development phase. Any changes to the public API are permitted and

[3] `https://wiki.eclipse.org/Version_Numbering`
[4] `https://commons.apache.org/releases/versioning.html`

no stability guarantees are made. Version 1.0.0 defines the first stable public API, subject to the following rules:

- The patch version must be incremented if a new release contains only bug fixes, i.e., changes that correct previously incorrect behavior. Backwards compatibility to the previous release must be maintained, except for bug fixes. Nothing must be added to or removed from the public API and no API interface must change.

- The minor version must be incremented if new functionality is introduced, for example if new functions are added to the public API. Any new functionality must not affect backwards compatibility of the existing functionality. Additionally, a minor version may also include bug fixes. The patch version must be reset to zero when the minor version is increased.

- If a new version contains any backwards incompatible changes, the major version must be incremented. Backwards incompatible changes include removing or renaming public APIs and changing existing behavior. Patch and minor versions must be reset to zero if the major version is increased.

  Pre-release versions are denoted by a *hyphen* ("-") followed by a string of ASCII alphanumeric and hyphen characters. Semantically, a pre-release version indicates that the software may be unstable and that the normal version semantics may not apply. Version 1.1.0-beta1 might be missing some parts of the public API from 1.0.0, for example. Build metadata is a part of the semantic versioning specification, but is not currently supported or used by the Curry package manager and thus not described here.

  The semantic versioning standard also describes how to compare two versions to one another. We can formalize this description into a total order on versions if we represent a version as a four-tuple of three version numbers and an optional pre-release specifier. To this end, we define $\Sigma_{pre} := \{\text{-}, 0, ..., 9, A, ..., Z, a, ..., z\}$ as the alphabet of valid characters for pre-release specifiers. We then define $V := \{(a, b, c, p) | a, b, c \in \mathbb{N}, p \in \Sigma_{pre}^*\}$ to be the set of all semantic versioning version numbers, i.e., the major, minor and patch version numbers are natural numbers while pre-release specifiers are words over the alphabet Σ_{pre}. Versions without a pre-release specifier have the empty word (ϵ) as their last component. We define $\leq_{\Sigma_{pre}}$ to be the order on Σ_{pre} that sorts all characters into the order given in the definition of the set above, which is the same order the characters are given in the ASCII character set. $\leq_{\Sigma_{pre}}$ is a total order on Σ_{pre}, since a function $f : \Sigma_{pre} \to \{0, ..., 62\}$ that assigns 0 to -, 1 to 0 and so on is an order isomorphism from $(\Sigma_{pre}, \leq_{\Sigma_{pre}})$ to $(\mathbb{N}, \leq)$.

Using $\leq_{\Sigma_{pre}}$ and the shortlex order $\leq_{sx}$ on Σ^*_{pre} (see [Sip12]), which is itself a total order, we can define the order $\leq_{pre}$ on Σ^*_{pre}. In shortlex, shorter strings precede longer strings while strings of the same length are compared lexicographically. By the rules laid out in the semantic versioning standard, if both pre-release specifiers are purely numeric, i.e., all characters are digits, then the specifiers are compared as integers. If one of the specifiers is numeric, but the other is not, then the numeric specifier is always considered smaller. If both specifiers are non-numeric, then they are compared using shortlex order. Assuming a function *strToInt* that will map a numeric pre-release specifier to corresponding natural number, we define:

$$\begin{aligned}
\leq_{pre} := &\{(a,b) \mid a,b \in \Sigma^*_{pre},\ a,b \text{ numeric},\ strToInt(a) \leq strToInt(b)\} \\
&\cup \{(a,b) \mid a,b \in \Sigma^*_{pre},\ a \text{ numeric},\ b \text{ non-numeric}\} \\
&\cup \{(a,b) \mid a,b \in \Sigma^*_{pre},\ a,b \text{ non-numeric},\ a \leq_{sx} b\}
\end{aligned}$$

$\leq_{pre}$ is a total order on Σ_{pre}, a proof is given in Appendix A. We now use $\leq_{pre}$ to define an order $\leq_{ver}$ on V:

$$\begin{aligned}
prC((a,b,c,p_1),(x,y,z,p_2)) &:= p_1 \leq_{pre} p_2 \\
paC((a,b,c,p_1),(x,y,z,p_2)) &:= c < z \vee (c = z \wedge prC((a,b,c,p_1),(x,y,z,p_2))) \\
miC((a,b,c,p_1),(x,y,z,p_2)) &:= b < y \vee (b = y \wedge paC((a,b,c,p_1),(x,y,z,p_2))) \\
maC((a,b,c,p_1),(x,y,z,p_2)) &:= a < x \vee (a = x \wedge miC((a,b,c,p_1),(x,y,z,p_2))) \\
\leq_{ver} &:= \{(v_1,v_2) \mid v_1,v_2 \in V,\ maC(v_1,v_2)\}
\end{aligned}$$

It can be shown that $\leq_{ver}$ is a total order on V, a proof is given in Appendix A.

3.2 *Dependency Management*

Packages often need other packages to function correctly. For example, a JSON (see [Bra14]) parser package might depend on a package offering parser combinators. Each package typically lists the packages it depends on in its metadata, including *constraints* on the range of versions of these package it is compatible to. Version 1.0.0 of the JSON package might require the parser combinator package in a specific

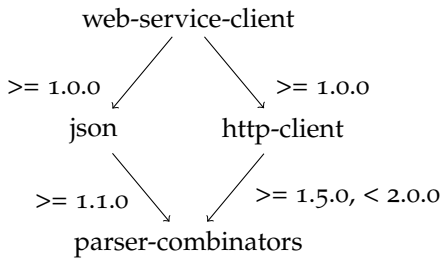


Figure 3.1.: A simple dependency graph

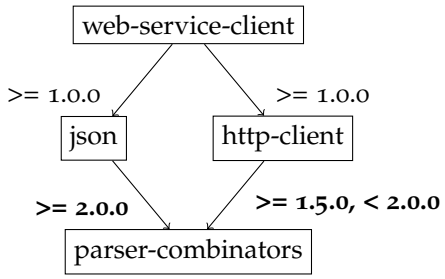


Figure 3.2.: A conflicting dependency graph

version, e.g. 1.1.5, or specify an acceptable range of versions, e.g. anything above 1.1.0 but below 2.0.0.

Dependencies can also be transitive: if a dependency has dependencies of its own, then those dependencies become dependencies of the original package. A web service client might, for example, depend on some HTTP client package and the JSON package. The web service client package then also depends on the parser combinator package.

The direct and transitive dependencies of a package form a *dependency graph*. Each package is a node in the graph. A directed edge from package A to package B signifies that package A depends on package B. The edges are labeled with the version constraints of the dependencies. An example dependency graph for the web service client described above is shown in Figure 3.1. The example graph also shows why the dependency graph is a graph and not just a tree: multiple (transitive) dependencies of the original package can depend on the same package. In the example graph, both the `json` and `http-client` packages depend on the `parser-combinators` package.

Packages with multiple incoming edges, i.e., packages that are depended upon by more than one other package, are a common source of conflicts: if the version constraints on two or more of these edges are incompatible, then there is no way to choose one of the available versions that will satisfy all constraints. Figure 3.2 shows a dependency graph with conflicts. The `json` package requires `parser-combinators` in version 2.0.0 or higher, while the `http-client` package cannot work with anything newer than 1.*y*.*z*. There is no way to form a set containing only one version of `json`, `http-client` and `parser-combinators` that will satisfy all version constraints.

We will give a few definitions adapted from Burrows [Buro5] to make the above observations more precise. As above, a package is simply defined by its name, e.g. *json* or *parser-combinators*. We define $\mathcal{P}$ to be the set of all packages known to a package system. A package version is represented by the name of the package and its version number separated by a hyphen, e.g. *json-1.0.0* or *parser-combinators-3.8.1-b5*. We define $\mathcal{V}$ to be the set of all package versions known to a package system and $PkgOf(v)$ to be the package of any package version v:

Definition 1. *Given a package version $v \in \mathcal{V}$, $PkgOf(v) \in \mathcal{P}$ is the corresponding package.* △

Next, we define *version constraints*:

Definition 2. *A version constraint is written p OP ver, where $p \in \mathcal{P}$ is a package name, OP is one of the operators $=, >, \geq, <, \leq$, and ver $\in V$ is a version number.* △

An example version constraint is *json* $\geq$ 1.0.0. A version constraint is satisfied for a package version v iff $PkgOf(v)$ equals the package in the version constraint and v's version satisfies the condition. If version numbers are in the semantic versioning format discussed in the previous section, we can use the $\leq_{ver}$ relation to check when a constraint is satisfied. Note that individual package managers may support different version number formats and operators.

We can build *combined version constraints* from individual version constraints:

Definition 3. *A combined version constraints is a combination of one or more version constraints for the same package using the boolean operators **and** ($\wedge$) and **or** ($\vee$) in disjunctive normal form, i.e., a disjunction of conjunctions of version constraints. A combined version constraint is **satisfied** iff its value is true when satisfied individual version constraints are interpreted as true and non-satisfied version constraints as false.* △

An example combined version constraint that requires either the JSON package in

a version between 1.0.0 and 2.0.0 or from version 3.0.0 onwards is: $(\mathit{json} \geq 1.0.0 \wedge \mathit{json} < 2.0.0) \vee \mathit{json} \geq 3.0.0$.

Additionally, we define *VPkg* and *VCPkg* for version constraints and combined version constraints:

Definition 4. *Given a version constraint $c = p\ OP\ ver$, we define $VPkg(c) := p$, i.e., the package referenced by the version constraint.* △

Definition 5. *Given a combined version constraint d, the set*

$\{VPkg(c) \mid c \text{ is a version constraint in } d\}$

contains all packages referenced by the version constraints that make up d. Note that since all version constraints in a combined version constraint must reference the same package, it will always be a singleton set. We define $VCPkg(d)$ to be the sole member of that singleton set, i.e., the package depended upon by the combined version constraint d. △

We can now define *dependency constraints*, which let us specify which other packages a package version depends on.

Definition 6. *Given a package version v and a combined version constraint c, we write $v \Rightarrow c$ if v depends on a package version that satisfies c. $v \Rightarrow c$ is called a dependency constraint. $\mathcal{S}$ is the set of all dependency constraints.* △

For example, if *json-1.0.0* depends on *parser-combinators* in version 1.0.0 or above, we can write *json-1.0.0* $\Rightarrow$ *parser-combinators* $\geq$ 1.0.0. Next, we define *DependingPkg*, *Deps* and *DepPkgs*:

Definition 7. *Given a dependency constraint $d = v \Rightarrow c \in S$ where v is a package version and c is a combined version constraint, we define $DependingPkg(d) := v$, i.e., the depending package version, and $DependedVC(d) := c$, i.e., the combined version constraint depended upon.* △

Definition 8. *Given a package version v, we define*

$Deps(v) := \{d \in \mathcal{S} \mid DependingPkg(d) = v\}$.

$Deps(v)$ is the set of of all known dependency constraints for v. △

Definition 9. *Given a package version v, we define*

$DepPkgs(v) := \{VCPkg(d) \mid d \in \mathcal{S} \text{ with } DependingPkg(d) = v\}$.

$DepPkgs(v)$ is the set of all package names mentioned in the dependency constraints of v, i.e., the packages that v depends on directly. △

Note that a concrete implementation of a package management system might not offer exactly the same capabilities as described here using versions and dependency constraints, for example, not all systems offer boolean disjunctions.

Given a set of package versions I, we say that I satisfies a dependency constraint d, written $I \vdash d$, if d is satisfied by the package versions in I. We call a set I of package versions *consistent* if for all dependency constraints d of all package versions in I, I either satisfies d or I does not contain a version of $VCPkg(DependedVC(d))$.

We say that I is *complete* if it contains exactly one version of each package depended upon by a package version in I, i.e., of each package in $\cup\{DepPkgs(v) \mid v \in I\}$. If I is both consistent and complete, we say that I is a *solution*.

The process of extending I with additional package versions from $\mathcal{V}$ until it is a solution is called *dependency resolution*. The resulting set is called $R(I)$.

From now on, we will use the term *version constraint* to mean both simple version constraints as well as combined version constraints.

Dependency resolution can take place at different times and with sets of package versions of varying sizes. In some package management systems, e.g. most operating system package managers or Haskell's Cabal (see Section 3.5), dependency resolution takes place at install time. The set of installed packages must always form a solution, i.e., a newly installed package version must leave it consistent and complete and only one version of a package can be installed at any given time. Other package management systems perform dependency resolution at run-time or compile-time on a much smaller set of package versions, namely the packages that are required to run or compile the program. The set of installed packages versions does not have to be a solution. Ruby's Bundler and JavaScript's npm both use this approach, see Sections 3.3 and 3.4, respectively.

3.3 Ruby's Gems and Bundler

Ruby ships with a default package manager called RubyGems that can install and manage Ruby packages, called *gems*. Gems are installed globally in the current Ruby

installation via the *gem* command, e.g. `gem install rails`. A gem can be installed from a local `.gem` file, which is just a ZIP archive with a different extension, or from a central repository of gems, which offers an API that the *gem* utility can use to find and download gems.

RubyGems is integrated into the Ruby language: if the `json` gem is installed on the system, then its files are automatically added to Ruby's load path and it is available for use via `require 'json'` [5]. Since it is possible to install multiple versions of a gem, RubyGems will run a dependency resolution algorithm when a gem is required from a Ruby program: the first gem that is loaded will always be loaded in the newest available versions, unless the user explicitly specifies another version to load. Each subsequent gem will be loaded in the newest version that is compatible with the set of already loaded gems. If no compatible version exists on the system, execution of the current program is stopped with an error.

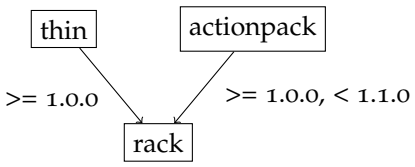


Figure 3.3.: A dependency graph with different results depending on order.

Note that using this approach, the order in which gems are loaded has an effect on the versions loaded and even whether or not a compatible set of gems can be found at all. Consider the scenario shown in Figure 3.3, taken from [Kat10]. If we assume that the system has the `rack` gem installed in versions 1.0.5 and 1.1.3, the following two `requires` will result in an error:

```
require 'thin'
require 'actionpack'
```

Since `thin`'s only requirement is that the version of `rack` be greater than or equal to 1.0.0, RubyGems will load `rack` in version 1.1.3. When `actionpack` is required, RubyGems checks if the already loaded version of `rack` is compatible with its dependency requirements. Since `actionpack` explicitly declares that is not compatible with `rack` in version 1.1.0 or above, execution stops with an error. The following load order will, however, not result in an error:

[5] The `require` command is used to load other files into a Ruby program. The load path is a list of directories that are searched when a file is required without an absolute path.

```
require 'actionpack'
require 'thin'
```

`actionpack` is loaded first, and with it `rack` in version 1.0.5. Since 1.0.5 also matches the dependency specification of `thin`, both gems can be loaded without error.

This consecutive resolution of dependencies can quickly become unwieldy in larger projects, especially since gems may themselves require other gems in unknown order. Because of this and a few other shortcomings (see [Kat10] for a detailed explanation), the Ruby community developed the Bundler [6] tool. Bundler provides a way to explicitly declare and install a list of dependencies for a single Ruby project. To this end, the user can create a file called `Gemfile` in the project's root directory in which they list the gems their project needs, including the desired versions or version ranges. A `Gemfile` for the example above might look like this:

```
gem 'actionpack'
gem 'thin'
```

When the user runs `bundle install` in the project's root directory, Bundler will check if there are versions of `actionpack`, `thin` and their dependencies installed on the system that form a compatible set and choose the newest compatible version of each gem. If some packages are missing, it will ask RubyGems to install them in the newest compatible version. Dependency resolution is no longer done after each `require`, but ahead of time, with all dependencies known to the dependency resolution algorithm. In addition, a developer new to a Ruby project but familiar with the language and its ecosystem knows where to find all project dependencies and how to install them.

To further increase the ease of collaboration and to prevent bugs arising from slightly different gem versions being used on different developer's systems, Bundler creates a `Gemfile.lock` file when `bundle install` is first run. In this file, it notes the exact versions of all gems installed and used during the installation run. When `bundle install` is run again, this time with a `Gemfile.lock` file present in addition to the `Gemfile`, Bundler will use the exact versions noted in the `Gemfile.lock`, even if different – potentially newer – versions of some gems are available that satisfy the version constraints in the `Gemfile`. This ensures that exactly the same gem versions are used on different systems.

[6] `http://www.bundler.io`

3.4 JavaScript's npm

npm is a popular package manager in the JavaScript ecosystem and comes bundled with the widely-used JavaScript environment *Node.js*. It encourages creating many small packages that only do a few things each, resulting in a large number of packages published to the central npm package index. The creators of npm claim that over a 250,000 packages have been published thus far.[7]

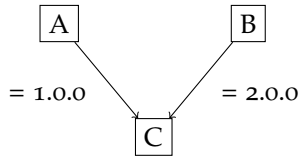


Figure 3.4.: A dependency graph with the same package required in two versions.

npm's most distinguishing feature is that it allows the user to not only install multiple versions of a package, but also use multiple versions during one execution run. A scenario in which the user's package depends on packages *A* and *B*, which depend on different version of package *C* (e.g. *A* depends on *C-1.0.0* while *B* depends on *C-2.0.0*, see Figure 3.4) does not lead to a dependency conflict in npm. Instead, the code in package *A* can use the code from *C-1.0.0* while the code in package *B* can use the code from *C-2.0.0*. This is made possible by the way Node.js loads JavaScript code.

Node.js provides a `require` function to import other JavaScript modules.[8] Unlike Ruby's `require` statement or Haskell's `import`, Node's `require` function does not load the definitions of the required module into the global environment. Instead, the other module's exports are returned from the function. In the following example, the module `foo.js` loads the module `circle.js` from the same directory and uses its exports.[9] The contents of `circle.js`:

```
const PI = Math.PI;
exports.area = function(r) {
  return PI * r * r;
};
exports.circumference = function(r) {
  return 2 * PI * r;
```

[7] `http://www.npmjs.com`
[8] In Node.js, modules are JavaScript files.
[9] Example taken from https://nodejs.org/api/modules.html#loading_from_node_modules_Folders

```
};
```

The contents of `foo.js`:

```
const circle = require('./circle.js');
console.log('The area of a circle of radius 4 is ' + circle.area(4));
```

Note that the local declaration of `PI` in `circle.js` remains invisible to `foo.js`. Only the functions exported from `circle.js` are usable, and only through the new constant `circle`. Loading another version of `circle.js` – e.g. from a file called `circle-2.js` – and using it alongside the original version is possible if we assign the exports of the second version to a different constant.

```
const circle = require('./circle.js');
const circle2 = require('./circle-2.js');
console.log('The areas of a circle of radius 4 are ' + circle.area(4) + '
    ↪ and ' + circle2.area(4));
```

In addition to loading a JavaScript module by specifying the name of a file, it is also possible to load a directory as a module. Node.js will look for a directory with the required name, search for an `index.js` file inside that directory and load it.[10] Unless a specific path is given to `require` (such as `./circle.js` in the above example), Node.js will search a global load path for the required module. Additionally, it will search the `node_modules` subdirectory of directory containing the module *executing the call to require*. This is used by npm to enable scenarios such as the one shown in Figure 3.4. If the module `foo.js` depends on *A* and *B* from the graph and loads them via `require`, then npm will install the packages in the directory structure shown in Figure 3.5: The dependencies of a package are always installed into the `node_modules` subdirectory of that package.[11] Modules *A* and *B* will look for *C* in their own `node_modules` subdirectories and find the versions they need. Since, as explained above, Node.js modules cannot easily influence the global environment when they are required, using both versions of *C* inside the same program is possible as long as incompatible data from *C* are not shared between *A* and *B* through *foo*.

[10] There are a few more ways a module can be loaded, see `https://nodejs.org/api/modules.html#loading_from_node_modules_Folders`

[11] This behavior has been optimized in npm version 3 to reduce duplication of packages and prevent very deep trees. The basic idea remains the same, however.

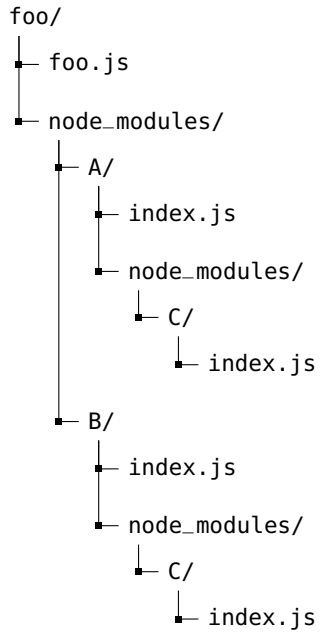


Figure 3.5.: npm directory structure

3.5 *Haskell's Cabal*

Cabal [Jon05] is the main package manager in the Haskell ecosystem and ships with the Glasgow Haskell Compiler (GHC). It uses the central Haskell package database *Hackage*[12] to search for and retrieve packages. Like Ruby's Gems, Cabal installs packages in a global location on the user's system. Unlike Gems, Cabal does not allow more than one version of a package to be installed at the same time and requires all installed packages to be compatible with one another, i.e., the set of installed package version has to be consistent and must not contain more than one version of a package.

The main reason for this restriction is that Cabal resolves the dependencies of a package when it is installed and then compiles the package against those dependencies. In a scenario such as the one shown in Figure 3.6, either `json-1.0.0` or `http-client-1.0.0` can be installed on the system, but not both. If `json-1.0.0` is installed first, then `parser-combinators` will be installed in some version greater than

[12] `https://hackage.haskell.org`

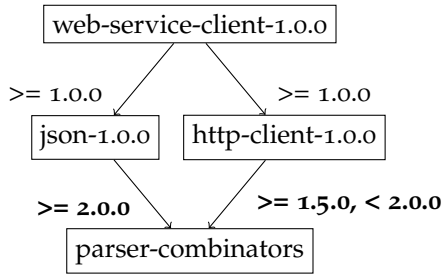


Figure 3.6.: A conflicting dependency graph

or equal to 2.0.0, e.g. 2.0.1. When the user now tries to install `http-client-1.0.0`, it cannot be installed since `parser-combinators` is installed in version 2.0.1, which is not smaller than 2.0.0 and thus incompatible to `http-client-1.0.0`'s dependency constraints. Installing `http-client-1.0.0` first results in a similar scenario. Reinstalling `parser-combinators` in a version smaller than 2.0.0 is also not an option, since that would render the already installed `json-1.0.0` unusable. If there were a version of `json` that was compatible with `parser-combinators-2.0.1` and all other installed package versions, we could reinstall `json` in that compatible version, but would then have to recompile every package that depends on the installed version of `json`. These kinds of situations are quite common since package authors on Hackage are free to choose their dependency constraints and version ranges in those constraints, quickly leading to situations in which popular packages or their dependencies are incompatible with one another. These situations are known in the community as "Cabal hell"[13].

To ease the burden of "Cabal hell", the Cabal authors have added a feature called *sandboxes*[14], which allows the user to create multiple isolated installation locations. Running `cabal sandbox init` inside a directory A will create a sandbox for this directory. All Cabal operations executed in A will use the sandbox as their package installation location. Since sandboxes are isolated from one another there is no problem with conflicts between the sandboxes. The package versions inside each sandbox must, of course, still be free of conflicts.

The *stack* build tool[15] is built on top of the Glasgow Haskell Compiler and Cabal. In addition to isolated build environments similar to Cabal's sandboxes, it aims

[13] `http://www.yesodweb.com/blog/2012/11/solving-cabal-hell`
[14] `https://www.haskell.org/cabal/users-guide/installing-packages.html#developing-with-sandboxes`
[15] `http://haskellstack.org`

to provide *reproducible builds*: projects are built using a fixed version of GHC and a curated and compatible set of popular Haskell packages, providing a consistent environment every time the project is compiled.

4

A Package Manager for Curry

As we have seen in Chapter 3, the core tasks of most package management systems are similar. There are many design decisions, however, that affect the day-to-day usage and user-friendliness of the system. Additionally, the kind of support the underlying language offers for loading files and/or modules from within a program imposes some constraints, especially on dependency management and installation locations. For example, the `require` function in Node.js does not allow a loaded module to easily impact the global environment of the loading module, enabling the use of multiple versions of one package within the same program. In Ruby and Haskell, a loaded module usually does alter the global environment, making it very hard or impossible for a package system to support loading multiple versions of a package into one program.

In this chapter, we will explain the choices made during the design of the Curry package manager (*cpm*) and the trade-offs involved in these choices as well as the restrictions imposed by the Curry language and ecosystem that influenced them. We will give a brief overview of the functionality offered by the package manager's command line interface, discuss how a Curry package is structured and distributed and how a user can find and obtain new packages. Afterwards, we will turn our attention to how packages are installed on the user's system and how dependencies can be specified and how they are resolved. Finally, we will describe how the package manager interacts with the Curry compiler and show a way for a package developer to check whether their version increments are compatible to semantic versioning. First, however, we will discuss some major design goals for the Curry package manager.

The overarching design goal we started out with was to develop a package system for Curry that works with the two major Curry compilers, KiCS2 and PAKCS, and is itself written in Curry. If possible, the system should require very few or even no support from and thus changes to the compilers themselves. It should be possible for the user to specify the dependencies of a package or project and have the package system calculate a conflict-free set of package versions that satisfies these dependencies, if such a set exists. The system should then be able to automatically acquire and install any missing dependencies, i.e., the user should not need to manually search for, acquire and then install package files. It should, however, still be *possible* for a user to manually install a package from a local file. In case of a dependency conflict, the user should be able to easily identify the source of the conflict, i.e., the package versions that are responsible for the conflict, and which dependency constraints conflict one another.

To aid the discovery and acquisition of new packages, there should be a central, searchable package index. It should be easy for a package developer to publish a new package or a new version of a package to the index. Package versioning should be done according to the semantic versioning standard discussed in Section 3.1 and the system should support package developers as much as possible in adhering to the rules established in the standard.

Early on we decided for simplicity's sake to require every consumer of a Curry package to itself be a Curry package. Thus, the term *package* can, in the context of the Curry package manager, refer not only to libraries of Curry modules that are meant to be used by other Curry modules, but also to Curry programs that only consume other packages but do not themselves offer any reusable functionality in the form of Curry modules. Node's npm has made a similar choice, while Haskell's Cabal and Ruby's Bundler allow package modules to be used outside of other packages. A downside of this approach is that a globally installed package, e.g. `xml`, cannot be used simply by starting an interactive Curry session and loading a module from the `xml` package, unless the Curry session is started from within the directory of another package with a dependency on `xml`.

4.1 The Command Line Interface

The user interacts with the Curry package manager through the `cpm` executable, which offers a command line interface. It supports multiple commands, most of

which perform operations on the *current package*. The current package is defined by the directory `cpm` is executed from: since a Curry package is simply a directory structure with a specification file called `package.json` at its root, the package manager will search for such a file in the current directory and – for convenience – in the parent directories of the current directory. If it finds a `package.json` file, it assumes the package described in it is the current package. The directory structure of a Curry package and the contents of the `package.json` file are explained further in the next section.

The following commands are supported by cpm: `install`, `uninstall`, `upgrade`, `update`, `search`, `info`, `deps`, `link`, `exec`, `curry`, `new` and `diff`.

As their names suggest, `install` and `uninstall` can be used to install and uninstall packages on the local system. While `uninstall` simply expects the name and version of the package to uninstall, e.g. `cpm uninstall xml 1.0.5`, and then removes the corresponding package version from the global package cache – see Section 4.4 for an explanation of the global package cache – `install` can be used in multiple ways. Just executing `cpm install` without any arguments will install all dependencies and transitive dependencies of the current package into the global package cache. Specifying a package name, e.g. `cpm install xml`, will install the latest version of the package from the central package index, while a distinct version can be installed by specifying it as another parameter, e.g. `cpm install xml 1.0.5`. Finally, a package can be installed from a ZIP file by specifying the name of the file in place of the package name, e.g. `cpm install xml-1.0.5.zip`.

`upgrade` can be used to install the newest compatible version of one or more of the current package's dependencies. If `upgrade` is called without further arguments, then all dependencies of the current package are upgraded to the newest available versions that form a conflict-free set. A specific package can be upgraded to the newest compatible version by passing its name as an argument to `upgrade`, e.g. `upgrade xml`. This will also upgrade all of the upgraded package's dependencies and transitive dependencies.

The `deps` and `info` commands print out information about the current package. `info` prints general information from the package metadata, such as the package's name and author as well as its dependency specifications. Using `deps`, the user can run the dependency resolution algorithm and see the package versions chosen for all dependencies and transitive dependencies, or information about a conflict that may have occurred.

`curry` allows the user to start the Curry compiler with all dependencies of the current package resolved and available to the compiler, assuming there exists a conflict-free set of packages that can satisfy all of the package's dependency constraints. Any arguments to `curry` will be passed verbatim to the Curry compiler. `exec` can be used to execute an arbitrary command with the dependencies known to any Curry compiler that might be started via that command. A sample use case is starting `currycheck` to execute tests within the package with all dependencies available.

When the user wants to replace a dependency or transitive dependency of the current package with a version somewhere on their local system, e.g. because they have fixed a bug in that dependency and the next version containing the bug fix has not been published yet, they can use `link` to declare a local directory as the source of a specific package version. Imagine that the `xml` package has a bug in its newest version 1.0.5 that is easily fixable, but the package is maintained by some third party and not by the user. The user acquires a copy of the package's source code, fixes the bug and sends a patch to the maintainers. Since the maintainers are busy, a new version containing the fix, e.g. 1.0.6, is not released immediately. In this case, the user can execute `cpm link ~/src/xml-1.0.5-fixed` to instruct the package manager to use copy of `xml-1.0.5` in the `~/src/xml-1.0.5-fixed` directory instead of the one installed on the system.

`new` offers a quick way to get started with a new package. It asks the user a few questions and then creates a bare bones directory structure and package metadata file.

The `diff` command can be used to compare the public APIs and behavior of the current package to another version of that package. It expects the version of the other package as its first argument, e.g. `cpm diff 1.0.5` to compare the current package version to version 1.0.5 of the same package. By default, the public APIs and behavior of all exported modules will be compared. The `--modules` option can be used to specify the modules to be compared, the `--api-only` and `--behavior-only` flags switch off the behavior and API comparison, respectively. More information on API and behavior comparison can be found in Section 4.7.

Using `update`, the user can keep the local copy of the central package index (see Section 4.3) up to date. `search` searches the central package index for the term passed as a parameter.

4.2 What's in a Package?

A Curry package is, at its core, a directory structure following a certain layout with a metadata file in the structure's root directory. The minimum directory structure for a package is as follows:

```
some-package/
├─ src/
└─ package.json
```

The `src` directory contains the source code of the Curry modules that make up the package, while the `package.json` file contains the metadata of the package. Curry packages are distributed and installed in source code form to reduce dependency conflicts, as explained in Sections 4.4 and 4.5. A package offering functionality for parsing and building XML files might contain the modules `XML.Types`, `XML.Parser` and `XML.Pretty` to be used by the consumers of the package, and `XML.Internal.ParserPrimitives` for internal use. The directory structure of such a package might look like this:

```
xml/
├─ src/
│  └─ XML/
│     ├─ Types.curry
│     ├─ Parser.curry
│     ├─ Pretty.curry
│     └─ Internal/
│        └─ ParserPrimitives.curry
└─ package.json
```

As the file extension suggests, the `package.json` file contains the package metadata in the JSON file format, since this format is easy to parse and widely used. The structure of the file is adapted from JavaScript's npm (see Section 3.4), which also uses JSON as its metadata file format.

A Curry package can have a variety of metadata associated with it, but only a few

fields are mandatory. Most of the metadata fields are purely informational, i.e., their content does not influence how the package system works when performing operations concerning the package. The following list explains all metadata fields briefly, with mandatory fields marked with a *.

- **Name***, `name` in the `package.json` file. The name of the package. It must only contain lowercase ASCII letters, digits, and hyphens as well as start with a letter. In particular, package names may not contain spaces. Sample package names are `json` and `http-client`. This field is mandatory.

- **Version***, `version` in the `package.json` file. The version of the package. Must follow the format specified in the semantic versioning standard (see Section 3.1), e.g. `1.5.6` for a regular version or `2.0.0-beta5` for a pre-release version. This field is mandatory.

- **Author***, `author` in the `package.json` file. The author or authors of the package. This is a free-form field that is only used for informational purposes. The suggested format is either just the names of the authors separated by commas, or the names of the authors with their email addresses added in angular brackets, e.g. `John Doe <john.doe@goldenstate.gov>`.

- **Maintainer**, `maintainer` in the `package.json` file. The current maintainers of the package, if different from the original authors. This field allows the current maintainers to indicate the best person or persons to contact about the package while recognizing the original authors. Like the author field, this is a purely informational field. The suggested format is the same as for the author field.

- **Synopsis***, `synopsis` in the `package.json` field. A short form summary of the package's purpose. It should be kept as short as possible, ideally under 100 characters. This is a purely informational field. This field is mandatory.

- **Description**, `description` in the `package.json` file. A longer form description of what the package does. This is a purely informational field.

- **License**, `license` in the `package.json` file. A license under which the package is distributed. This is a free-form field and purely informational. In case of a well-known license such as the GNU General Public License[1], the SPDX license identifier[2] should be specified. If a custom license is used, this field should be left blank in favor

[1] `https://www.gnu.org/licenses/gpl-3.0.en.html`
[2] `https://spdx.org/licenses/`

of the license file field.

- **License File**, `licenseFile` in the `package.json` file. The name of a file in the root directory of the package containing explanations regarding the license of the package or the full text of the license. The suggested file name is `LICENSE`.
- **Copyright**, `copyright` in the `package.json` file. Copyright information regarding the package. This field is free-form and purely informational. There is no suggested format.
- **Homepage**, `homepage` in the `package.json` file. The package's web site. This field should contain a valid URL.
- **Bug tracker**, `bugReports` in the `package.json` file. A place to report bugs found in the package. This is a free-form field, but the suggested formats are either a valid URL to a web based bug tracker or an email address.
- **Repository**, `repository` in the `package.json` file. The location of an SCM repository containing the package's source code. This should be a valid URL, pointing either to the repository itself – e.g. a Git URL – or to a web site explaining how to get to the repository.
- **Dependencies***, `dependencies` in the `package.json` file. The package's dependencies. The exact format is explained below. Note that this is a mandatory field and must be present even if the package has no dependencies.
- **Compiler compatibility**, `compilerCompatibility` in the `package.json` file. This field can be used to declare which compiler versions are required by the package. The format is similar to dependency declarations. If this field is missing, then the package is assumed to be compatible to all compilers in all versions.
- **Source**, `source` in the `package.json` file. This field is used to indicate where the specific version described in the current metadata set can be obtained. The exact format is described below.
- **Exported modules**, `exportedModules` in the `package.json` file. A list of modules that this package exports for use by consumers of the package. This list is used when checking for semantic versioning compliance, see Section 4.7. Note that modules not in this list are still accessible to the consumers of the package. See Section 4.6 for details.

As mentioned above, the metadata go into a JSON file called `package.json` in the package's root directory. We will now give a brief introduction to the JSON file format. The full specification can be found in IETF RFC 7159 [Bra14].

A JSON value can be either a primitive value – a string, a number, a boolean or `null` to represent a missing value – or one of two more complex structures – a list of values and a collection of key/value pairs. A JSON string consists of zero or more Unicode[3] characters enclosed in double quotes. As in many programming languages, the `\` character is used as an escape character within a JSON string. The exact rules for escaping inside of strings can be found in the JSON standard. Examples of valid JSON strings are `"Hello, World!"`, `""`, and `"This contains \"escaping\"\n"`.

A JSON number is a floating point number written in the format found in many programming languages: an optional leading minus sign followed by one or more digits, optionally followed by a decimal point and one or more digits. It is also possible to specify an exponent to the base of 10 in scientific ("`e`") notation.

Booleans in JSON are simply represented by the two literals `true` and `false`, while the missing value is represented by the literal `null`.

A list of JSON values, called an *array*, is represented by an opening bracket (`[`), zero or more JSON values separated by commas, and a closing bracket (`]`). JSON arrays can contain values of different types, i.e., a JSON array can contain a mix of strings, numbers, other JSON arrays and so on. Examples of valid JSON arrays are `[]` and `[true, "Hello, World!", 5.27e10]`.

Collections of key/value pairs are called *objects*. A JSON object starts with an opening brace (`{`) and ends with a closing brace (`}`). Zero or more key/value pairs can be placed between the braces. A key/value pair consists of a JSON string as the key, the colon (`:`) character as a delimiter, and the value for the key, which can be an arbitrary JSON value. The values inside a JSON object do not need to be of the same type. Examples of valid JSON objects are:

```
{}
```

```
{
  "Hello": "World",
  "Test": 1.27e10,
  "List": ["A", "B", "C"],
```

[3] `http://unicode.org`

```
  "Nested": {
    "Hi": "Hello"
  }
}
```

The `package.json` file contains one top-level JSON object, whose possible keys are mentioned in the list of metadata fields above. For most keys, e.g. `name`, `version` or `author`, the expected value is a JSON string, while a package's dependencies, compiler compatibility and source are specified using JSON objects and the package's exported modules are listed using a JSON array.

To specify the dependencies of a package, we add a new JSON object under the `dependencies` key of the package specification object. This new object maps the package names of our dependencies to version constraints in the form of a JSON strings, e.g. `">= 1.0.0"`. If we were developing the `web-service-client` package and needed to use the `xml-parser` package in some version greater than 1.5.0 and the `http-client` package in some version greater than 2.1.2, we could use the `package.json` file in Listing 4.1. The full format for version constraints is described in Section 4.5.

Listing 4.1: A simple `package.json` file including dependencies.

```
{
  "name": "web-service-client",
  "version": "1.0.8",
  "synopsis": "A web service client",
  "author": "John Doe",
  "dependencies": {
    "xml-parser": "> 1.5.0",
    "http-client": "> 2.1.2"
  }
}
```

Compiler constraints can be specified similarly to dependencies. The value of `compilerCompatibility` key should be a JSON object containing the names of Curry compilers as keys and version constraints as values. Each entry constrains the package to the versions of the compiler matching the version constraint. Currently, the Curry package manager knows about the KiCS2 and PAKCS compilers, referenced via the `kics2` and `pakcs` keys, respectively. If at least one compiler is specified, the package is assumed to be incompatible to any compiler that does not occur in the

`compilerCompatibility` object. If no compilers are specified, the package is assumed to be compatible to all compilers in all versions. Assuming the `web-service-client` package used features introduced in KiCS2 0.6.0 and PAKCS 1.3.0, the `package.json` file from Listing 4.1 could be extended as shown in Listing 4.2.

Listing 4.2: A `package.json` file including compiler compatibility specifications.

```
{
  "name": "web-service-client",
  "version": "1.0.8",
  "synopsis": "A web service client",
  "author": "John Doe",
  "dependencies": {
    "xml-parser": "> 1.5.0",
    "http-client": "> 2.1.2"
  },
  "compilerCompatibility": {
    "kics2": ">= 0.6.0",
    "pakcs": ">= 1.3.0"
  }
}
```

The `source` key specifies how the version of the package described in the metadata file can be obtained. When the Curry package manager is asked to install a package from the central package index, it uses the the source description to acquire the package's files. Packages without source specifications cannot be installed automatically. A package source can be a HTTP URL pointing to a ZIP file, or a Git repository URL and an optional revision to check out. We can set a HTTP source using the `http` key:

```
{
  ...
  "source": {
    "http": "http://curry-packages.org/web-service-client-1.0.8.zip"
  }
}
```

To use a Git repository, we can set the `git` key to the repository's URL. If we only set the `git` key and do not indicate a revision to use, the Curry package manager will check out the most recent revision in the repository. We can use the `tag` or `ref` keys

to specify either a Git tag or revision identifier.[4] `tag` and `ref` are mutually exclusive, i.e., we can use `tag` *or* `ref` to indicate the revision to use, but not both.

```
{
  ...
  "source": {
    "git": "git+ssh://git@github.com:john-doe/web-service-client.git",
    "tag": "version-one"
  }
}
```

The Curry package manager supports a shortcut to avoid having to change the source specification for each new package version: if the `tag` key is set to `$version`, then the system will automatically check out the the tag v*version*, where *version* is replaced by the package version. For a package in version 1.0.8, the system will check out the tag `v1.0.8`.

```
{
  ...
  "source": {
    "git": "git+ssh://git@github.com:john-doe/web-service-client.git",
    "tag": "$version"
  }
}
```

Finally, the `exportedModules` key indicates which modules are intended for use by the consumers of the package. Its value should be a JSON array containing module names as JSON strings, e.g. `["WebServiceClient.Request", "WebServiceClient.Response"]` for our example `web-service-client` package. The packages inside this list are checked by the `cpm diff` command, as explained in Section 4.7.

4.3 Finding Packages

Many modern package management systems have a centralized package index containing metadata for published packages that can be used to find new packages and acquire dependencies for existing packages. For example, the three package managers presented in Chapter 3 all have centralized indexes with web-based interfaces

[4] A Git tag is a named revision.

for searching and viewing packages: Ruby's Gems are searchable via `RubyGems.org`, npm packages can be found in the central *registry* at `npmjs.com` and Hackage provides an index of Cabal packages at `hackage.org`.

Without a centralized package index, the user has to find the packages they want to use by other means and then try to find and acquire all dependencies and transitive dependencies of these packages. A centralized package index allows the package manager to automatically find all available versions of all transitive dependencies and search for a compatible set of these package versions. Afterwards, the package manager can acquire the actual package files and install the packages on the system. Clearly, a centralized package index enables a much better user experience and aids the discoverability of packages, leading to more code reuse and less reinvention of the wheel. However, a full-fledged web application providing a browser interface for end-users and an API for the package manager on the user's system is outside the scope of this thesis.

A simpler alternative is to collect all package specification files in a fixed directory structure and distribute this directory structure to the user's systems for use as an index. CocoaPods[5], a package manager for the Apple iOS and macOS ecosystems, as well as Homebrew[6], a package manager for macOS operating system packages, use this approach. Since a specification file contains the name, version and dependencies of a package as well as a description of how to obtain the actual package contents (the `source` key), a collection of all package specifications is sufficient to resolve the dependencies of any given package and automatically acquire and install any missing packages.

One simple approach to storing package specifications in a directory structure is to create a directory for each package, containing subdirectories for all versions of the package, which in turn contain the `package.json` specification files for these versions. This is the approach used by CocoaPods and the one we adopted for the Curry package manager. An index containing specifications for `xml-1.0.0`, `xml-1.0.1` and `web-service-client-1.1.7` might look like this:

[5] `http://www.cocoapods.org`
[6] `http://www.brew.sh`

```
cpm-index/
├─ xml/
│  ├─ 1.0.0/
│  │  └─ package.json
│  └─ 1.0.1/
│     └─ package.json
└─ web-service-client/
   └─ 1.1.7/
      └─ package.json
```

To distribute this structure of package specifications to user's systems, we use the Git version control system. The index is stored in a publicly accessible Git repository at a well-known URL distributed alongside the package manager. Git supports efficiently updating a local copy of a repository from another location, transferring only the files that are different from the local copy. Since changing a package specification once it has been published, i.e., added to the index, is discouraged by semantic versioning, most files in the index will never change. Only new files will be added, and thus all that Git will transfer from the central location when the index is updated are these new files. Furthermore, since the audience for the Curry package manager are Curry developers, it is not unreasonable to expect Git to be installed on the user's system.

4.4 Installing Packages

Once the user has found a package they want to use, they have to install it on their system. Usually, they will declare a dependency in the metadata of their own package and use `cpm install` without additional arguments to automatically install the package from the central package index in the newest compatible version. In addition, the `cpm install` command supports manually installing package versions from the central package index or local ZIP files. Installing packages from local ZIP files can be used to distribute packages without having them appear in the central package index. Installing specific versions from the central package index is useful when the user wants to compare a package under development to an older version using the `cpm diff` command.

All packages installed on the system are stored in a central location, the *global package cache*. The location of the global package cache can be changed through the .cpmrc configuration file, see the user's manual included in Appendix D. The global package cache is a directory containing the installed package versions in subdirectories called `<package name>-<package version>`:

```
packages/
├─ xml-1.0.0/
├─ xml-1.0.1/
└─ web-service-client-1.1.7/
```

To enable the installation of multiple versions of one package, all packages are installed in source code form. Packages are only compiled when they are used, i.e., when a Curry program inside the package or a dependent package is compiled to be executed, and not upon installation. This allows us to install multiple versions of a package, since the installed package versions have no relation to one another until they are used. See Section 4.6 for details on the compilation process.

4.5 Resolving Dependencies

As described in Section 3.2, dependency resolution is the process of finding a consistent and complete set of package versions based on an original, not necessarily consistent set of package versions. In the terms of Section 3.2: given a set of package versions I, we want to find a consistent and complete set of package versions $R(I)$ – a solution – such that $I \subseteq R(I)$.

In the context of the Curry package manager the set I is always a singleton set containing the package version whose dependencies we want to resolve. To resolve those dependencies, we need to decide what makes up the set $\mathcal{V}$ of all package versions known to the package manager, how dependencies on other packages can be declared and with which operators, what package version to choose if multiple candidates exist, and if there are any factors other than the ones listed for the general case in Section 3.2 that can influence the dependency resolution process.

For individual version constraints, the Curry package manager supports all operators introduced in Section 3.2: $<$, $\leq$, $=$, $\geq$, and $>$. Combined version constraints –

individual version constraints combined via the boolean AND and OR operators – are also supported. The Curry package manager requires all combined version constraints to be in disjunctive normal form, i.e., they have to consist of one disjunction of arbitrarily many conjunctions. $e_1\ OR\ (e_2\ AND\ e_3)\ OR\ e_4$ is a valid expression in disjunctive normal form, while $e_1\ AND\ (e_2\ OR\ e_3)$ and $e_1\ OR\ (e_2\ AND\ (e_3\ OR\ e_4))$ are not.

Package specification files use a different textual representation for combined version constraints: the AND operator is represented by a comma (`,`) and the OR operator by two pipes (`||`). The first example from above becomes e_1 `||` $(e_2$ `,` $e_3)$ `||` e_4. Since all version constraints must be in disjunctive normal form, conjunctions can only have version constraints or other conjunctions as their children, so we can leave out the parenthesis in the above example without introducing ambiguity: e_1 `||` e_2 `,` e_3 `||` e_4. Furthermore, since the boolean AND is associative, we do not need parenthesis when a conjunction has another conjunction as its child either, e.g. e_1 `||` e_2 `,` e_3 `,` e_4 `||` e_5. A dependency specification requiring the `xml` package in any version larger than 1.5.0, but smaller than 1.6.0 and the `http-client` package in any version between 1.2.0 and 2.0.0 or from 2.1.0 onwards might look like this:

```
{
  ...
  "dependencies": {
    "xml": ">= 1.5.0, < 1.6.0",
    "http-client": ">= 1.2.0, < 2.0.0 || >= 2.1.0"
  }
}
```

In systems encouraging semantic versioning, it is common to specify constraints that restrict a package to a specific minor version. For example, the version constraint for the `xml` package in the example above restricts the package to the minor version 1.5.x. Since semantic versioning allows only bug fixes and no behavior-altering changes in patch version increments, locking a package to a minor version will allow the consumer to benefit from any defects resolved in new patch versions while reducing the chances of introducing bugs through altered behavior in those new versions. The risk can, of course, not be eliminated entirely, since semantic versioning is only a guideline that cannot be enforced completely, as we will see in Section 4.7.

As locking a dependency to a specific minor version is so common, we introduce a special operator called the *semantic versioning arrow*, or *semver arrow*, as syntactic sugar: a version constraint $\sim>$ 1.5.0 is equivalent to the dependency constraint

$\geq$ 1.5.0, $<$ 1.6.0, while $\sim>$ 1.5.5 is equivalent to $\geq$ 1.5.5, $<$ 1.6.0. That is, the version behind the semver arrow is the minimum required version and all patch versions greater than this version that are still within the same minor version are accepted as well. Similar operators are supported by Ruby's Bundler as well as Node's npm.

After discussing the kinds of dependency constraints the Curry package manager supports, we will now turn our attention to how packages are chosen based on these constraints. To resolve any dependencies, we need a set $\mathcal{V}$ of all package versions known to the dependency resolution algorithm. All packages in the central package index are potentially available to us. Even if a package version is not yet installed on the user's system, it can be acquired and installed using the source given in the package specification. Thus, we add all package versions from the central package index to the set $\mathcal{V}$. As we have seen in Section 4.4, it is also possible for the user to install a package manually from a local ZIP file. These manually installed packages are not necessarily included in the package index, so we add any packages from the global package cache that are not part of the package index to the set $\mathcal{V}$.

We now have dependency constraints to resolve and a set of package versions to operate on. Depending on the constraints given and the versions available in $\mathcal{V}$, we might run into situations where more than one package version matches the dependency constraints: assume that we are resolving the dependencies of the package `http-client-1.0.0`, which depends on `network >= 1.0.0, < 2.0.0`. Furthermore, $\mathcal{V}$ contains `network` in versions 1.0.0, 1.0.5, 1.1.0, and 2.0.3. Each of the three versions 1.0.0, 1.0.5, and 1.1.0 is a valid choice that satisfies the dependency constraints. Since the package manager has nothing left to base its decision on except the version numbers of the package, it optimistically assumes that *newer is always better* and chooses the newest compatible version, 1.1.0 in this case. Note that each choice between multiple potentially compatible versions of a package influences the set of dependency constraints that need to be satisfied, since each version of a package may specify different dependencies. In this case, `network-1.1.0` might depend on the package `ipv6`, while `network-1.0.5` might depend on `ipv4`. When the package manager chooses version 1.1.0 of `network`, it also needs to satisfy the dependency on `ipv6`. More details on how this affects the dependency resolution algorithm can be found in Section 5.3.

The *newer is always better* strategy is a good approach since, in general, newer versions of a package can be assumed to contain bug fixes and features not present in older versions. This is not true, however, when we consider pre-release versions. Assume once again that we are resolving the dependencies of `http-client-1.0.0`. This

time, though, the HTTP client depends on `network >= 1.0.0`, so any version of the network package from 1.0.0 onwards is compatible. Further assume that $\mathcal{V}$ contains `network` in versions 1.0.0, 1.1.0 and 2.0.0-beta1. *Newer is always better* would lead the package manager to choose `network-2.0.0-beta1`, since it is the newest version that satisfies the dependency constraint. Clearly, it is not safe to assume that the user intended to potentially choose a non-stable pre-release version given the dependency constraint `network >= 1.0.0`. The sensible choice for the dependency resolution algorithm would have been to choose `network-1.1.0`. To mitigate this fault of the *newer is always better* strategy, we introduce a simple change to how dependency constraints are evaluated: a dependency constraint only matches a pre-release version if a pre-release version is explicitly mentioned in one of its version constraints. If the user wanted to use the beta version of `network`, they would have to specify a dependency constraint such as `network = 2.0.0-beta1` or `network >= 2.0.0-beta1`.

After a successful dependency resolution run on a package version v with $I = \{v\}$ we obtain the set of resolved package versions, $R(I)$. Since the set $\mathcal{V}$ of available package versions is the union of the set of package versions in the package index and the set of locally installed package versions, the package versions in $R(I)$ might be from the package index, but not locally installed. In this case, all dependencies were resolved successfully, but not all packages are actually available to be used. If the action that originally led to the dependency resolution algorithm being run only requires the dependency metadata and not the contents of the actual package versions, e.g. if the user asked the Curry package manager to simply print out all dependencies and transitive dependencies of the current package, then this is not a problem. If the user invoked some action that *does* require the package contents, e.g. they asked for a compiler to be started with all dependencies available, then an error is given and the user is asked to run the `cpm install` command to automatically install the missing packages on the local system (see Section 4.1 for a brief overview of all commands available).

We can use the set of resolved package versions $R(I)$ to compile v, as described in the next section. The next time we want to compile v, we re-run the dependency resolution algorithm since v's dependencies might have changed in the meantime. If new versions of some packages have become available in the meantime, then *newer is always better* might result in different versions being chosen for some dependency. Revisiting the `http-client` example from above, on the first dependency resolution run `network` might be available in versions 1.0.0 and 1.0.3, so the algorithm chooses 1.0.3. On the next run, there might be a new version of `network` available, e.g 1.1.0, which is then chosen over 1.0.3 and added to $R(I)$. Based on this new version of

```
http-client/
└─ .cpm/
   └─ package_cache/
      ├─ det-parse-1.0.0 -> /global/package/cache/det-parse-1.0.0
      └─ xml-1.0.5 -> /global/package/cache/xml-1.0.5
```

Figure 4.1.: A sample local package cache

$R(I)$, the user is either asked to run `cpm install` to install `network-1.1.0` or any of its dependencies that might be missing on the local system, or, if all packages are already available on the system, `network-1.1.0` is used instead of `network-1.0.3` without further notice.

Both of these behaviors are problematic from a user experience standpoint: in the first situation, the user is asked to install new packages even though they did not change any dependencies, which is unexpected and confusing. In the second situation, a new version of some dependency is used without the user having done or changed anything, which might lead to unexpected changes in behavior if the dependency has changed significantly. Instead of choosing the newest version of any dependency on each dependency run, we only want to choose the newest versions when doing a fresh installation of all dependencies, or when the user explicitly asks us to upgrade one or more packages via the `cpm upgrade` command. In all other cases, the resolution algorithm should choose the same package versions as in the last install or update run, as long as they still satisfy all dependency constraints.

In order to remember which package versions were chosen in the last dependency resolution run and prefer these versions the next time, we introduce a *local package cache*. The local package cache is stored in the `.cpm/package_cache` directory in the root of the package whose dependencies are being resolved. It contains symbolic links to package versions in the global package cache, which are created when a package's dependencies are installed via `cpm install` or upgraded via `cpm upgrade`. The local package cache for a package `http-client` dependent on `det-parse` and `xml` might look like the tree shown in Figure 4.1. We now modify the *newer is always better* strategy to also search the package versions in the local package cache and prefer those versions to the ones from other sources. If versions 1.1.0 and 1.0.5 of a package are in the local package cache and additional versions 1.0.6 and 1.2.5 exist in the package index, the modified strategy will choose version 1.1.0 instead of 1.2.5. This

way, when a new dependency resolution run is triggered, the versions from the local package cache will be preferred as long as they are still compatible. When the user wants to upgrade all dependencies via `cpm upgrade`, we clear the local package cache to effectively forget our former choices before running the dependency resolution algorithm, which will then choose the newest compatible versions of all packages.

The user can also use `cpm upgrade` to upgrade a single package p to the newest compatible version. In this case, we have two options: we can clear all entries for p from the local package cache, or clear all entries for p and its transitive dependencies. In the former case, the dependency resolution algorithm will choose the newest available version of p that is compatible to the versions of its transitive dependencies that remain in the local package cache, which might restrict its choices. In the latter case, the algorithm is not limited to those versions of p's transitive dependencies that were chosen in a previous dependency resolution run. We choose the second approach for the Curry package manager, since we assume that when a user explicitly asks to upgrade a single package they want the newest possible version, even if it means upgrading the dependencies of that package.

The local package cache is also used to implement the `link` command explained in Section 4.1: if the local package cache contains a symbolic link to some other location than the global package cache, then the package is automatically used from that location if it is chosen during dependency resolution.

As we have seen in Section 4.2, package specifications also contain a field for the user to declare which compiler versions the package is compatible to. Compiler version dependencies are declared under the key `compilerCompatibility` using the same dependency constraint syntax used for package dependencies. During dependency resolution, we check if the current compiler version satisfies one of the compiler constraints. If so, we can use the package provided its dependency constraints are satisfied. If not, the package is marked incompatible.

4.6 Interacting with the Compiler

To compile the Curry files inside a package we need to resolve the dependencies and transitive dependencies of that package and then make the contents of these dependencies available to the compiler. As described in Section 2.3, there are two main Curry compilers that we want to support: KiCS2 and PAKCS.

Both KiCS2 and PAKCS have a setting for the paths that they will search for when they encounter an `import` statement in a Curry file that is to be compiled. When a module called `ParserCombinators` is imported via `import ParserCombinators`, the Curry compiler will search all directories in its *paths* setting for the file `ParserCombinators.curry`. If a nested module such as `XML.Schema.Parser` is imported, the Curry compiler will search all directories in *paths* for a directory called `XML` containing a directory called `Schema`, in turn containing a file called `Parser.curry`. Since all Curry packages have their source code inside the `src` directory by convention, we need to add the `src` directories of all dependencies to the compiler's path list. Luckily, both compilers will consult the `CURRYPATH` environment variable for additional paths to search whenever they are started, so all we have to do for the compiler to discover the source files of our packages is to set this environment variable to the list of `src` directories separated by colon characters.

Since we add the `src` directories of all dependencies to the Curry compiler's search path, all modules inside of these directories will be available to any module being compiled. There is no way to restrict the set of available modules to the ones in the package metadata `exportedModules` field. Since one of our design goals was to avoid modifying how the existing Curry compilers work if at all possible and this approach allows us to do that, we accept the disadvantage of not being able to hide internal modules as a trade-off. A possible approach to making modules hideable with modifications to the Curry compilers is discussed in Chapter 7 on future work.

When a Curry compiler compiles a Curry file, it stores intermediate results as well as any resulting binaries in the `.curry` directory next to the source files. If we have a directory `parser-combinators` containing the `ParserCombinators.curry` file and we start a Curry compiler in the `parser-combinators` directory and load the `ParserCombinators` module from the corresponding file, a `.curry` directory will be created in the `parser-combinators` directory. For nested modules, the `.curry` directory will be created at the start of the hierarchy. If the compiler is asked to load the module `XML.Schema.Parser` and it finds this module in a directory called `xml-package` – either because this directory is in the compiler's paths setting or because it is the current directory – then the `.curry` directory will be created in the `xml-package` directory and not in the `xml-package/XML/Schema` directory. In our scenario, the `.curry` directories will thus be created in the `src` directories of the package being compiled and its dependencies.

The concrete information that is stored in the `.curry` directories is dependent on the compiler being used. What is important, however, is that the generated files for a

compiled module m may be reused by the compiler when it is next asked to compile m to avoid repeating expensive computations. Additionally, the generated files may be specific to the exact versions of any modules imported by m. A single package version p with a set of dependency constraints $Deps(p)$ may be depended upon by arbitrarily many package versions p_i with $i \in \mathbb{N}$, each with its own set of dependency constraints $Deps(p_i)$.[7] When the dependency resolution algorithm is run on two such package versions p_k and p_j with $j, k \in \mathbb{N}$ and $j \neq k$, then the dependency constraints in $Deps(p_k)$ and $Deps(p_j)$ may cause the concrete versions of the packages in $DepPkgs(p)$ to differ in the resolution result sets $R(\{p_k\})$ and $R(\{p_j\})$. This means that we cannot, in general, reuse the files generated by the Curry compiler for p when compiling p_k during the compilation of p_j, since p may have been compiled with different versions of its dependencies during the compilation of p_k than the ones calculated for the compilation of p_j.

To avoid reusing any intermediate files between different packages, we copy the contents of each package version in the resolution result set from the local package cache (see Section 4.5 on dependency resolution for an explanation of the local package cache) to the *runtime package cache*. Like the local package cache, the runtime package cache is a subdirectory of the `.cpm` directory in the current package's root directory. Note that unlike the local package cache, the runtime package cache contains actual copies of the packages, and not just symbolic links to another location. The paths added to the `CURRYPATH` environment variable all point to the copies in the runtime package cache, so the Curry compiler will read the imported modules from there and also create its `.curry` directories and intermediate files inside the the copies in the runtime package cache. While this approach has the upside of being very simple and effective – there is no danger of ever reusing any intermediate results created with different dependency versions if we always create fresh copies before compiling any code – it has the distinctive downside of potentially repeating a lot of work that could be avoided by reusing the compiler's intermediate results when possible. One potential approach to safely reuse intermediate results in this scenario is given by Dolstra, Löh, and Pierron [DLP10] and discussed in more detail in Chapter 7 on future work.

[7] In practice, the number of packages will, of course, be finite.

4.7 Enforcing Semantic Versioning

As explained in Section 3.1, semantic versioning aims to be a cross-ecosystem standard for the format of version numbers and, more importantly, for encoding semantic meaning into those version numbers, or rather into the difference between two version numbers. Early on, we decided to adopt semantic versioning for the Curry package manager, following the example of both Ruby's Gems[8] and Node's npm[9]. We adopted the *semantic versioning arrow* (see Section 4.5) from Ruby's Bundler to help the user specify version constraints that take advantage of semantic versioning to minimize the chances of introducing breaking changes when upgrading a package to a newer version. What is still missing is a way for package developers to gain confidence that they are not violating semantic versioning when releasing new versions of their packages, i.e., that no changes to the previous version are introduced that are incompatible with the change in version number.

To this end, we developed the `diff` command briefly mentioned in Section 4.1. The `diff` command compares two versions of a package and tries to determine what has changed between them and whether these changes are compatible with the change in version numbers. Semantic versioning requires that only backwards-compatible changes be introduced in a patch version, i.e., if only the last part of the version number changes, then only bug fixes are allowed. Increasing the minor version allows the developer to introduce new functionality, as long as it is backwards-compatible, i.e., no existing behavior must be impacted by the new functionality. Backwards-incompatible changes can be introduced in new major versions, i.e., functionality can be removed or changed at will. No guarantees are made regarding compatibility to any other major versions. Since semantic versioning applies only to public APIs, `diff` only compares those modules listed in the `exportedModules` field in the package specification and only the functions and data types exported from those modules. Internal functions and modules are not compared.

The Curry package manager compares two aspects of all exported modules: the types of exported functions and data types, i.e., the API of the package, and the actual behavior of the exported functions. There are various tools for comparing the public APIs of libraries for other languages, e.g. *JDiff*[10] for Java or *hackage-diff*[11]

[8] `http://guides.rubygems.org/patterns/`
[9] `https://docs.npmjs.com/getting-started/semantic-versioning`
[10] `http://javadiff.sourceforge.net`
[11] `https://hackage.haskell.org/package/hackage-diff`

Change in API	Required version increase
Module added	Minor
Module removed	Major
Function added	Minor
Function removed	Major
Function type changed	Major
Data type added	Minor
Data type removed	Major
Data type changed	Major

Table 4.2.: Changes in API and semantic versioning

for Haskell. Elm[12], a functional programming language that compiles to JavaScript, ships with a package manager that supports comparing two API versions and reporting semantic versioning offenses[13]. Comparing the behavior of two packages seems to be less common. Leslie-Hurd [Les13] explores comparing versions of verified packages via automated proof to automatically compute version constraints. We were, however, unable to find an existing package manager that incorporates this or any other form of behavior comparison between two package versions.

To compare the APIs of two package versions, we first examine the list of exported modules of both package versions from their respective package specifications. From here on, we define A to refer to the older or lower version of the package and B to refer to the newer or higher version. Any modules listed in one package but not the other can obviously not be compared, since there is nothing to compare them to in the other version. If a new module was added in version B, i.e., the newer version, then the version difference between A and B must be at least a minor version for the change to be allowed under semantic versioning, since new functionality was introduced. If a module was removed in B that was present in A, then B must be a new major version of the package, since existing functionality was removed or changed in a backwards-incompatible way (it may have moved to another module, for example).

Once we have the list of exported modules that are present in both A and B, we read in each module in both versions in AbstractCurry format (see Section 2.2) and

[12] `http://elm-lang.org`
[13] `https://github.com/elm-lang/elm-package#publishing-updates`

compare the list of exported functions and data types in those two modules. Once again, if a function or data type was added in version B, then new functionality was introduced and B has to be at least a minor version above A. If a function or data type was removed in version B, then existing functionality was changed and B has to be a new major version. For functions or data types that are present in both versions of a module, we compare their structure. If the type of a function has changed, then this is a backwards incompatible change and B has to be a new major version. If the type of a data type or any of its constructors has changed, then this is a backwards incompatible change as well and the same rules apply. A summary of the different API changes that can be found and what version changes they require is given in Table 4.2.

We can use the information from the API comparison process to alert package developers to a large class of semantic versioning violations. Comparing the API of two package versions will not, however, find changes in the implementations of those functions whose types have remained the same. To find some of these implementation changes, we compare the behavior of both versions using CurryCheck [Han16]. CurryCheck is a tool distributed with both the KiCS2 and PAKCS compilers that allows the programmer to specify unit tests and property tests. Unit tests are implemented similarly to other languages: the programmer writes a function that uses comparison operators provided by the testing library to check if some property of the code under test holds. Property tests are similar, but they are *parameterized*, i.e., each test function can take one or more arguments and CurryCheck will call these test functions multiple times with different values for these arguments. A property test checking whether 0 is the additive identity on Curry's `Int` type might look like this:

```
test_zeroIsAdditiveIdentity :: Int -> Test.EasyCheck.Prop
test_zeroIsAdditiveIdentity x = x <~> x + 0
```

`<~>` is a comparison operator provided by CurryCheck that checks whether the set of values of both of its arguments are equal. More such operators can be found in the KiCS2 manual [Han+16b]. This test can be executed by calling CurryCheck with the name of the module containing the test. CurryCheck will then find all test functions in the module and execute them, generating different values for the `Int` parameter. The idea behind property tests is to check the attribute described by the test function for many different values in the hopes of discovering edge cases for which it does not hold and thus revealing bugs in the implementation.

We can use property tests to compare two versions of a function and check whether

their implementations are equivalent: for each such function, we generate a property test parameterized over the argument types of the function. Two functions `addInts_1 :: Int -> Int -> Int` and `addInts_2 :: Int -> Int -> Int` could, for example, be compared via the following property test:

```
test_addInts :: Int -> Int -> Test.EasyCheck.Prop
test_addInts a b = addInts_1 a b <~> addInts_2 a b
```

A test such as this is, of course, no proof that both implementations are equivalent. CurryCheck generates a limited number of parameters and there is always a chance that the implementations show different behavior for values that are not among those parameters. However, it is still useful to gain some confidence in the equivalence of both implementations. To prevent confusion, we will call the property tested by test functions such as the one above *limited equivalence* from now on.

Sadly, there are some limitations that prevent us from checking any arbitrary two functions for limited equivalence using CurryCheck:

- Functions that perform infinite computations cannot be checked for limited equivalence, since a property test comparing two versions of such a function will not terminate. Assume we have two versions of a function `ones` that generate infinite lists of ones. A test comparing these functions will look like this:

  ```
  test_ones :: Test.EasyCheck.Prop
  test_ones = ones_1 <~> ones_2
  ```

  Neither version of `ones` will terminate and thus `test_ones` will not terminate either. We could, of course, only take the first 100 values of each version, e.g. `take 100 ones_1 <~> take 100 ones_2`, which will terminate thanks to Curry's laziness. Since we want to generate the test functions automatically, however, and have no way of determining automatically which part of an infinite data structure such as the one generated by `ones` is sufficient for establishing limited equivalence with reasonable certainty, we cannot, in general, compare functions such as `ones`. In fact, we require the programmer to explicitly mark such functions as *do not compare* using pragmas in the source code to avoid generating tests for these functions, since we have no way of automatically determining which functions will perform infinite computations:

  ```
  {#- NOCOMPARE -#}
  ones :: [Int]
  ```

- Functions that take other functions as parameters, e.g. `map :: (a -> b) -> [a] -> [b]` cannot be tested, since CurryCheck does not have a generator for function types as of this writing.
- Functions that take `Float` parameters cannot be tested, since CurryCheck does not have a generator for floating point values as of this writing.
- CurryCheck only supports parameterized tests up to five arguments, limiting us to five arguments as well. Any functions taking more than five arguments cannot be checked automatically.
- Functions are only tested for their behavior when used as functions, not when used as relations via free variables.

  Outside of these limitations, however, the `diff` command can be used by package maintainers to gain more confidence that they have not inadvertently introduced any breaking changes in what was meant to be a patch level release. The implementation of API and behavior comparison, including some more problems that we encountered and their solutions, can be found in Sections 5.4 and 5.5.

5

Implementation

In this chapter, we will describe the concrete implementation of the Curry package manager. The package manager consists of various Curry modules that are used by the main module `CPM.Main`, which is compiled to the `cpm` executable mentioned in Chapter 4. We will now give a brief overview of the modules that make up the Curry package manager before going into detail on some of them in the next sections.

A hierarchical overview of all modules is shown in Figure 5.1. As mentioned above, the `CPM.Main` module contains the `main` function that becomes the entry point for the `cpm` executable. The other modules are concerned with:

- `CPM.AbstractCurry` provides helper functions that deal with reading, writing and mutating AbstractCurry files.
- `CPM.Config` contains default values for configuration options and functions to read the `.cpmrc` configuration file.
- `CPM.Diff.API` provides the API comparison functionality mentioned in Section 4.7.
- `CPM.Diff.Behavior` provides the behavior comparison functionality described in Section 4.7.
- `CPM.Diff.CurryComments` can read comments from Curry programs that belong to functions and data type definitions. It is adapted from the CurryDoc [Han02] tool.
- `CPM.Diff.Rename` renames (prefixes) modules in a package and its dependencies on the AbstractCurry level.

- `CPM.ErrorLogger` provides combinators for chaining IO operations that can fail and output log messages.
- `CPM.FileUtil` contains utility functions for dealing with files and directories.
- `CPM.LookupSet` provides the `LookupSet` data type, a set of packages used as candidates during dependency resolution.
- `CPM.Main` is the main entry point.
- `CPM.Package` contains data types for package specifications and functions for reading and writing specifications from and to JSON files.
- `CPM.PackageCache.Global` provides functions for managing packages in the global package cache, i.e., globally installed packages.
- `CPM.PackageCache.Local` contains functions that manage the local package cache described in Section 4.5.
- `CPM.PackageCache.Runtime` provides functions for the runtime package cache described in Section 4.6.
- `CPM.PackageCopy` contains functions that operate on a copy of a package, i.e., a package directory.
- `CPM.Repository` provides data types and functions for dealing with the central package index and its local copy.
- `CPM.Resolution` contains the dependency resolution algorithm.

We also use some modules that we have extracted into their own Curry packages, since they are useful in other contexts. These packages are described in Appendix B.

5.1 The Main Module

The `CPM.Main` module contains the `main` function, which is the entry point of the `cpm` executable. On startup, we first try to parse the command line arguments using the `OptParse` module described in Section B.3. If successful, we check if a number of applications are present on the user's system, e.g. `curl`, `git`, or the UNIX `cp` utility. We use these applications throughout the Curry package manager to fetch ZIP files

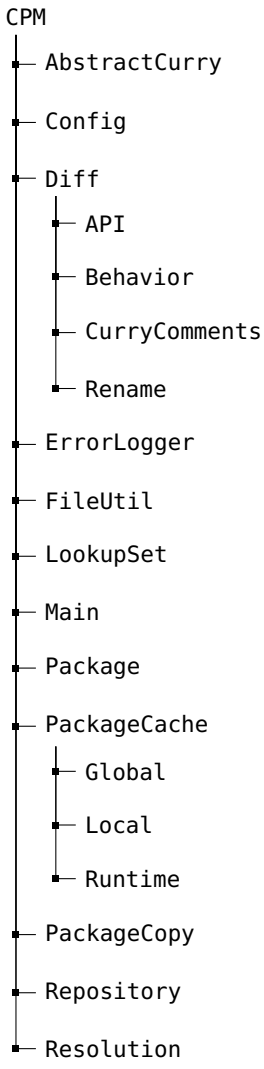


Figure 5.1.: The modules that make up the Curry package manager

from URLs, check out Git repositories, copy binary files and so on. Next, we try to read the `.cpmrc` configuration file from the user's home directory, which can be used to overwrite some settings such as the path where the global package cache is kept. We then try to read the package specifications from the central package index and the global package cache and dispatch to the function that handles the subcommand specified on the command line.

The command line is parsed into an `Options` value, a record data type with the two fields `optCommand` and `optLogLevel`:

```
data Options = Options
  { optLogLevel :: LogLevel
  , optCommand  :: Command }
```

`LogLevel` is defined by `CPM.ErrorLogger` and contains all possible log levels. `ErrorLogger` contains combinators for chaining IO operations and supports logging messages in between operations. The value of this field is used as the minimum level required for a log message to be printed, the default being `Info`. `Command` is a data type representing all commands known to the package manager:

```
data Command
  = Deps
  | NoCommand
  | Install   InstallOptions
  | Uninstall UninstallOptions
  | PkgInfo   InfoOptions
  | Compiler
  | Update
  | Search    SearchOptions
  | Upgrade   UpgradeOptions
  | Link      LinkOptions
  | Exec      ExecOptions
  | Diff      DiffOptions
  | New
```

Some commands, e.g. `Install` and `Exec`, take their own option data types as arguments. For example, the options for `Install` consist of an optional target and version as well as a flag specifying whether to install pre-release versions if available:

```
data InstallOptions = InstallOptions
```

```
  { instTarget     :: Maybe String
  , instVersion    :: Maybe Version
  , instPrerelease :: Bool }
```

The command line arguments to the application are parsed into these data structures via the `OptParse` module. `OptParse` provides functions to build a declarative model of the command line argument structure that is to be parsed. Much like Curry's built-in `GetOpt` module, `OptParse` supports arbitrary values as parse results. We use this flexibility to return functions as parse results that modify the above data types. For example, the following excerpt specifies the options of the `install` command:

```
[...]
  (   command "install" (help "Install a package.") (\a -> Right $ a {
      ↪ optCommand = Install (installOpts a) })
        (   arg (\s a -> Right $ a { optCommand = Install (installOpts a)
            ↪  { instTarget = Just s } })
              (  metavar "TARGET"
              <> help "A package name or the path to a file"
              <> optional)
        <.> arg (\s a -> readVersion' s >.> \v -> a { optCommand =
            ↪ Install (installOpts a) { instVersion = Just v } })
              (  metavar "VERSION"
              <> help "The package version"
              <> optional)
        <.> flag (\a -> Right $ a { optCommand = Install (installOpts a)
            ↪ { instPrerelease = True } })
              (  short "p"
              <> long "pre"
              <> help "Try pre-release versions when searching for newest
                  ↪  version.") )
[...]
```

Note that each function creates a modified version of a value of type `Options`, only changing what is necessary using Curry's record update syntax. The result type of each function is an `Either`, since some arguments, such as the package version in the example, may need to be parsed further which may result in errors. `OptParse` returns a list of parse results, one for each command, flag, and option. In our case, this will be a list of functions which we fold onto an initial, default `Option` value to obtain a value that represents all command line options.

5.2 *Packages and Dependencies*

In this section, we will present Curry data types for the most essential concepts introduced in Chapters 3 and 4, such as versions, packages and dependencies. All data types presented here are defined in the `CPM.Package` module.

First, we define a version to be a four-tuple, as seen in Section 3.1. The first, second and third components are the major, minor and patch versions, respectively. The fourth component is the optional pre-release specifier:

```
type Version = (Int, Int, Int, Maybe String)
```

The `CPM.Package` module also offers functions for rendering versions to strings and parsing strings into versions (using the `det-parse` package described in Appendix B).

Next, we use `Version` to define a data type for version constraints. Based on this data type, we define a conjunction as a list of version constraints and a disjunction as a list of conjunctions. Modeling disjunctions as lists of conjunctions is sufficient to represent dependency constraints found in package specifications since we only allow constraints in disjunctive normal form. As for versions, the module also contains functions for rendering and parsing version constraints, conjunctions and disjunctions.

```
data VersionConstraint = VExact      Version
                       | VGt         Version
                       | VLt         Version
                       | VGte        Version
                       | VLte        Version
                       | VCompatible Version

type Conjunction = [VersionConstraint]

type Disjunction = [Conjunction]
```

The `VCompatible` constructor represents the semantic versioning arrow discussed in Section 4.5. A dependency consists of the name of a package and a disjunction as a constraint on the version of that package. Similarly, a compiler compatibility constraint consists of the name of the compiler and a disjunction:

```
data Dependency = Dependency String Disjunction

data CompilerCompatibility = CompilerCompatibility String Disjunction
```

The `PackageSource` type represents the different sources that can be specified in a package description, namely a HTTP URL to a ZIP file of the package contents or a Git URL with an optional revision specifier:

```
data PackageSource = Http String
                   | Git String (Maybe GitRevision)
                   | FileSource String -- for internal use

data GitRevision = Tag String
                 | Ref String
                 | VersionAsTag
```

Finally, we define a record type to represent packages. Each field of the record corresponds to one of the possible metadata fields described in Section 4.2, with optional fields modeled as `Maybe` types where necessary.

```
data Package = Package {
    name                 :: String
  , version              :: Version
  , author               :: String
  , maintainer           :: Maybe String
  , synopsis             :: String
  , description          :: Maybe String
  , license              :: Maybe String
  , licenseFile          :: Maybe String
  , copyright            :: Maybe String
  , homepage             :: Maybe String
  , bugReports           :: Maybe String
  , repository           :: Maybe String
  , dependencies         :: [Dependency]
  , compilerCompatibility :: [CompilerCompatibility]
  , source               :: Maybe PackageSource
  , exportedModules      :: [String]
  }
```

5.3 Dependency Resolution

As we have seen in Sections 3.2 and 4.5, the goal of dependency resolution is to find a consistent and complete set of package versions $R(I)$ from an initial set I of package versions. In the context of the Curry package manager we always resolve the dependencies of a single package, so I is a singleton set. We already discussed some aspects of how we want to resolve dependencies in Section 4.5. In short, we have to make sure that all dependency constraints are satisfied, that all compiler compatibility constraints are satisfied and that we choose the newest compatible version of each dependency (*newer is always better*). In this section, we describe the algorithm that the Curry package manager uses to resolve dependencies and how it is implemented.

The problem of dependency resolution is a variant of the classic *constraint satisfaction problem* (CSP). Russell and Norvig [RN03] define the constraint satisfaction problem by a set of variables $X_1, ..., X_n$ with corresponding domains of possible values $D_1, ..., D_n$ as well as a set of constraints $C_1, ..., C_m$. A constraint C_i specifies allowable values for some subset of the variables. An assignment of values to variables is called a *state* and states are called *consistent* if they satisfy all constraints and *complete* if they assign a value to every variable. A state that is both consistent and complete is called a *solution* to the constraint satisfaction problem. Note that this terminology is equivalent to the definitions established in Chapter 3.

A first attempt at mapping the dependency resolution problem to the constraint satisfaction problem might look like this: interpret each package that is a transitive dependency of the initial package version as a variable. The domain of each such variable will be the set of available versions of the package it represents and the set of constraints will be all dependency constraints of the initial package version as well as its transitive dependencies. Sadly, this definition is flawed: since dependency constraints can be different for each version of a package, both the set of variables and the set of constraints might change when some variable is assigned a value – that is, when a specific version of some package is chosen. However, Nordin and Tolmach [NT01] present functional versions of classic algorithms for solving CSPs which can be adapted from statically known sets of variables and constraints to the dynamic nature of the dependency resolution problem. The implementation of the algorithm presented below is such an adaption of the examples given by Nordin and Tolmach.

A Simple Implementation

A naive approach to finding a set of package versions that is a solution is to enumerate all possible sets of package versions and look for one that satisfies all dependency and compiler constraints of all contained package versions. We can represent all *candidate sets* or *states* for this approach in a tree where each node is labeled with a state. The state of the root node is the singleton set containing the initial package version. For a given node n with state s, we choose an arbitrary package p from $\cup_{v \in s}\{x \in DepPkgs(v) \mid x \notin \{PkgOf(v') \mid v' \in s\}\}$, i.e., the set of packages depended upon by the package versions in s of which s does not yet contain any version. If such a p exists, we define the set of child states of s to be $\{s \cup \{v\} \mid v \in \mathcal{V}$ with $PkgOf(v) = p\}$. If no such p exists, we define the set of child states of s to be the empty set. A Curry program that generates such a tree might look like this:

```
data Tree a = Node a [Tree a]

simpleCandidateTree :: Package -> [Package] -> Tree [Package]
simpleCandidateTree initial allVersions =
  Node initial (buildTree [initial] (dependencies initial))
 where
  versionsOf p = filter ((== p) . name) allVersions
  buildTree pkgs ((Dependency p _):ds) =
    if p `elem` (map name pkgs)
      then buildTree pkgs ds -- skip packages already in tree
      else map (nodeForVersion pkgs ds) (versionsOf p)
  nodeForVersion pkgs deps v =
    Node (v:pkgs) (buildTree (v:pkgs) (dependencies v ++ deps))
```

A tree for the set of package versions $\mathcal{V}$ and set of dependencies $\mathcal{D}$ from the example in Figure 5.2 is given in Figure 5.3, with consistent states set in *italic* and states that are solutions – both consistent and complete – set in **bold**. For brevity, each node is labeled only with the package version that was added to the state of its parent node.

Assuming Curry functions `isConsistent :: [Package] -> Bool` and `isComplete :: [Package] -> Bool`, which will check a state for consistency and completeness, respectively, we can easily find a solution in a candidate tree if one exists:

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Node a cs) = Node (f a) (map (mapTree f) cs)
```

$$\begin{aligned}\mathcal{V} = \{&\texttt{ws-client-1.0.0},\\ &\texttt{http-client-1.0.0}, \texttt{http-client-1.0.5},\\ &\texttt{network-2.0.7}, \texttt{network-2.1.3},\\ &\texttt{xml-parser-1.0.0}, \texttt{xml-parser-1.2.0},\\ &\texttt{det-parse-0.5.7}, \texttt{det-parse-0.7.3}\}\end{aligned}$$

$$\begin{aligned}\mathcal{D} = \{&\texttt{ws-client-1.0.0} \Rightarrow \texttt{http-client} = \texttt{1.0.5},\\ &\texttt{ws-client-1.0.0} \Rightarrow \texttt{xml-parser} < \texttt{1.2.0},\\ &\texttt{http-client-1.0.0} \Rightarrow \texttt{network} \geq \texttt{2.0.0},\\ &\texttt{http-client-1.0.5} \Rightarrow \texttt{network} \sim> \texttt{2.1.0},\\ &\texttt{xml-parser-1.0.0} \Rightarrow \texttt{det-parse} \sim> \texttt{0.5.0},\\ &\texttt{xml-parser-1.2.0} \Rightarrow \texttt{det-parse} \geq \texttt{0.6.0}\}\end{aligned}$$

Figure 5.2.: An example resolution problem

```
leaves :: Tree a -> [a]
leaves (Node a []) = [a]
leaves (Node _ cs) = concatMap leaves cs

solution :: Package -> [Package] -> Maybe [Package]
solution = find isConsistent . filter isComplete . leaves
  . candidateTree
```

Note that even for a small example such as this with only two versions of each package and four dependencies in total, the tree grows rapidly. As the number of dependencies and available versions of each dependency grows, it quickly becomes infeasible to check every leaf of the tree – every complete state – for consistency. Luckily, if a state is inconsistent, then every descendant state will be inconsistent as well, since a package version is never updated or removed in subsequent states once it has been added to a state.

We can use this fact to our advantage by checking every state in the tree for inconsistency and removing all inconsistent states and their children, which will remove large subtrees, lowering the number of leaves in the tree that we have to check. A pruned version of the tree in Figure 5.3 is given in Figure 5.4. Assuming a Curry

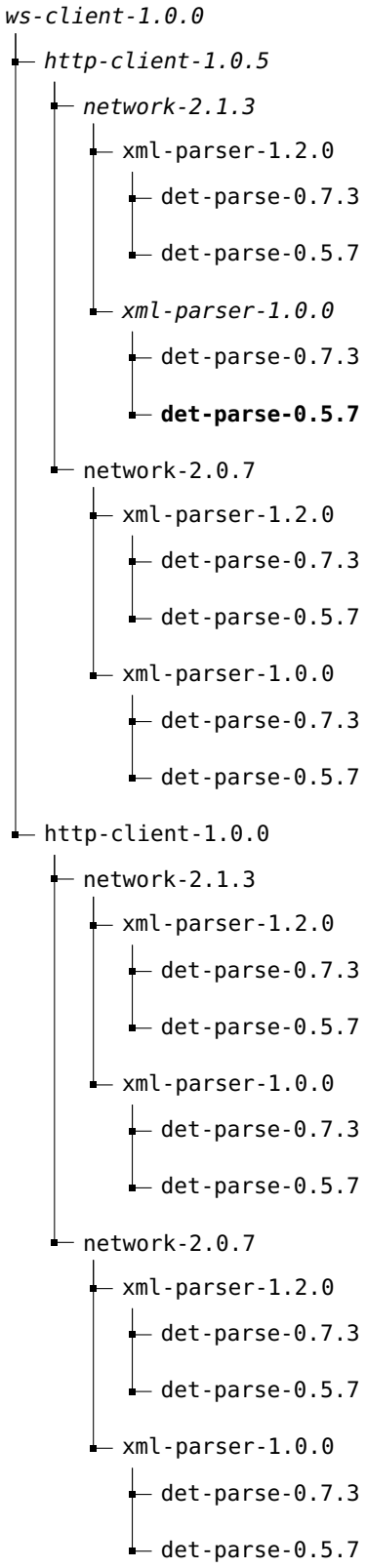


Figure 5.3.: A candidate tree

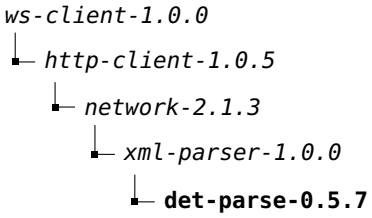


Figure 5.4.: A pruned candidate tree

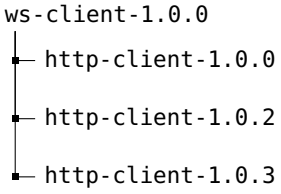


Figure 5.5.: Primary conflicts leading to a missing package version failure.

function `prune :: (a -> Bool) -> Tree a -> Tree a` that will remove all nodes and their descendants that match the predicate from a tree, we can build an optimized version of `solution`:

```
labelInconsistent :: [Package] -> ([Package], Bool)
labelInconsistent ps = (ps, not $ isConsistent ps)

solution :: Package -> [Package] -> Maybe [Package]
solution = find isComplete . map fst . leaves . prune snd
  . mapTree labelInconsistent . candidateTree
```

This approach is a functional generate-and-test variant of the classic backtracking algorithm for solving constraint satisfaction problems. Since Curry uses lazy evaluation, i.e., it evaluates expressions only when their results are actually needed, the above version of `solution` will never generate the subtrees of those nodes removed by the call to `prune`. Laziness lets us specify a compact and readable version of the algorithm in terms of operations on a tree of all possible states and still retain the performance gains of backtracking.

There are still a few things missing from the above implementation: we have not yet discussed where compiler compatibility is checked, the algorithm might not always choose the newest compatible version of each package and in case no solution can be found, the output of `solution` is `Nothing`, giving no hint as to what caused the failure. The check for compiler compatibility can be implemented by extending `isConsistent` to also check whether the current compiler is compatible with each package version in the state.

Giving Good Feedback

Giving good feedback about why the search for a solution failed is a bit more involved. There a three failure constellations that we want to treat and report separately:

1. **Compiler compatibility.** A package that is required by a dependency constraint is not compatible to the current compiler.
2. **Missing package versions.** There is a single dependency constraint that no available package version is able to meet.

3. **Conflicting dependency constraints.** Two dependency constraints for a package exist and no available package version is able to meet both.

For a compiler compatibility failure, we want to inform the user which package version was found to be incompatible and why that package version was considered in the first place, i.e., which package has a dependency on the incompatible package and with which constraints. A missing package version should be reported similarly: which package is missing, why it is needed and which versions would be compatible. To resolve a conflict between two dependency constraints, the user needs to know which package is affected, which two dependency constraints are conflicting and to which two packages these constraints belong.

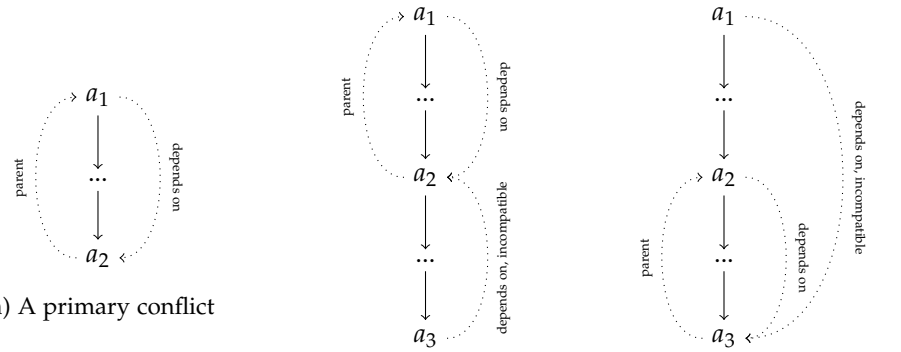
Currently, each node's state is simply a list of active package versions. To be able to present information about a conflict to the user, we change this to a list of *activations*. Each activation consists of the package version that was activated, the dependency constraint that led to it being activated and the activation of the package version containing that dependency constraint. The activation of the initial package version simply contains that package version. We define $actPkg(a)$ to be the package version of some activation a, $actDep(a)$ to be the dependency constraint on $actPkg(a)$ that led to the activation and $actParent(a)$ to be the parent activation. $actDep(a)$ and $actParent(a)$ are undefined if a is an initial activation. Thus, an activation lets us trace back which package versions and dependencies led to a package being added to a state, all the way to the initial package version. We define a data type for activations and rewrite the `candidateTree` function to generate a tree of `States` instead of lists of packages. Each state consists of a list of all activations up to the current node as well as a reference to the activation added at the current node for convenience.

```
data Activation = InitialA Package
                | ChildA Package Dependency Activation

actPackage :: Activation -> Package
actPackage (InitialA pkg) = pkg
actPackage (ChildA pkg _ _) = pkg

type State = (Activation, [Activation])

candidateTree :: Package -> [Package] -> Tree Activation
candidateTree initial allVersions = let a = InitialA initial in
  Node a (buildTree [a] $ zip (repeat a) (dependencies initial))
```



(a) A primary conflict

(b) A secondary conflict (1)

(c) A secondary conflict (2)

Figure 5.6.: Different dependency conflict constellations

```
  where
   versionsOf p = filter ((== p) . name) allVersions
   buildTree acts ((depAct, d@(Dependency p _)):ds) =
     if p `elem` (map (name . actPackage) acts)
       then buildTree acts ds -- skip packages already in tree
       else map (nodeForVersion acts depAct d ds) (versionsOf p)
   nodeForVersion acts depAct dep deps v =
     let
       a' = ChildA v dep depAct
       acts' = a':acts
       nextDeps = zip (repeat a') (dependencies v) ++ deps
     in
       Node acts' (buildTree acts' nextDeps)
```

Now that our states have more information than just the currently active package versions, we can label the tree with more than just a boolean value when we find an inconsistency. When we encounter an inconsistent state, we differentiate the three cases mentioned above: if the state is inconsistent because some package version is incompatible to the current compiler, we label the node with a *compiler incompatibility conflict*. Three different constellations can lead to a state being inconsistent because of an unmet dependency constraint, shown in Figure 5.6. In the first case, Figure 5.6a, a new activation a_2 conflicts the dependency constraint of its parent activation a_1 and no other active package version has a dependency constraint on a_2's package. This constellation appears when a_1's package is the first package to depend on a_2's

package in the current branch of the candidate tree. We call this kind of conflict a *primary conflict*. Note that if all child states of a node are inconsistent because of primary conflicts, we have a *missing package version* failure in that branch of the trees, since no compatible version of the package in question could be found. An example of this is shown in Figure 5.5, where `http-client-1.0.5` is not present but required by `ws-client-1.0.0` (see Figure 5.2).

In the second case, Figure 5.6b, the package version of a new activation a_3 has a dependency constraint on a package p that was activated in an earlier activation a_2, but the version that was activated is incompatible to a_3's dependency constraint. We call this constellation a *secondary conflict*. A version of p was activated in a_2 because the package version of a_2's parent activation a_1 depends on p. Note that a_2's package version may or may not be compatible to the dependency constraint from a_1's package version. If it is not, then the state that introduced a_2 will be inconsistent with a primary conflict. Figure 5.6c shows another constellation for a secondary conflict. The new package $actPkg(a_3)$ is depended upon both by its parent activation a_2 and by some earlier activation a_1. If $actPkg(a_3)$ is compatible to $actPkg(a_2)$'s dependency constraint, but not to $actPkg(a_1)$'s dependency constraint, then we have a secondary conflict. Otherwise, if $actPkg(a_3)$ is also incompatible to $actPkg(a_2)$'s dependency constraint, we have a primary conflict. A secondary conflict means that resolution failed because of the third failure condition, conflicting dependency constraints.

We define a Curry data type for the three kinds of conflicts and redefine `labelInconsistent` to label each state with either one of these conflicts, or `Nothing` in case the state is consistent. Note that we check the dependencies of the state in reverse order, from oldest to newest. Otherwise, secondary conflicts could mask primary conflicts if the current activation's package version is incompatible to some previous activation but also has a dependency constraint that conflicts an earlier dependency constraint.

```
data Conflict = CompilerConflict Activation
              -- the activation of the incompatible package
              | PrimaryConflict Activation
              -- the activation containing the original activation of the
              -- package and the activation of the incompatible
              -- dependency constraint
              | SecondaryConflict Activation Activation

stDependencies :: State -> [(Activation, Dependency)]
```

```
stDependencies (_, acts) = concatMap zippedDeps acts
 where
  zippedDeps a = zip (repeat a) (dependencies $ actPackage a)

labelInconsistent :: State -> (State, Maybe Conflict)
labelInconsistent s = (s, firstConflict s (reverse $ stDependencies s))

firstConflict :: State -> [(Activation, Dependency)] -> Maybe Conflict
firstConflict _ [] = Nothing
firstConflict s@(act, acts) ((depAct, d@(Dependency p disj)):ds) =
  if not $ isCompatibleToCompiler (actPackage act)
    then Just $ CompilerConflict act
    else case findPkg of
      Nothing -> firstConflict s ds
      Just  a -> if isDisjunctionCompatible (version $ actPackage a) disj
        then firstConflict s ds
        else if actParent a == depAct
          then Just $ PrimaryConflict act
          else Just $ SecondaryConflict a depAct
 where
  findPkg = -- finds previous activation of p

solution :: Tree State -> Maybe [Package]
solution = find isComplete . map fst . leaves
  . prune ((/= Nothing) . snd) . mapTree labelInconsistent
```

If we encounter a labeled candidate tree that contains no solution states, we have to decide which of the inconsistent states to report to the user as the cause of the failure that they need to investigate, since a candidate tree will usually contain many inconsistent states, such as primary conflicts for every version of a package that has been tried unsuccessfully. First, we disregard any inconsistent state that is a descendant of another inconsistent state. That is, we only consider the first inconsistent state on a path down the labeled tree, since as far as the search for solutions is concerned, states further down the path are unreachable.

This pruned tree might still contain many inconsistent states. Since one of our design goals was to provide clear and helpful information to the user in case of a resolution failure, we do not want to report every single inconsistent state. Instead, we try to find the most relevant conflict and report that. The most relevant incon-

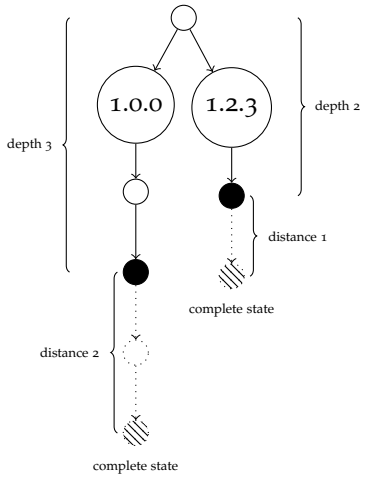


Figure 5.7.: Depth and level of completion in a tree

sistent states are the ones where the resolution algorithm came closest to finding a solution. Sadly, we cannot simply compare the lengths of the paths leading up to an inconsistent state, i.e., the depth of the state in the tree, as a measure for closeness to a solution: since different package versions chosen along different paths may depend on different packages and in particular on a different *number* of packages, a consistent state of depth five may be closer to a solution than a consistent state of depth ten, depending on the package versions chosen along the paths to those states. Instead, we use the number of packages that are missing for a state to become complete as our distance measure. We call this number the *level of completion* of a state. Figure 5.7 shows a tree where the conflict node – shown in black – is deeper in the left branch, but closer to a complete state in the right branch.

Once we have found the inconsistent states closest to a solution, we group them by their parent nodes. Note that all siblings of any state in the set of inconsistent states closest to a solution must be members of the set as well. If any sibling of such a state s were not inconsistent, then inconsistent states would exist that were closer to a solution than s. We then choose a single conflict from each group of siblings to represent that group: if the group consists solely of states labeled with primary conflicts, we simply choose the first of those. The information we would present to the user is the same for any of the states. The same is true for any group consisting only of states labeled with secondary or compiler conflicts. In mixed groups, secondary conflicts are usually the most interesting ones and give the most

information about what went wrong to the user: they contain the same information as primary conflicts, namely which dependency constraint could not be satisfied, as well as hinting at a possible way to resolve the conflict by adjusting the version of the package with the conflicting dependency constraint. Additionally, if some package version exists that is compatible to both dependency constraints from the secondary conflict, but was not chosen because it is incompatible to the current compiler, it is reasonable to expect the user to notice the overlap between the two version ranges and investigate why no package in that overlap was chosen. Thus, we always choose the first state labeled with a secondary conflict if any exists. If the mixed group only contains compiler conflicts and primary conflicts, we choose the first compiler conflict, since that will give the user most information: which package was required by which dependency constraint and that at least one of the package versions is not compatible to the current compiler.

We choose one state from the resulting set of inconsistent states to present to the user by the same rules.

Choosing the Newest Version

Now that we can give good feedback in case of a failure, we turn our attention to finding the solution containing the newest versions of each package. Note that `solution` searches the list of states returned by `leaves` for a complete state:

```
solution :: Tree State -> Maybe [Package]
solution = find isComplete . map fst . leaves
  . prune ((/= Nothing) . snd) . mapTree labelInconsistent

leaves :: Tree a -> [a]
leaves (Node a []) = [a]
leaves (Node _ cs) = concatMap leaves cs
```

As we can see, `leaves` returns all leaves of a tree in left-to-right order. If we generate the candidate tree with the versions of each package listed in the preferred order, then the states containing those preferred versions will be checked for completeness first. Recall that in Section 4.5 we discussed a slightly modified variant of the *newer is always better* strategy that prefers package versions from the local package cache to avoid choosing newer package versions on subsequent dependency resolution runs unless the user explicitly asks to do so. This strategy is implemented by the

`CPM.LookupSet` module. A `LookupSet` is a collection of package versions and their sources, i.e., the package index, the global package cache and the local package cache:

```
data LookupSource = FromRepository
                  | FromLocalCache
                  | FromGlobalCache

type PkgMap = TableRBT String [(LookupSource, Package)]

data LookupOptions = LookupOptions
  { ignoreLocalVersions :: [String] }

data LookupSet = LookupSet PkgMap LookupOptions

findAllVersions :: LookupSet -> String -> [Package]
findAllVersions = -- implementation omitted
```

Since the most common operation on a lookup set is `findAllVersions`, which retrieves a list of all versions of a specific package, we keep the package-source-pairs in a `TableRBT` – a table based on red-black-trees – indexed by package name. The `TableRBT` type is provided by a module of the same name as part of Curry's standard library. `findAllVersions` returns the versions of a package in the order laid out above: first, all versions from the local package cache, sorted from newest to oldest, then all versions from the global package cache and the package index, also sorted from newest to oldest. In an upgrade scenario, we want to prevent the versions in the local package cache from being ordered before those from the other sources for the packages being upgraded and their transitive dependencies. `ignoreLocalVersions` can be used to set a list of packages whose locally cached versions should not be preferred.

If we modify `candidateTree` to take a `LookupSet` instead of a list of package versions and use `findAllVersions` on that lookup set, the nodes of the tree will be ordered the way we want them to be. Building a lookup set from the package index, global package cache and local package cache and setting which versions, if any, should not be preferred from the local cache is left up to the consumers of the `CPM.Resolution` module.

The Public API

To hide the complexity of candidate trees and conflicts from the consumers of the `Resolution` module, the public API consists of only a few data types and methods:

```
module CPM.Resolution (ResolutionResult, showResult, resolutionSuccess
  , resolvedPackages, showDependencies, showConflict, resolve) where

data ResolutionResult = ResolutionSuccess Package [Package]
                      | ResolutionFailure (Tree ConflictState)

resolve :: Package -> LookupSet -> ResolutionResult

resolutionSuccess :: ResolutionResult -> Bool

resolvedPackages :: ResolutionResult -> [Package]

showResult :: ResolutionResult -> String

showDependencies :: ResolutionResult -> String

showConflict :: ResolutionResult -> String
```

A `ResolutionResult` is the result of `resolve`, which runs the algorithm discussed in this section. Note that only the type is exported, not its constructors. The consumers of the package have to use the methods provided to work with a result. `resolutionSuccess` will check if a resolution run was successful, `resolvedPackages` can be used to extract the list of resolved packages from a successful resolution result. `showDependencies` renders a tree of the resolved transitive dependencies from a successful package resolution while `showConflict` renders a human-readable representation of the different types of conflicts discussed above. `showResult` calls `showDependencies` for successful resolution results and `showConfict` for unsuccessful ones. An example of the output of both functions is given in Chapter 6 on evaluation.

5.4 Comparing APIs

As described in Section 4.7, the Curry package manager can compare the public APIs of two packages, i.e., exported functions and data types from the modules listed as exported in the package specification. This functionality is mostly implemented in the module `CPM.Diff.API`. To compare two versions of a package, we first create copies of those packages in a temporary directory, since we need to run the Curry compiler's frontend on them (see Section 4.6 on why we create copies of packages before compiling them). We then resolve all dependencies of each package version and finally call `compareApiModule` for each module from the combined list of exported modules (in case a module is exported in one package version, but not the other):

```
compareApiModule :: (Package, String, [Package])
                 -> (Package, String, [Package])
                 -> String -> IO (ErrorLogger Differences)
compareApiModule (pkgA, dirA, depsA) (pkgB, dirB, depsB) mod =
  if mod `elem` exportedModules pkgA
    then if mod `elem` exportedModules pkgB
      then readAbstractCurryFromPath dirA depsA mod >>= succeedIO |>=
        \p1 -> readAbstractCurryFromPath dirB depsB mod >>= succeedIO |>=
        \p2 -> let
          funcDiffs = diffFuncsFiltered funcIsPublic p1 p2
          typeDiffs = diffTypesFiltered typeIsPublic p1 p2
          opDiffs   = diffOpsFiltered   (\_ _ -> True) p1 p2 in
        succeedIO $ (Nothing, funcDiffs, typeDiffs, opDiffs)
      else succeedIO $ (Just $ Addition mod, [], [], [])
    else succeedIO $ (Just $ Removal mod, [], [], [])
```

The result type of `compareApiModule` is `Differences` wrapped in an `IO ErrorLogger`, which is a type synonym for a four-tuple of `Difference` values of different types:

```
type Differences = (Maybe (Difference String), [Difference CFuncDecl]
                   , [Difference CTypeDecl], [Difference COpDecl])

data Difference a = Addition a
                  | Removal a
                  | Change a a
```

The first component is a single optional `Difference String`, which is used to represent an added or removed module. If the first component is present, then the other components will be empty lists, since there is nothing to compare if there is only one version of a module. For modules that are present in both versions of the package, we read their AbstractCurry versions via `readAbstractCurryFromPath` from `CPM.AbstractCurry`.

`CPM.AbstractCurry` uses the AbstractCurry modules from Curry's standard library to generate, read, and write AbstractCurry files for the modules inside a package. Curry's frontend is used to generate AbstractCurry files from Curry source files. To generate an AbstractCurry representation for a Curry module, the frontend needs access to imported modules, which may be contained in one of the package's dependencies. For this reason, `CPM.AbstractCurry` makes all dependencies available to the frontend when generating AbstractCurry files. Additionally, `CPM.AbstractCurry` provides functionality to read modules from any dependency, not just from the package itself, and to apply a transformation function to any of those modules.

Once we have both versions of a module, we use `diffFuncsFiltered`, `diffTypesFiltered` and `diffOpsFiltered` to calculate the difference in functions, types and operators defined in both modules. A function, type or operator is marked as an *addition* if it is present in package B, but not in package A. Conversely, it is marked as a *removal* if it is present in package A, but not in package B. If a function is present in both versions of the package, we compare its arity and type expression. The function is marked as *changed* if they are different. For types that are present in both package versions, we compare their type variables and constructors. If any are different, they are also marked as *changed*. For operators, we compare fixity and precedence.

As the `filtered` suffix in the above function names suggests, all entities to be compared from both modules are filtered by a predicate. For types and functions, we check whether they are marked as public, i.e., exported from the module (via the `typeIsPublic` and `funcIsPublic` predicates, respectively). Operators do not have any visibility on the AbstractCurry level, so we compare all of them. Since only public functions and types are in the list of candidate functions from both versions of the module, any change in visibility will show up as an addition or removal in the list of differences, not as a change, which is in line with semantic versioning: if a function was public but is no longer, it has effectively been removed from the public API of the package.

Once we have obtained a list of `Differences` for the various modules in both package versions, we use `showDifferences` to format them for the user and annotate those changes that are not allowed by semantic versioning for the jump between the two versions. An example comparison is shown in Chapter 6.

5.5 *Comparing Program Behavior*

In Section 4.7, we gave a rough overview of how the behavior of two package versions can be compared. In this section, we will outline the process in a bit more detail and discuss some of the problems that can occur and how to work around them.

The functionality for comparing package behavior is largely implemented in the `CPM.Diff.Behavior` and `CPM.Diff.Rename` modules. Our goal is to generate a Curry program that contains a series of CurryCheck property tests which compare two versions of a function. An example test for a function called `sayHello` might look like this:

```
test_sayHello :: String -> Test.EasyCheck.Prop
test_sayHello x = sayHello_1 x <~> sayHello_2 x
```

Note that we have assumed two versions of the `sayHello` function: `sayHello_1` and `sayHello_2`. When comparing two package versions, the functions we want to compare will have the same names in both packages, however. Furthermore, the modules that contain these functions will also have the same name, posing the first challenge: how do we import two versions of a module with the same name?

The only way to import two versions of a module is to rename one or both of them. If we assume that `sayHello` is contained in the `Greetings` module of the `greetings` package, and that we want to compare versions 1.0.0 and 1.0.1 of that package, we can rename the `Greetings` module in version 1.0.0 to `V_1_0_0_Greetings` and to `V_1_0_1_Greetings` in version 1.0.1. We can then use qualified imports to reference both versions of the `sayHello` function in our test program:

```
import qualified V_1_0_0_Greetings
import qualified V_1_0_1_Greetings

test_sayHello :: String -> Test.EasyCheck.Prop
test_sayHello x = V_1_0_0_Greetings.sayHello x
```

```
<~> V_1_0_1_Greetings.sayHello x
```

Since both package versions can have different dependency constraints for the same dependencies, it is not enough to just rename the modules inside the two packages to be compared. Assume, for example, that `greetings-1.0.0` depends on package `name-parser` in version $= 2.0.0$, while `greetings-1.0.1` requires `name-parser` in version $\geq 2.0.5$, since an important bug was fixed in this version. If `name-parser` exports the module `NameParser`, then both versions of `Greetings` will include an `import NameParser` statement. We need to rename `NameParser` in both versions *and* replace all references to `NameParser` in `Greetings` with the correct name, i.e., `Greetings` in version 1.0.0 must refer to `V_2_0_0_NameParser` while version 1.0.1 must refer to `V_2_0_5_NameParser`. `name-parser` might, of course, have dependencies of its own that we have to rename and then change the references to. In short, we have to rename all modules in all transitive dependencies and change all references to other modules in those modules. Renaming, or rather prefixing, packages and dependencies like this is implemented in `CPM.Diff.Rename`.

The `prefixPackageAndDeps` function in `CPM.Diff.Rename` takes the directory of a package which should be prefixed, the string to prefix module names with, and a directory where the renamed modules should be stored. `prefixPackageAndDeps` will write the modules themselves to the destination directory in the correct directory structure, but not the package specification files. Instead, the destination directory can be added to the `CURRYPATH` directly and the modules from the original package as well as any dependencies will be available to the compiler.

Once we have obtained renamed versions of both packages, we can generate test programs using the `AbstractCurry.Build` module from Curry's standard library. We will still run into problems, however, if a tested function takes a parameter of a type that is defined in a module of the original package or one of its dependencies. Imagine that `Greetings` contains a data type `Name`:

```
-- First name and last name.
data Name = Name String String
```

Additionally, instead of taking a `String` parameter, the `sayHello` function now takes a parameter of type `Name`:

```
sayHello :: Name -> String
sayHello (Name first last) = "Hello, " ++ first ++ " " ++ last
```

Which version of the `Name` type should we use to parameterize our test function? If we choose `V_1_0_0_Greetings.Name`, then we cannot pass the generated values to `V_1_0_1_Greetings.sayHello` and vice versa, since both versions expect the type from their own module. Since comparing the behavior of two versions of a function only makes sense if both functions are of the same type, we only compare functions if all types referenced by those functions are unchanged between both versions. If both versions of a type are exactly the same except for their names, however, we can easily generate a function that will translate from one version to the other and use that in the property test:

```
tt_Name :: V_1_0_0_Greetings.Name -> V_1_0_1_Greetings.Name
tt_Name (V_1_0_0_Greetings.Name a b) = V_1_0_1_Greetings.Name a b

test_sayHello :: V_1_0_0_Greetings.Name -> Test.EasyCheck.Prop
test_sayHello x = V_1_0_0_Greetings.sayHello x
              <~> V_1_0_1_Greetings.sayHello (tt_Name x)
```

The same technique can be used to translate nested types. Another function `greetPeople :: [Name] -> String` might, for example, generate a greeting for every person in the list and concatenate them. If we want to compare these functions, we need to translate both the list of `Name` values and the `Name` values themselves:

```
tt_0 :: V_1_0_0_Greetings.Name -> V_1_0_1_Greetings.Name
tt_0 = (V_1_0_0_Greetings.Name a b) = V_1_0_1_Greetings.Name a b

tt_1 :: [V_1_0_0_Greetings.Name] -> [V_1_0_1_Greetings.Name]
tt_1 [] = []
tt_1 (x:xs) = (tt_0 x) : (tt_1 xs)

test_greetPeople :: [V_1_0_0_Greetings.Name] -> Test.EasyCheck.Prop
test_greetPeople x = V_1_0_0_Greetings.greetPeople x
                 <~> V_1_0_1_Greetings.greetPeople (tt_1 x)
```

When we generalize what we have described in the example above, we arrive at the following steps to generate a CurryCheck program that will compare all exported functions from two versions of a package:

1. **Rename all modules and dependencies.** First, we prefix every module from each of the two package versions and every one of its transitive dependencies with the version number of the package. We copy all renamed files to two temporary directories,

one for each package version.

2. **Find functions to compare.** We read the modules in AbstractCurry form to obtain a list of all functions and their types. To exclude those functions whose types have changed between the two versions of the module, we use the functionality from `CPM.Diff.API` to compare the module's public APIs. We then filter the resulting list further to exclude all non-public functions, functions that are marked with the `NOCOMPARE` pragma discussed in Section 4.7, functions that have argument or result types whose definitions have changed, as well as functions that take more than five parameters or that take a parameter of functional or `Float` type.

3. **Generate translator functions.** For each of the functions to be compared, we collect all argument and return types that are not built-in, i.e., not defined in Curry's standard library, and generate a type translator function for each one. If a type in turn references other non-built-in types, we recursively generate translator functions for those types as well and use them in the translator function for the original type.

4. **Generate comparison functions.** We generate one comparison test for each function that is to be compared. In these functions, we use the type translator functions generated in the step before on parameters and return types.

5. **Write module and call CurryCheck.** Finally, we write the generated module to a temporary location and execute CurryCheck on this module, with all renamed modules on the `CURRYPATH`. We present the output of CurryCheck to the user.

   In the `CPM.Diff.Behavior` module, the first step is performed by `preparePackageDirs`, which takes source directories of both modules as its arguments. It copies both packages to a temporary directory in their original form. It then uses `copyAndPrefixPackages` from `CPM.Diff.Rename` to prefix all modules in the package and its transitive dependencies and copy them to a temporary directory. Finally, it returns a `ComparisonInfo` value, which contains both package specifications, the locations of the original and renamed modules, the prefixes that the modules were renamed with, and maps from original to renamed module names for both versions.

   Based on this `ComparisonInfo`, the `diffBehavior` function will execute the rest of the steps. Note that some parameters and return values such as the central package index or a cache for AbstractCurry files have been omitted from the listings below for brevity. `|>=`, `|>`, and `succeedIO` are combinators from `CPM.ErrorLogger` that chain IO operations.

```
diffBehavior :: ComparisonInfo -> [String] -> IO (ErrorLogger ())
```

```
diffBehavior info mods = getBaseTemp |>=
  \baseTmp -> findFunctionsToCompare (infSourceDirA info) (infSourceDirB
      ↪ info) |>=
  \funcs ->
    let
      filteredFuncs = filter ((`elem` mods) . fst . funcName) funcs
    in case funcs of
      [] -> succeedIO ()
      _  -> genCurryCheckProgram filteredFuncs info |>
        putStrLn infoText >> putStrLn "" >> succeedIO () |>
        callCurryCheck info baseTmp filteredFuncs
```

Here, we first find the temporary directory and use `findFunctionsToCompare` to read all functions in both modules and find the functions that we can compare. We then further filter that list to only include functions from the modules we are supposed to compare. If no functions are left, we do nothing. Otherwise, we use `genCurryCheckProgram` to generate the comparison program, output a message informing the user of what is about to happen, and then use `callCurryCheck` to execute the tests in the comparison program.

The main work is done in `genCurryCheckProgram`, which in turn calls `genTestFunction` and `genTranslatorFunction` to generate test and translator functions, respectively. First, it calls `genTranslatorFunction` for every type that needs to be translated from one package version to the other. `genTranslatorFunction` will generate a translator from version A of the type to version B of the type. The relevant types are all argument and return types of all functions that contain or are themselves a type inside a module that was renamed.

```
genCurryCheckProgram funcs info baseTmp =
  foldEL genTranslatorFunction emptyTrans translateTypes |>=
  \transMap ->
    let
      testFunctions  = map (genTestFunction info transMap) funcs
      transFunctions = transFuncs transMap
      allFunctions   = testFunctions ++ transFunctions
      prog = CurryProg "Compare" imports [] allFunctions
    in
      writeFile (baseTmp </> "Compare.curry") (showCProg prog) >>
      succeedIO ()
```

```
 where
  translateTypes = filter (needToTranslatePart info) allReferencedTypes
  allReferencedTypes =
    (concatMap (argTypes . funcType) funcs) ++
    map (resultType . funcType) funcs
  needToTranslatePart info (CTVar _) = False
  needToTranslatePart info (CFuncType e1 e2) =
    needToTranslatePart info e1 || needToTranslatePart info e2
  needToTranslatePart info (CTCons n es) =
    isMappedType info n || any (needToTranslatePart info) es
  isMappedType info (mod, _) = isJust $ lookup mod (infModMapA info)
```

In the listing above, `info` refers to a `ComparisonInfo` value and `infModMapA` accesses its map of translated module names for one of the versions. Note that since we only compare functions whose types are identical in both versions, the map of translated module names for either version of the the package will contain all modules that occur in the types of those functions. `genTranslatorFunction` takes a `TransMap`, a data type that contains a map from type expressions to functions that can translate those type expressions, as well as the generated translator functions themselves, and returns a modified `TransMap` with additional translator functions. Since a translator function added to the `TransMap` by `genTranslatorFunction` might be needed in the translation of another type further down the list, we use `foldEL` to thread one `TransMap` through all invocations of `genTranslatorFunction`.

`genTranslatorFunction` itself takes a `ComparisonInfo`, the current `TransMap` as well as the type expression to generate a translator for as its arguments:

```
genTranslatorFunction info tm t@(CTCons (mod, n) te) =
```

Note that we pattern match against `CTCons`, which is the AbstractCurry type for a constructor application. A type expression can also be a type variable or a function type. Function types cannot occur, since we do not compare any functions that take functions as arguments because of a limitation in CurryCheck. Type variables cannot occur either, since those are not deemed translation-worthy by `genCurryCheckProgram` (see `needToTranslatePart` above) because they will be instantiated to `Bool` when generating the test functions, as we will see below.

The first thing `genTranslatorFunction` does is to check whether there already is a translator function for the requested type. If so, it simply returns the `TransMap` it

was passed. Otherwise, it loads the type declaration for the data type referenced in the constructor application from its module:

```
if isJust $ findTrans tm t' then succeedIO tm else findType |>=
```

`t'` is the type expression `t` with all type variables instantiated to Bool. Since we generate our test functions for Bool-instantiated types, we also generate our translator functions for Bool-instantiated types. If a translator function does not already exist and we succeeded in finding the current constructor's type declaration, we next instantiate the type declaration with the type from the type expression using the `instantiate` function:

```
\typeDecl -> succeedIO (instantiate typeDecl t') |>=
```

Since the type declaration of a polymorphic type and specifically its constructors will contain type variables in their type expressions and we are looking to generate a translator function for a fully instantiated version of this type – those type variables that were not already instantiated in the original type taken as a parameter by the function to be compared were instantiated to `Bool` instead – we build a version of the type declaration with all type variables instantiated to the ones from `t'`. This instantiated type declaration might contain types that also need to be translated by a translator function: in our introductory example, we saw one function, `greetPeople`, which took a list of values of type `Name`. We would call `genTranslatorFunction` on the type expression `[Name]` and get the instantiated type declaration for `[Name]`. The type `Name` inside the list needs to be translated from one version to the other. So we find the parts of our instantiated type declaration that need to be translated and recursively generate translator functions for them. First, we add an entry for our own, yet to be generated translator function to the `TransMap`, though, to avoid running into an infinite loop if we are dealing with a recursive data type.

```
\instTypeDecl -> succeedIO (addEntry tm t') |>=
\(tm', name) -> foldEL (genTranslatorFunction info) tm' (transExprs
    ↪ instTypeDecl) |>=
```

Note that the `addEntry` function returns a tuple of the new `TransMap` and a name that it has automatically assigned for our new function. Next, we calculate the type of our new translator function:

```
\tm'' ->
  let
    aType = prefixMappedTypes (infPrefixA info) t'
```

```
bType = prefixMappedTypes (infPrefixB info) t'
fType = CFuncType aType bType
```

`prefixMappedTypes` is a function that will prefix all constructor names in a given type expression that are defined in a module that has been renamed with the given prefix. We define two functions, one for package *A* and one for package *B*, that will prefix a module name if it is one of the modules that has been renamed:

```
mapIfNeeded modMap m = if isMappedType info (m, "")
  then fromJust $ lookup m modMap
  else m
mapIfNeededA = mapIfNeeded (infModMapA info)
mapIfNeededB = mapIfNeeded (infModMapB info)
```

Next, we define a function that takes a constructor and builds a rule for the translation function from it. The rule will match version *A* of the constructor and build a new value using version *B* of the constructor:

```
ruleForCons (CCons (m, cn) _ es) = simpleRule [pattern] call
 where
  pattern = CPComb (mapIfNeededA m, cn) (pVars (length es))
  call = applyF (mapIfNeededB m, cn) $ map transformer vars
  vars = zip (take (length es) [0..]) es
```

`simpleRule`, `pVars` and `applyF` are from Curry's built-in `AbstractCurry.Build` module. `pVars` generates pattern variables starting from `x0`, `applyF` creates a function application, and `simpleRule` creates a function rule from a list of patterns and a body expression. We build a pattern that matches on the constructor name in version *A*, using `mapIfNeededA` from above to translate the module name in case versions *A* and *B* are different. Our function body is a call to version *B* of the constructor, applied to the output of the `transformer` function function mapped over a list of tuples of the constructor's argument type expressions alongside the variable numbers assigned to these arguments in the pattern.

The `transformer` function tries to find a translator for the type of the argument and returns an expression that will call this translator function on the corresponding variable in the pattern. If there is no translator function, it simply returns an expression for the variable:

```
varX i = CVar (i, "x" ++ (show i))
```

```
    transformer (i, CTVar _) = varX i
    transformer (i, CFuncType _ _) = varX i
    transformer (i, e@(CTCons _ _)) = case findTrans tm'' e of
      Nothing -> varX i
      Just tn -> applyF ("Compare", tn) [varX i]
```

We now have everything we need to generate our translator function:

```
    rules = map ruleForCons cs
  in
    succeedIO $ addFunc tm'' $ cfunc fName 1 Public fType rules
```

Note that the full version of `genTranslatorFunction` is a bit more complex since it also supports record constructors and type synonyms as well as types declared via `newtype`.

Now that we have seen how the type translator functions are generated, we turn our attention to `genTestFunction`, which is a bit simpler than `genTranslatorFunction`. Recall that the goal of `genTestFunction` is to generate a CurryCheck property test that checks whether both versions of the function under test behave similarly. To this end, we want to parameterize the CurryCheck test on the argument types of the function being tested. `genTestFunction` takes a `ComparisonInfo`, a `TransMap` containing translator functions, and the function to generate a test for. First, we define the name of the test function, calculate the names of the *A* and *B* versions of the module and instantiate the function type with `Bool` as explained above:

```
genTestFunction info tm f =
  let
    (mod, localName) = funcName f
    testName = "test_" ++ underscorifyFuncName (funcName f)
    modA = infPrefixA info ++ "_" ++ mod
    modB = infPrefixB info ++ "_" ++ mod
    instantiatedFunc = instantiateBool $ funcType f
```

`underscorifyFuncName` takes a function name and replaces all module separators, i.e., dots, by underscores. Next, we generate the type for the test function via `genTestFuncType`, which will replace the return type of the function with `Test.EasyCheck.Prop`, the return type of CurryCheck tests. It will also rename all module names that occur in the function's argument types to version *A*, if necessary, since we want to generate our values in version *A* to be able to apply the translator functions

generated before. Additionally, we generate pattern variables for the arguments of the new test function, which are exactly as many arguments as the function under test takes.

```
fType = genTestFuncType f
vars = pVars (realArity f)
```

We are now ready to generate the calls to versions A and B of the function that is being tested. For the call to version A, we do not need to translate the test function's parameters, since CurryCheck will generate them in version A. We might have to translate version A's return value, however, if it is of a type that needs to be translated, since version B's return value will be in version B and we have only generated translators from A to B but not the other way around. So we generate arguments in version A, but compare results in version B. We define the function `returnTransform` which is either the identity function if the return type does not need to be translated, or a function that will generate a call to the appropriate translator function. We then use `returnTransform` to define `callA`:

```
returnTransform = case findTrans tm (resultType instantiatedFunc) of
  Nothing -> id
  Just tr -> \t -> applyF (modName, tr) [t]
cvars = map (\(CPVar v) -> CVar v) vars
callA = returnTransform $ applyF (modA, localName) cvars
```

For the call to version B, we need to translate every argument of a type from a renamed module. We use a function called `transformedVar` for this, which is similar to the `transformer` function from above. We do not need to translate the return value, since it will already be in version B:

```
args = argTypes instantiatedFunc
callB = applyF (mod, localName) $ map transformedVar $ zip args vars
```

Finally, we use our definitions to generate the test function:

```
in
  cfunc (modName, testName) (realArity f) Private newType [
    simpleRule vars (applyF ("Test.EasyCheck", "<~>") [
      callA, callB])]
```

We have given a rough but accurate overview of how the Curry package manager compares the behavior of two versions of a package, all the way from an illustration

of the possible problems by example to the implementation in `CPM.Diff.Behavior` and how `diffBehavior`, `genCurryCheckProgram`, `genTranslatorFunction`, and `genTestFunction` interact to generate a Curry program containing CurryCheck tests that compare the two package versions.

6

Evaluation

In this chapter, we will give a few examples of successful and conflicting dependency resolutions as well as API and behavior comparisons. We will then evaluate the performance of the resolution algorithm for success and conflict cases. Finally, we will give some performance measurements for API and behavior comparisons.

6.1 Comparing Package Versions

To evaluate the API and behavior comparison functionality, we create a package called `foo` in versions 1.0.0 and 1.0.1. `foo` contains only one exported module, `Foo.Bar`. In version 1.0.0 of `Foo.Bar`, there is a data type `Greeting` which is a record containing fields `first` and `last`. `last` is of type `String`, while `first` is of type `Name`, which is a type synonym for `String`. The function `sayHello` takes a `Greeting` and returns a `String` containing a greeting:

Listing 6.1: Version 1.0.0

```
module Foo.Bar (sayHello, Greeting (..), Name) where

import DetParse
import Char

type Name = String

data Greeting = Greeting
```

```
  { first :: Name
  , last :: String }

sayHello :: Greeting -> String
sayHello (Greeting first last) = "Hi " ++ first ++ " " ++ last
```

In version 1.0.1, the data types are the same as before, but the string generated by `sayHello` is different. Additionally, it contains a new function `hiThere`, which returns a `String`.

Listing 6.2: Version 1.0.1

```
module Foo.Bar (sayHello, hiThere, Greeting (..), Name) where

type Name = String

data Greeting = Greeting
  { first :: Name
  , last :: String }

sayHello :: Greeting -> String
sayHello (Greeting first last) = "Hello " ++ first ++ " " ++ last

hiThere :: String
hiThere = "Hi!"
```

We run the following command from the directory of `foo-1.0.1` to compare its public API to that of version 1.0.0.

```
cpm diff 1.0.0 --api-only
```

Since the types of `Name`, `Greeting` and `sayHello` are unchanged, the only output we receive is that `hiThere` has been added in version 1.0.1. Since adding new functionality is not allowed in a patch version, we are also notified that we are in violation of semantic versioning:

```
Now running API diff

Added hiThere :: [Char]
  Adding features in a patch version is a violation
  of semantic versioning.
```

To compare the behavior of both versions, we use the `--behavior-only` option of the `cpm diff` command and receive the following output:

```
The following functions were not compared:
    Foo.Bar.hiThere - Different function types or function missing

Now running behavior diff. You will be presented with the raw output of
    ↪ CurryCheck. The test functions are named after the functions they
    ↪  compare. If a test fails, the implementations differ.

Comparing functions Foo.Bar.sayHello
----------------------------------------------------------------------
CurryCheck: a tool for testing Curry programs (version of 17/08/2016)
----------------------------------------------------------------------
Analyzing module 'Compare'...
Properties to be tested:
test_Foo_Bar_sayHello
Generating main test module 'TEST8317'...and compiling it...
Executing all tests...
test_Foo_Bar_sayHello (module Compare, line 7) failed
Falsified by first test.
Arguments:
(Greeting [] [])
Results:
"Hello  "
======================================================================
FAILURES OCCURRED IN SOME TESTS:
test_Foo_Bar_sayHello (module Compare, line 7)
======================================================================
```

Since the two versions of `sayHello` produce different strings, the test `test_Foo_Bar_sayHello` has failed. `hiThere` was not compared, since it is only present in one version of the module.

Next, we introduce a new version, 1.0.5, of package `foo` which removes the `Name` type synonym and changes the `first` field of the `Greeting` data type to a `String`:

Listing 6.3: Version 1.0.5

```
module Foo.Bar (sayHello, hiThere, Greeting (..)) where
```

```
data Greeting = Greeting
  { first :: String
  , last :: String }

sayHello :: Greeting -> String
sayHello (Greeting first last) = "Hello " ++ first ++ " " ++ last

hiThere :: String
hiThere = "Hi!"
```

The API comparison to version 1.0.1 will complain that the `Name` type has been removed and that the type of `Greeting` has changed:

```
Now running API diff

Removed type Name = String
  Removing features in a patch or minor version is
  a violation of semantic versioning.
Changed data Greeting (1 constructor) to data Greeting (1 constructor)
  Changing APIs in a patch or minor version is a
  violation of semantic versioning.
```

As the `Greeting` type has changed, the `sayHello` functions cannot be compared using CurryCheck anymore, so running a behavior comparison from version 1.0.5 to version 1.0.1 will not actually compare anything:

```
The following functions were not compared:
    Foo.Bar.hiThere - Different function types or function missing
    Foo.Bar.sayHello - Some types inside the function type differ
```

6.2 *A Sample Dependency Resolution*

In this section, we show three resolution problems of moderate complexity. One that can be resolved successfully, one that results in a conflict because of incompatible dependency constraints, and a third that results in a compiler conflict. The packages versions $\mathcal{V}$, dependency constraints $\mathcal{D}$ and compiler constraints $\mathcal{C}$ shown in Figure 6.1 will be the basis for all three cases. We will run the resolution algorithm

on the `rest-client` package. Figures 6.2, 6.3 and 6.4 give a clearer picture of the dependencies of `rest-client` in versions 1.0.0, 1.0.1, and 1.0.2, respectively.

There are no dependency or compiler conflicts for `rest-client-1.0.0`, so resolution will succeed. Upon successful resolution, the Curry package manager will print a tree representation of all transitive dependencies and the versions chosen for each package. For `rest-client-1.0.0`, the tree looks like this:

```
rest-client-1.0.0
  |- http-client-1.0.0
    |- network-1.2.0
      |- sockets-1.0.0
        |- bytes-1.0.0
      |- ip-addresses-1.0.0
        |- bytes-1.0.0
    |- det-parse-1.0.0
    |- string-encodings-1.0.6
  |- json-1.0.0
    |- det-parse-1.0.0
    |- bytes-1.0.0
```

Note that this is essentially the tree from Figure 6.2. As can be seen in Figure 6.3, dependency resolution on `rest-client-1.0.1` will result in a dependency conflict: the constraints on `bytes` of `json-1.0.0` and `sockets-1.0.1` are incompatible. If a resolution is unsuccessful, the Curry package manager prints information on the conflict that was encountered:

```
There was a conflict for package bytes
rest-client
  |- http-client (http-client = 1.0.1)
    |- network (network = 1.2.2)
      |- sockets (sockets = 1.0.1)
        |- bytes (bytes >= 1.0.5)
rest-client
  |- json (json = 1.0.0)
    |- bytes (bytes = 1.0.0)
```

The conflicting package is named and the two dependency constraints are printed along with their origins. This gives the user as much information as possible on how to resolve the problem. In this case, they can quickly see that `http-client` and `json`

are not compatible to one another in the currently required versions.

The dependencies of `rest-client-1.0.2` are compatible, but the `json-1.0.1` package is only compatible to the KiCS2 compiler in version 0.4.0 or lower. Since we are using KiCS2 0.5.1, resolving the dependencies for `rest-client-1.0.2` will result in a compiler conflict:

```
The package json-1.0.1, dependency constraint json = 1.0.1, is not
    ↪ compatible to the current compiler. It was activated because:
rest-client
  |- json (json = 1.0.1)
```

6.3 *Performance of the Resolution Algorithm*

The implementation of the classic backtracking algorithm given in Section 5.3 uses lazy evaluation to evaluate only those parts of a – potentially very large – candidate tree that are needed to find a solution or conflict. This lets us specify a concise and readable implementation. In this section, we want to examine the performance of this implementation.

To make the performance measurements as relevant as possible, we want to run the algorithm on a complex but realistic problem. To find such a problem, we used information from the *npm registry*, the central package index of the Node package manager (see Section 3.4). The npm registry is particularly well suited for finding realistic examples, because its package specification format is similar to the Curry package manager's and it provides usage statistics for all packages as well as leaderboards of the most-used packages. Furthermore, npm encourages its users to create many small packages that depend on one another, leading to larger problems for the algorithm to solve.

First, we have to acquire the relevant package specifications from the npm registry and convert them to the Curry package manager's format. At its core, the npm registry is a large Apache CouchDB[1] database. We used CouchDB's built-in replication functionality to copy the whole npm metadata registry to a local Couch DB instance, 314,633 packages in total. Next, we chose five packages from the npm leader board,

[1] `http://couchdb.apache.org`

$\mathcal{V} = \{$`rest-client-1.0.0`, `rest-client-1.0.1`, `rest-client-1.0.2`, `http-client-1.0.0`,
`http-client-1.0.1`, `network-1.2.0`, `network-1.2.2`, `det-parse-1.0.0`,
`string-encodings-1.0.5`, `string-encodings-1.0.6`, `sockets-1.0.0`, `sockets-1.0.1`,
`ip-addresses-1.0.0`, `bytes-1.0.0`, `bytes-1.0.5`, `json-1.0.0`, `json-1.0.1`$\}$

$\mathcal{D} = \{$`rest-client-1.0.0` $\Rightarrow$ `http-client` $=$ `1.0.0`,
`rest-client-1.0.0` $\Rightarrow$ `json` $=$ `1.0.0`,
`rest-client-1.0.1` $\Rightarrow$ `http-client` $=$ `1.0.1`,
`rest-client-1.0.1` $\Rightarrow$ `json` $=$ `1.0.0`,
`rest-client-1.0.2` $\Rightarrow$ `http-client` $=$ `1.0.1`,
`rest-client-1.0.2` $\Rightarrow$ `json` $=$ `1.0.1`,
`http-client-1.0.0` $\Rightarrow$ `network` $=$ `1.2.0`,
`http-client-1.0.0` $\Rightarrow$ `det-parse` $=$ `1.0.0`,
`http-client-1.0.0` $\Rightarrow$ `string-encodings` $\geq$ `1.0.5`,
`http-client-1.0.1` $\Rightarrow$ `network` $=$ `1.2.2`,
`http-client-1.0.1` $\Rightarrow$ `det-parse` $=$ `1.0.0`,
`http-client-1.0.1` $\Rightarrow$ `string-encodings` $\geq$ `1.0.5`,
`network-1.2.0` $\Rightarrow$ `sockets` $=$ `1.0.0`,
`network-1.2.0` $\Rightarrow$ `ip-addresses` $=$ `1.0.0`,
`network-1.2.2` $\Rightarrow$ `sockets` $=$ `1.0.1`,
`network-1.2.2` $\Rightarrow$ `ip-addresses` $=$ `1.0.0`,
`sockets-1.0.0` $\Rightarrow$ `bytes` $=$ `1.0.0`,
`sockets-1.0.1` $\Rightarrow$ `bytes` $\geq$ `1.0.5`,
`ip-addresses-1.0.0` $\Rightarrow$ `bytes` $\geq$ `1.0.0`,
`json-1.0.0` $\Rightarrow$ `det-parse` $=$ `1.0.0`,
`json-1.0.0` $\Rightarrow$ `bytes` $=$ `1.0.0`,
`json-1.0.1` $\Rightarrow$ `det-parse` $=$ `1.0.0`,
`json-1.0.1` $\Rightarrow$ `bytes` $\geq$ `1.0.0`$\}$

$\mathcal{C} = \{$`json-1.0.1` $\Rightarrow$ `kics2` $\leq$ `0.4.0`$\}$

Figure 6.1.: The example resolution problem.

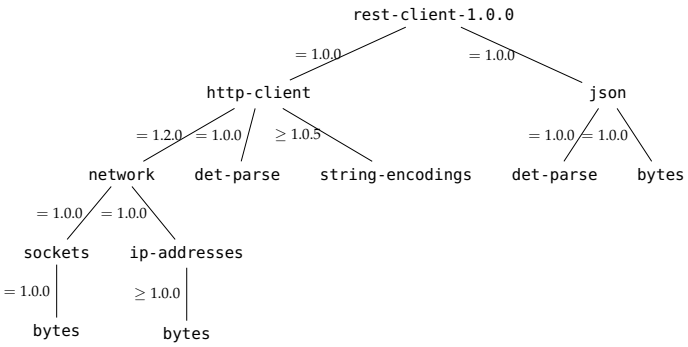


Figure 6.2.: Dependencies for `rest-client-1.0.0`

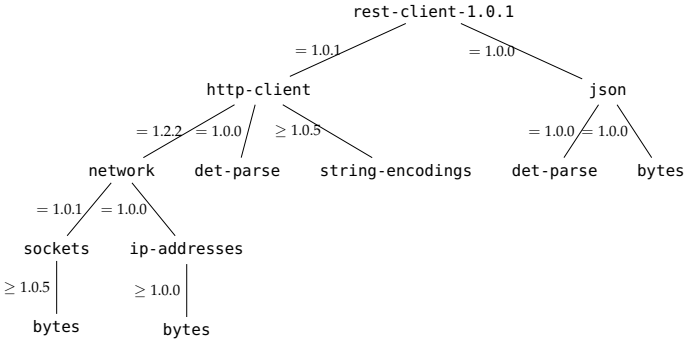


Figure 6.3.: Dependencies for `rest-client-1.0.1`

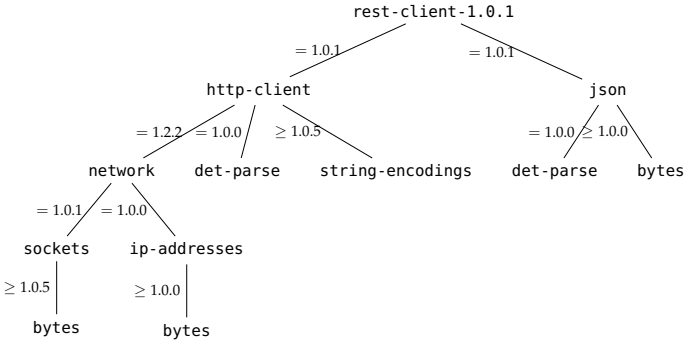


Figure 6.4.: Dependencies for `rest-client-1.0.2`

`express`[2], a framework for writing web applications, `chalk`[3], a library for coloring terminal output, `request`[4], a HTTP client, `mocha`[5], a testing framework, and `karma`[6], another testing framework. As of August 30th, 2016, all packages were downloaded between 300,000 and several million times per month.

Using a simple Ruby script[7], we extracted all versions of the packages above as well as all of their transitive dependencies from the npm registry. The script reads each package version's specification from CouchDB and converts it to a format understood by the Curry package manager. The Node package manager supports a few more ways of specifying dependency constraints and is not as strict regarding the format of version numbers. To avoid over complicating the Ruby script, we skip package versions whose version numbers are not in semantic versioning format. Furthermore, we replace npm's caret operator (`^`), which allows all versions from the specified version up to but not including the next major version, by a simple greater than or equal operator, which widens the compatibility range of the dependency constraint. Afterwards, the script writes the CPM package specification files to disk in the index directory structure described in Section 4.3. On our copy of the registry, the script extracted a total of 2,058 packages and 29,553 versions of those packages.

We wrote a performance test program, implemented in the `CPM.PerformanceTest` module, to load the package index created by the script into a lookup set and run the resolution algorithm on the packages we selected. To test the algorithm on problems of varying sizes and include successful as well as unsuccessful resolutions, we experimented with various versions of these packages. Ultimately, we chose version 4.14.0 of `express` and version 2.74.0 of `request` as examples of complex problems with a solution. Version 1.1.3 of `chalk` is a small problem that can be resolved successfully. `express-3.9.0` is complex and cannot be resolved successfully because of a missing dependency in our data set. `mocha-1.21.5` is complex and results in a failure because of a dependency conflict, while the algorithm fails to arrive at a result for `karma-1.2.0`. We run the algorithm five times on each problem and use Curry's built-in `Profile` module to measure the time taken. Additionally, we measure the time taken to load the package index from the directory structure generated by the Ruby script. The gist of the test program can be found in Listing 6.4. We compiled

[2] `https://www.npmjs.com/package/express`
[3] `https://www.npmjs.com/package/chalk`
[4] `https://www.npmjs.com/package/request`
[5] `https://www.npmjs.com/package/mocha`
[6] `https://www.npmjs.com/package/karma`
[7] The script is distributed with the CPM source code under the path `misc/extract_npm_packages.rb`

the test program using KiCS2 0.5.1 on GHC 7.10.2 and PAKCS 1.14.1 on SICStus Prolog 4.3.3 and ran each version on a system with an Intel Core i5-5257U processor running Mac OS X 10.11. The raw performance figures and instructions on how to use the performance test program to reproduce them are given in Appendix C.

The first weakness exposed by the performance measurements is the current structure of the package index: Reading a file for each package version does not scale to large data sets. Loading the 29,533 package specifications from the sample data sets takes an average of 12.5 seconds on KiCS2 and fails with an out of memory error on PAKCS. Since the Curry ecosystem is small, it will take some time for the central package index to reach a number of package versions that will noticeably impact performance. Possible approaches to improve the performance of the index are discussed in Chapter 7 on future work.

Listing 6.4: Performance test program

```
main = do
  putStrLn "Reading package specifications..."
  ls <- profileTime (readLS "/tmp/npm-repo")
  e4140 <- return $!! fromJust $ findVersion ls "express" (4, 14, 0,
      ↪ Nothing)
  e390 <- return $!! fromJust $ findVersion ls "express" (3, 9, 0,
      ↪ Nothing)
  c113 <- return $!! fromJust $ findVersion ls "chalk" (1, 1, 3, Nothing)
  r2740 <- return $!! fromJust $ findVersion ls "request" (2, 74, 0,
      ↪ Nothing)
  m1215 <- return $!! fromJust $ findVersion ls "mocha" (1, 21, 5,
      ↪ Nothing)
  k120 <- return $!! fromJust $ findVersion ls "karma" (1, 2, 0, Nothing)
  putStrLn "Now starting resolution"
  -- five times for each of the above.
  profileTime (putStrLn $ showResult $ resolve e4140 ls)
```

Since we want to measure the algorithm's performance on both PAKCS and KiCS2, we have to find a way to work around PAKCS'/SICStus' limitations when reading many small files. Curry's standard library includes the `ReadShowTerm` module, which contains functions for rendering Curry values into string representations and parsing these string representations back into Curry values. We can use these functions to write out one large file containing a string representation of the packages in the lookup set from a program compiled using KiCS2. A version of the performance test

Table 6.1.: Median time used for dependency resolution on KiCS2 and PAKCS in msec

Package	No. of transitive dependencies	No. of package versions of those dependencies	KiCS2	PAKCS
`express-4.14.0`	1,795	23,295	5	350
`express-3.9.0`[a]	1,794	23,286	241	25,070
`chalk-1.1.3`	8	65	0	10
`request-2.74.0`	1,789	23,229	17	1,330
`mocha-1.21.5`[a]	1,789	23,229	4,532	Did not finish[b]
`karma-1.2.0`	1,850	24,264	Did not finish[b]	Did not finish[b]

[a] Results in conflict.
[b] Aborted after 10 minutes.

program compiled using PAKCS will then be able to read this larger file and run the performance tests. The median time taken by each compiler to run the resolution algorithm for the different packages is shown in Table 6.1.

The data show that PAKCS is slower than KiCS2, that conflicts take a lot longer than successful resolutions, and that some problems are too large for the algorithm to handle. The performance difference between KiCS2 and PAKCS is consistent with the measurements in [Bra+11], which show that KiCS2 produces much faster programs than PAKCS. Successful resolution runs finishing faster than unsuccessful ones can be explained by the different strategies for the two cases: if a solution exists, we stop searching the tree once we have found one. To arrive at the conclusion that there is no solution, however, we have to search the entire tree to make sure that all possible branches result in failure and then find the most relevant conflict.

`karma-1.2.0` transitively depends on 1,850 packages with a total of 24,264 versions, while `mocha-1.21.5` transitively depends on 1,789 packages with a total of 23,229 versions, so both problems are roughly similar in size. The dependency constraints for `mocha-1.21.5`, however, are very strict for some its direct dependencies and those dependencies are considered early in the resolution process, resulting in a small tree. In contrast, `karma`'s dependency constraints are more liberal and its conflicts appear deeper in the tree, so many subtrees are examined multiple times.

The algorithm's performance is acceptable on large problems in successful cases,

using both KiCS2 and PAKCS, even though KiCS2 is much faster. Problems of a size similar to those presented here – excluding `chalk` – are unlikely to occur in practice for some time due to the current size of the Curry ecosystem. Multiple optimizations are possible should the algorithm's performance become a problem in the future, some of which are discussed in Chapter 7 on future work.

6.4 Performance of API and Behavior Comparison

Section 6.1 shows the API and program behavior comparison described in Sections 5.4 and 5.5 from the user's perspective. In this section, we will evaluate the runtime performance of both functionalities on artificially generated Curry modules of varying sizes.

We run the API comparison algorithm on two artificially generated Curry modules, A and B. A contains n functions and types that B does not, and vice versa, where n is varied between 10, 100, 1,000 and 10,000. Additionally, B contains n functions and types that are also present in A, but with different types and constructors, respectively.

For the behavior comparison test, we generate two Curry modules that contain exactly the same functions and types. Each of the $m \in \{10, 100, 1000\}$ generated functions takes a single parameter of a newly generated type τ_1 and returns that parameter. In effect, all generated functions are the identity function on τ_1. To exclude those functions that cannot be compared via CurryCheck, the type of each parameter is inspected recursively – i.e., the types mentioned in its constructors are inspected as well – for function types, `Floats`, and so on (see Section 4.7 for details). We generate not only a type τ_1, but multiple types $\tau_1, ..., \tau_n$ for some $n \in \{10, 100, 1000\}$ where τ_i references τ_{i+1} for $i \leq n$ and τ_n references `String`, to measure the performance impact of these parameter type checks. We call n the *type nesting depth*.

As in the previous section, we compiled the test program using KiCS2 0.5.1 on GHC 7.10.2 and PAKCS 1.14.1 on SICStus Prolog 4.3.3 and ran each version on a system with an Intel Core i5-5257U processor running Mac OS X 10.11. The raw performance figures and instructions on how to reproduce them can be found in Appendix C.

Table 6.2 shows that the API comparison algorithm is sufficiently fast on KiCS2, even for large problems. It does fail to arrive at a result for a problem with 10,000

Table 6.2.: Median time used for API diff on KiCS2 and PAKCS in msec

No. of changed, added, removed functions and types each	KiCS2	PAKCS
10	2	60
100	2	50
1,000	2	Out of memory[a]
10,000	Did not finish[b]	Out of memory[a]

[a] Aborted after 15GB of memory usage.
[b] Aborted after 10 minutes.

Table 6.3.: Median time used for behavior diff on KiCS2 and PAKCS in msec

Type nesting depth	No. of functions	KiCS2	PAKCS
10	10	290	1,450
10	100	530	7,460
10	1,000	5,256	Out of memory[a]
100	10	460	10,960
100	100	718	18,990
100	1,000	5,666	Out of memory[a]
1,000	10	8,141	Out of memory[a]
1,000	100	8,611	Out of memory[a]
1,000	1,000	17,627	Out of memory[a]

[a] Aborted after 15GB of memory usage.

changed, removed, and added types and functions each. However, a package with 60,000 exported functions and types is unlikely to occur in practice. On PAKCS, the algorithm is much slower than KiCS2 on smaller problems, although its performance is still acceptable for real-world use. For larger problems of 6,000 and 60,000 total exported functions and types, the test program fails while generating the Curry modules to be compared.

The performance figures for the behavior comparison algorithm are shown in Table 6.3. Note that the time measured is for the generation of the test program only and does not include the time taken by CurryCheck. The data show that a type nesting depth of 1,000 has a significant impact on the total runtime of the algorithm when compared to a depth of 100 or 10. Luckily, a type with a nesting depth of 1,000 is unlikely to occur in practice. Even a nesting depth of 10 is rather rare: out of 343 type declarations in all modules in Curry's standard library, only 26 have a nesting depth of 10 or higher. The highest nesting depth is 13.[8]

With a nesting depth of 10, the performance figures on KiCS2 are acceptable for everyday use. Modules with 1,000 exported functions that need to be compared should be rare in practice. PAKCS fails when either the number of functions or the type nesting depth is set to 1,000. Performance on smaller problems is significantly worse than on KiCS2 – as is to expected according to Braßel et al. [Bra+11] – but still tolerable for a type nesting depth of 10.

[8] The program used to calculate the nesting depths of the types in Curry's standard library is distributed alongside the CPM source code in the `misc` directory.

7

Summary & Future Work

In this thesis, we discussed the design and implementation of the Curry package manager, a package management system for the Curry programming language. First, we determined the state of the art in programming language-specific package managers by examining three popular package management systems, Ruby's Gems, JavaScript's npm, and Haskell's Cabal.

Based on our findings and the particulars of the Curry language and compiler ecosystem, we developed a format for Curry packages as well as approaches for distributing, installing, and using them with the KiCS2 and PAKCS compilers. Specifically, we developed a simple specification for package metadata files, including a way to indicate the dependencies of a package in a flexible manner. We also decided on a fixed format for version numbers, namely the one defined in the semantic versioning specification, and a simple directory structure for packages.

To avoid some of the frequent dependency conflicts that plague the Cabal package manager, we decided not to compile packages upon installation. Instead, packages are installed in source code form and only compiled when needed, i.e., when a dependent package is compiled. We chose a simple mode of interaction with the KiCS2 and PAKCS compilers: we add the paths of all dependencies to their module search paths via an environment variable supported by both systems. This way, we avoid having to modify either of the compilers and having compiler-specific behavior in the Curry package manager.

A mainstay of many popular package management systems, including the ones examined in detail in this thesis, is some form of central package index. Central pack-

age indices aid the user in finding new packages and lower the barrier to using them. We presented a simple form of centralized package index based on a directory structure distributed via the Git version control system and implemented support for it in the package manager. This implementation enables easy installation of packages, but only slightly improves their discoverability. A possible future improvement is a web interface for the package index, which is discussed in more detail in the next section.

We gave a general definition of packages, package versions, dependencies and dependency resolution and adapted those definitions to the Curry package manager. In particular, we developed a textual format for version constraints and defined how to choose between multiple compatible package versions. Based on this, we developed a functional implementation of the classic backtracking algorithm for dependency resolution, as well as a model for finding and reporting the most relevant conflict when dependency resolution fails. The performance of this implementation is acceptable for day-to-day use, but it fails to arrive at a solution on large problems. Possible improvements are discussed in the next section.

In addition to using the semantic versioning version number format, we also strongly encourage package authors to follow the guidelines from the semantic versioning specification on when to increase which part of the version number. To aid package authors in following these guidelines, we developed a way to automatically compare the public APIs and the behavior of two package versions. While at least one package manager – *elm-package*[1] – exists that can compare the public APIs of two package versions, to the best of our knowledge the Curry package manager is the first package manager incorporate any form of behavior comparison.

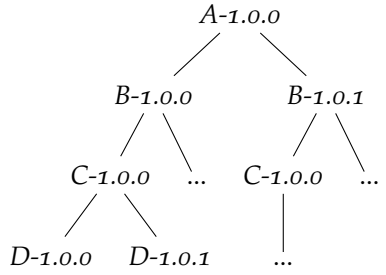
7.1 Future Work

While the Curry package manager is usable for day-to-day work in its current form, there are several aspects that could be improved.

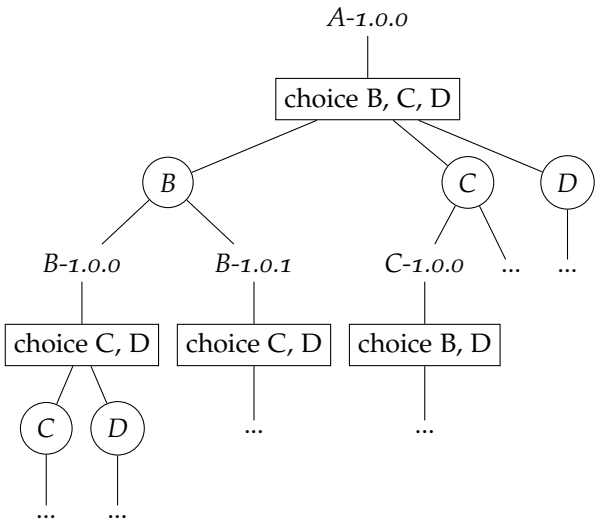
Improve Performance of the Resolution Algorithm

In Section 6.3, we have seen that the implementation of the backtracking algorithm developed in Section 5.3 may not arrive at a solution in any reasonable timeframe for large problems – in this case, over 1,500 dependencies and 23,000 total versions

[1] `http://elm-lang.org`



(a) A tree from our implementation



(b) Cabal's tree containing choice nodes

Figure 7.1.: Our tree and Cabal's tree

of those dependencies. Two possible approaches for performance improvements can be found in the implementation of Cabal's resolution algorithm, described by Löh [Löh11]. Cabal's implementation, like ours, is based on the ideas presented by Nordin and Tolmach [NT01].

In our implementation, when we generate the search tree for the backtracking algorithm, we add the dependencies of a package version to the tree in the order they are given in the package specification. If, for example, package *A* depends on packages *B*, *C*, and *D*, which have no further dependencies themselves, then the search tree will list all versions of *B* on the first level, all versions of *C* on the second level, and all versions of *D* on the third level. In contrast, Cabal's search tree contains additional *choice nodes*. Each choice node branches off into nodes for each of the dependencies that are not yet contained in the tree. In the above example, the root node would have a choice node as its sole child, which would then branch out into package nodes for *B*, *C*, and *D*. These package nodes would in turn branch out into nodes for each package version, just like our implementation's search tree. Each of those package version nodes would lead to another choice node. Both trees are shown in Figure 7.1. Choice nodes are shown as rectangles, package nodes as circles.

Encoding the order of dependencies into the tree enables Cabal to choose which

dependency to explore next based upon potentially better criteria than the order in which they are listed in a package specification. Example criteria are the number of versions available of each package or whether all versions of a package are incompatible to some existing dependency constraint. Before the tree is explored for solutions, every choice node is replaced by its first child. As we have seen in Section 6.3, the order in which dependencies are explored can have a significant impact on the execution time of the resolution algorithm. Implementing a tree based on choice nodes in the Curry package manager would enable quick experimentation with different heuristics for which dependencies to explore first.

In addition to introducing choice nodes, Cabal uses *backjumping* instead of backtracking when exploring the tree for solutions. In backtracking, subtrees of inconsistent nodes are removed from the tree, on the grounds that no descendant of an inconsistent node can ever become consistent. Backjumping adds the upwards propagation of inconsistencies: if a node has an inconsistent child whose inconsistency does not involve the node's package, i.e., the node does not contribute to the conflict via one of its package version's dependencies, then we can mark that node with its child's inconsistency. This will lead to inconsistencies higher up in the tree and thus larger subtrees that can be pruned.

Improve Performance of the Package Index

In Section 6.3, we saw that the simple file-based approach for the central package index does not scale to a large number of packages. KiCS2 took 12.5 seconds to read an index containing roughly 30,000 package versions, while PAKCS failed with an out of memory error. A possible approach to improving startup times with large indexes is not reading the whole index at once. Instead, we would only read a package version's specification when it is needed. For large resolution problems, such as the ones presented in Section 6.3, however, this approach is not likely to lead to a big increase in overall performance, since more than 20,000 packages would still have to be read from the index.

Other possible approaches include using a database as a cache in front of the package index and replacing the package index by a web service that can be queried for packages and their (transitive) dependencies.

Website for Central Package Index

To aid discoverability of packages in the central package index, a website listing all packages in the index is desirable. Ideally, such a website would offer search

functionality and CurryDoc [Han02] documentation for each version of a package.

Improve Integration with CurryCheck

We use CurryCheck to compare the behavior of two package versions. Currently, we only generate the test program, execute `currycheck` on this test program and present the user with its output. The user is left to correctly interpret the output of CurryCheck and the names of the generated property tests. If there was a way to interact with CurryCheck via an API, we could present the results of the tests to the user in a more friendly manner.

Integration with Compilers

Currently, the only interaction with the KiCS2 and PAKCS compilers is via the `CURRYPATH` environment variable. This approach has the upside of not requiring any customization of the compilers themselves. A distinctive downside, however, is that we cannot use the list of exported modules inside a package specification to hide any internal modules a package may contain from the consumers of that package. Package authors have to rely on convention, e.g. putting all internal modules inside of an `Internal` namespace, to reduce the chances of internal modules being used.

If KiCS2 and PAKCS were extended to have knowledge about the Curry package manager, they could offer an option that would allows us to specify the available modules for each package alongside the paths to their modules.

Cache Compilation Results

In Section 4.6, we saw that when compiling a package's modules, we create copies of all of its transitive dependencies to avoid inadvertently reusing incompatible compilation results. Specifically, compilation results for a package version v_c obtained with version v_{d_1} of one of its dependencies may not be reusable when v_c is required to work with version v_{d_2} of the same dependency. The current behavior is wasteful, since compilation results are *never* reused, even when they are compatible.

Dolstra, Löh, and Pierron [DLP10] propose a method for caching compilation results when different dependency versions are involved, which could also be applied to the Curry package manager: when a package version v is compiled with a set of dependencies D, convert the names and versions of those dependencies into a canonical format – for example, a comma-separated list sorted alphabetically. Then, hash the canonicalized dependency information using a suitable hash function, e.g.

SHA256. Store the compilation results in a directory named `package-version-hash`, e.g. `foo-1.0.0-b6a890cfde...`. The next time v is to be compiled with a set of dependencies D', convert the names of those dependencies into the same canonical format, compute the hash and check if a directory named with that hash exists. If it does, we can reuse those compilation results.

Allow Distribution of Tools

Currently, a Curry package can only contain Curry modules to be used by other Curry packages via `import`. In package managers for other languages, e.g. Ruby's Gems and Node's npm, it has proven useful to allow package authors to include tools in a package that are then added to the user's `PATH`, either globally or local to a package. For example, a unit testing library might include a command to run tests or a web application framework might offer a command that can start a development web server.

Implementing such a feature in the Curry package manager poses a few challenges. Ideally, the commands or executables distributed with a Curry packages should themselves be implemented on Curry. In this case, however, they would have to be compiled either when the package is installed globally if the executable is to be available globally, or, for local availability, when the package is activated during a `cpm install` run. Furthermore, a system is needed for running locally installed executables from the dependent package's directory. Possible solutions are requiring the user to run everything through `cpm exec`, or creating *binstubs*[2] on package installation. Similarly, a strategy for making the executables of globally installed packages available on the user's `PATH` will have to be devised: the commands of which version of a package should be available globally? Should there be a way to access the commands of other versions? What happens when a global executable is run from within a package directory that has another version of that executable available locally?

[2] `https://github.com/rbenv/rbenv/wiki/Understanding-binstubs`

A

Total Order on Versions

In Section 3.1, we defined the relation $\leq_{pre}$ on pre-release specifiers as well as the relation $\leq_{ver}$ on versions. We will show that both of these relations are total orders on Σ^*_{pre} and V, respectively.

Lemma A.1. *The relation $\leq_{pre}$ is a total order on Σ^*_{pre}.*

Proof. By definition, $\leq_{pre}$ is the union of three sets:

$$\begin{aligned}\leq_{pre} = &\{(a,b) \mid a,b \in \Sigma^*_{pre},\ a,b \text{ numeric},\ strToInt(a) \leq strToInt(b)\}\\ &\cup \{(a,b) \mid a,b \in \Sigma^*_{pre},\ a \text{ numeric},\ b \text{ non-numeric}\}\\ &\cup \{(a,b) \mid a,b \in \Sigma^*_{pre},\ a,b \text{ non-numeric},\ a \leq_{sx} b\}\end{aligned}$$

Since an element of Σ^*_{pre} is either numeric or non-numeric but never both, the three sets are disjunct. Let $a \in \Sigma^*_{pre}$. If a is numeric, then $strToInt(a) \leq strToInt(a)$, since $\leq$ is a total order on $\mathbb{N}$ and thus reflexive. Thus, $a \leq_{pre} a$. If a is non-numeric, then $a \leq_{sx} a$, since $\leq_{sx}$ is a total order on Σ^*_{pre} and thus $a \leq_{pre} a$. It follows that $\leq_{pre}$ is reflexive.

Now let $a, b \in \Sigma^*_{pre}$ with $a \leq_{pre} b$ and $b \leq_{pre} a$. If both a and b are numeric, then $a = b$ follows from the antisymmetric property of $\leq$ on $\mathbb{N}$. If neither a nor b are numeric, then $a = b$ follows from the antisymmetry of $\leq_{sx}$ on Σ^*_{pre}. By definition, a cannot be numeric if b is non-numeric and vice versa, so the above cases are all that we need to cover. Thus, $\leq_{pre}$ is antisymmetric.

Let $a, b, c \in \Sigma^*_{pre}$ with $a \leq_{pre} b$ and $b \leq_{pre} c$. The sixteen possible cases of a, b and c being numeric/non-numeric are given in the following table, with n standing for numeric and $\bar{n}$ for non-numeric.

a	b	c	
n	n	n	$a \leq_{pre} c$, since $\leq$ is transitive on $\mathbb{N}$
n	n	$\bar{n}$	$a \leq_{pre} c$, since numeric is always smaller
n	$\bar{n}$	n	impossible, since numeric is always smaller and thus $b > c$
n	$\bar{n}$	$\bar{n}$	$a \leq_{pre} c$, since numeric is always smaller
$\bar{n}$	n	n	impossible, since numeric is always smaller and thus $a > b$
$\bar{n}$	n	$\bar{n}$	impossible, since numeric is always smaller and thus $a > b$
$\bar{n}$	$\bar{n}$	n	impossible, since numeric is always smaller and thus $b > c$
$\bar{n}$	$\bar{n}$	$\bar{n}$	$a \leq_{pre} c$, since $\leq_{sx}$ is transitive in Σ^*_{pre}

In all cases that can occur, we have $a \leq_{pre} c$. Thus, $\leq_{pre}$ is transitive.

Since $\leq_{pre}$ is reflexive, antisymmetric and transitive on Σ^*_{pre}, it is a total order on Σ^*_{pre}. □

Lemma A.2. *The relation $\leq_{ver}$ is a total order on V.*

Proof. We recall the definition of $\leq_{ver}$:

$$
\begin{aligned}
prC((a,b,c,p_1),(x,y,z,p_2)) &:= p_1 \leq_{pre} p_2 \\
paC((a,b,c,p_1),(x,y,z,p_2)) &:= c < z \vee (c = z \wedge prC((a,b,c,p_1),(x,y,z,p_2))) \\
miC((a,b,c,p_1),(x,y,z,p_2)) &:= b < y \vee (b = y \wedge paC((a,b,c,p_1),(x,y,z,p_2))) \\
maC((a,b,c,p_1),(x,y,z,p_2)) &:= a < x \vee (a = x \wedge miC((a,b,c,p_1),(x,y,z,p_2))) \\
\leq_{ver} &:= \{(v_1,v_2) \mid v_1, v_2 \in V,\ maC(v_1,v_2)\}
\end{aligned}
$$

Let $(a,b,c,d) \in V$. Then $prC((a,b,c,d),(a,b,c,d))$ holds since $\leq_{pre}$ is a total order and thus $d \leq_{pre}$. Since $c = c$, $paC((a,b,c,d),(a,b,c,d))$ holds as well and similarly $miC((a,b,c,d),(a,b,c,d))$ and $maC((a,b,c,d),(a,b,c,d))$ hold since $b = b$ and $a = a$. Thus, $(a,b,c,d) \leq_{ver} (a,b,c,d)$ and $\leq_{ver}$ is reflexive.

Let $a, b \in V$ with $a \leq_{ver} b$ and $b \leq_{ver} a$. Then there exist $a_1, b_1, a_2, b_2, a_3, b_3 \in \mathbb{N}$ and $a_4, b_4 \in \Sigma^*_{pre}$ with $a = (a_1, a_2, a_3, a_4)$ and $b = (b_1, b_2, b_3, b_4)$. Since both $maC(a,b)$

and $maC(b, a)$ must hold by choice of a and b, we can conclude that $a_1 = b_1$ since otherwise $a_1 < b_1$ and $b_1 < a_1$, which is a contradiction. From $a_1 = b_1$ we have that $miC(a, b)$ and $miC(b, a)$ hold and we can conclude $a_2 = b_2$ by a similar argument. $a_2 = b_2$ in turn gives us that $paC(a, b)$ and $paC(b, a)$ hold and since $a_3 = b_3$, which gives us that $prC(a, b)$ and $prC(b, a)$ hold. Since $\leq_{pre}$ is antisymmetric, we get $a_4 = b_4$. Thus $a = b$ since all their components are equal and $\leq_{ver}$ is antisymmetric.

Let $a, b, c \in V$ with $a \leq_{ver} b$ and $b \leq_{ver} c$. Then there exist $a_1, b_1, c_1, a_2, b_2, c_2, a_3, b_3, c_3 \in \mathbb{N}$ and $a_4, b_4, c_4 \in \Sigma^*_{pre}$ with $a = (a_1, a_2, a_3, a_4)$, $b = (b_1, b_2, b_3, b_4)$ and $c = (c_1, c_2, c_3, c_4)$. We know that $maC(a, b)$ and $maC(b, c)$ hold, and thus $a_1 < b_1 \leq c_1$ or $a_1 = b_1 \leq c_1$. In the first case, we have $maC(a, c)$ and thus $a \leq_{ver} c$. In the second case, we know that $miC(a, b)$ and $miC(b, c)$ hold. Thus $a_2 < b_2 \leq c_2$ or $a_2 = b_2 \leq c_2$. Again, in the first case we can conclude $a \leq_{ver} c$ since we have $miC(a, c)$. In the second case, we know that $paC(a, b)$ and $paC(b, c)$ hold and thus $a_3 < b_3 \leq c_3$ or $a_3 = b_3 \leq c_3$. In the first case, we know that $paC(a, c)$ holds and thus $a \leq_{ver} c$. In the second case, we know that $prC(a, b)$ and $prC(b, c)$ hold and thus $a_4 \leq_{pre} b_4$ and $b_4 \leq_{pre} c_4$, which gives us $a_4 \leq_{pre} c_4$ and thus $prC(a, c)$ by transitivity of $\leq_{pre}$. It follows that in every case $a \leq_{ver} c$ and thus $\leq_{ver}$ is transitive.

We can conclude that $\leq_{ver}$ is a total order on V since it is reflexive, antisymmetric and transitive on V. □

B

A Few Curry Packages

In this chapter, the `README` files of the packages `det-parse`, `json`, `opt-parse` and `boxes` are printed verbatim.

B.1 det-parse

det-parse is a library of deterministic parser combinators. It is based on the material presented in Frank Huch's functional programming lecture at Kiel University. To use it, you build a `Parser a` using the provided combinators and then apply it to a string using `parse`. The simplest parsers are provided by the primitives `yield`, `failure`, `anyChar` and `check`.

`yield` always results in the given value, consuming no input. `yield 1` will successfully parse the empty string to the value `1`. `failure` is a parser that always fails. `anyChar` is a parser that consumes a single character and uses it as the parse result. `check` takes a parser and a predicate on the result type of the parser. From these, it builds a new parser that applies the existing parser and succeeds only if the predicate holds for the parse result.

`char` and `word` build parsers for single characters and whole strings from these primitives. `char 'c'` is a parser that consumes the single character `c` and results in the unit value `()`. `word "hello"` consumes the string `hello` and results in the unit value. `empty` is a parser that recognizes an empty string and results in the unit value.

The operators `*>` and `<*` are provided to combine parsers into more complex ones. `*>`

applies two parsers and returns the result of the second one if both were successful. `char 'a' *> yield 1` is successful if applied to the string `a` and results in the value `1`. `<*` applies two parsers in the same order, i.e., left to right, but returns the result of the first one.

`<|>` combines two parsers by applying them both. If the first one is successful, it returns its result. If it is not, but the second one is, then it returns the result of the second one. If both are unsuccessful, the combined parser is unsuccessful as well. The parser `char 'a' *> yield 1 <|> char 'b' *> yield 2` parses the string `a` to the value `1` and the string `b` to the value `2`. `<!>` works similarly to `<|>`, but does not backtrack. That is, it only tries the second parser if the first one was unsuccessful, and only on the remaining input. It can be used if the alternatives do not overlap. The above example would also work if `<|>` were replaced by `<!>`, while `word "ab" *> yield 1 <!> word "abc" *> yield 2` would fail to parse `abc` into the value `2` since the first alternative has already consumed `ab`.

`<$>` builds a new parser from an existing parser by applying a function to the result of that parser. For example, `(+ 1) <$> (char 'a' *> yield 1)` is a parser that parses the string `a` into the value `2`.

`<*> :: Parser (a -> b) -> Parser a -> Parser b` combines two parsers, one that results in a function from `a` to `b`, and one that results in an `a` value. It applies the parsers in order and then applies the function result of the first parser to the value result of the second parser.

`many :: Parser a -> Parser [a]` builds a parser that parses whatever the original parser parses arbitrarily many times. `some` is similar, but requires that the original parser succeed at least once. Applying `many (char 'a' *> yield 1)` to the string `aaaa` results in the value `[1,1,1,1]`.

B.2 json

This package provides data types, a parser and a pretty printer for JSON[1].

Representing JSON values in Curry

A JSON value can be a primitive, i.e., `true`, `false`, `null`, a string or a number, an

[1] `http://json.org`

array of JSON values or an object mapping strings to JSON values. In Curry, a JSON value is represented by the data type `JValue` from the `JSON.Data` module:

```
data JValue = JTrue
            | JFalse
            | JNull
            | JString String
            | JNumber Float
            | JArray [JValue]
            | JObject [(String, JValue)]
```

Parsing JSON strings

`parseJSON` from `JSON.Parser` can be used to parse a JSON string into a `JValue`:

```
> parseJSON "{ \"hello\": [\"world\", \"kiel\"] }"
Just (JObject [("hello", JArray [JString "world", JString "kiel"])])
```

Printing JSON strings

`ppJSON` from `JSON.Pretty` will turn a `JValue` into a pretty printed string. If you want more control over the layout of the resulting string, you can use `ppJValue` from the same package to obtain a `Doc` for Curry's `Pretty` module from a `JValue`.

B.3 opt-parse

opt-parse is an advanced command line parser for Curry. It features support for options with and without values (i.e., flags), positional arguments and commands that can define their own sub-parsers. It borrows heavily from Paolo Capriotti's Haskell package optparse-applicative[2] and Curry's GetOpt[3] module.

You use opt-parse by declaring a *parser specification* and then running that parser specification on a command line. A parser specification is made up from individual parsers for options, flags, position arguments and commands. Each individual parser results in an arbitrary value, though all parsers in a parser specification must result in values of the same type.

[2] `https://hackage.haskell.org/package/optparse-applicative`
[3] `https://www-ps.informatik.uni-kiel.de/kics2/lib/GetOpt.html`

A Simple Example

A simple command line parser example might look like this:

```
cmdParser = optParser $
      option (\s -> readInt s)
        (  long "number"
        <> short "n"
        <> metavar "NUMBER"
        <> help "The number." )
  <.> arg (\s -> readInt s)
        (  metavar "NEXT-NUMBER"
        <> help "The next number." )

main = do
  args <- getArgs
  parseResult <- return $ parse (intercalate " " args) cmdParser "test"
  putStrLn $ case parseResult of
    Left err -> err
    Right  v -> show v
```

This defines a parser that supports a `number` option and requires a single positional argument. Both values are parsed into an integer. The `parse` function is called with the command line as a single string, the parser specification and the name of the current program. It results in either a `Left` if there was a parse error or a `Right` with the list of parse results. Running `test --help` prints out usage information:

```
test NEXT-NUMBER

-n, --number NUMBER      The number.

NEXT-NUMBER     The next number.
```

If we run `test --number=5 2`, we get the list of parse results:

```
[2, 5]
```

`metavar` and `help` are modifiers that can be applied to any argument parser, command, option, flag or positional. The `help` text is what is printed in the detailed usage output, the `metavar` is the placeholder to be printed for the argument's value

in the usage output. The `optional` modifier can also be applied to all argument types, although flags and options are already optional by default.

The `long` and `short` modifiers are specific to options and flags.

Right now, the result of our parser is a list of the individual parse results. Usually, we want our parse result to be a single value, for example a Curry data type such as this:

```
data Options = Options
  { number :: Int
  , nextNumber :: Int }
```

To parse a command line to an `Options` value, we return functions from our individual parsers instead of integers:

```
cmdParser = optParser $
      option (\s a -> a { number = readInt s })
        (  long "number"
        <> short "n"
        <> metavar "NUMBER"
        <> help "The number." )
  <.> arg (\s a -> a { nextNumber = readInt s })
        (  metavar "NEXT-NUMBER"
        <> help "The next number." )
```

The result of a successful parse will now be a list of functions that change an `Options` value. We can fold this list onto a default `Options`:

```
applyParse :: [Options -> Options] -> Options
applyParse fs = foldl (flip apply) defaultOpts fs
 where
  defaultOpts = Options 0 0

main = do
  args <- getArgs
  parseResult <- return $ parse (intercalate " " args) cmdParser "test"
  putStrLn $ case parseResult of
    Left err -> err
    Right  v -> show $ applyParse v
```

Executing `test --number=5 1` results in:

```
(Options 5 1)
```

Positional Arguments and Flags

Positional arguments can be created via `arg` and `rest`. `arg` is a normal positional argument which can be optional or mandatory. `rest` is a positional argument that consumes the rest of the command line as-is. Positional arguments are expected in the order they occur in the parser definition.

`flag` can be used to create flag arguments. A flag argument expects no value.

Commands

In addition to options, flags and positional arguments, opt-parse also includes support for commands. A command is a positional argument that dispatches to subparsers depending on its value. If we have a calculator program that supports addition and multiplication, we could model its command line interface using commands:

```
data Options = Options
  { operation :: Int -> Int -> Int
  , operandA :: Int
  , operandB :: Int }

cmdParser = optParser $
  commands (metavar "OPERATION")
    (   command "add" (help "Adds two numbers.") (\a -> a { operation =
        ↪ (+) })
          (   arg (\s a -> a { operandA = readInt s }
                (  metavar "OPERAND-A"
                <> help "The first operand." )
          <.> arg (\s a -> a { operandB = readInt s }
                (  metavar "OPERAND-B"
                <> help "The second operand." ) )
    <|> command "mult" (help "Multiplies two numbers.") (\a -> a {
        ↪ operation = (*) })
          (   arg (\s a -> a { operandA = readInt s }
                (  metavar "OPERAND-A"
                <> help "The first operand." )
```

```
      <.> arg (\s a -> a { operandB = readInt s }
            (  metavar "OPERAND-B"
            <> help "The second operand." ) ) )
```

The corresponding usage output for `test` run with no further arguments is:

```
test OPERATION

Options for OPERATION
add      Adds two numbers.
mult     Multiplies two numbers.
```

If we choose an operation, e.g. `add`, the output is:

```
test add OPERAND-A OPERAND-B

OPERAND-A     The first operand.
OPERAND-B     The second operand.
```

B.4 boxes

boxes is a pretty-printing library for laying out text in two dimensions. It is a direct port of the Haskell library boxes[4] by Brent Yorgey.

boxes' core data type is the `Box`, which has a width, a height and some contents. A box's contents can be text or other boxes. There are functions for creating boxes from text and for combining boxes into bigger boxes.

Creating Boxes

The `text` function can be used to create a box from a string, which will have height 1 and length N, where N is the length of the string (Nx1). `char` creates a 1x1 box containing a single character. `emptyBox` creates an empty box of arbitrary width and height.

`para :: Alignment -> Int -> String -> Box` creates a box from a string with a specific width. The box will be as high as necessary to fit the text, which is laid out according to the given alignment.

[4] `https://hackage.haskell.org/package/boxes`

Combining Boxes

The <> and <+> operators combine boxes horizontally with and without a column of space between both boxes, respectively. The // and /+/ operators are similar, but combine boxes vertically instead of horizontally. `hcat` and `vcat` are versions of <> and // that combine whole lists of boxes instead of two at a time. `hsep` and `vsep` are versions of <+> and /+/ that operate on lists, with a configurable amount of space between each two boxes. `punctuateH` and `punctuateV` also combine lists of boxes horizontally and vertically, but allow us to specify another box which is copied in between each two boxes.

The `align`, `alignVert` and `alignHoriz` functions can be used to create new boxes which contain other boxes in some alignment. `moveUp`, `moveLeft`, `moveDown` and `moveRight` move boxes by some amount inside larger boxes.

`table` creates a table from a list of rows and a list of widths for each column.

Rendering Boxes

The `render` function renders a box to a string. The `printBox` function prints a box to stdout.

C

Raw Performance Figures

This chapter contains the raw figures obtained for the performance measurements described in Sections 6.3 and 6.4, as well as instructions on how to reproduce them.

To run performance tests for the resolution algorithm as well as the API and behavior comparison algorithms, the CPM distribution contains the source code for a performance test program. This program can be built using `make buildperf` and executed via `bin/perftest`.

Using `perftest api -n NUMBER`, we can run the API comparison performance test, where `NUMBER` is the number of added, removed and changed functions and types to generate each. That is, running the API performance test with a `NUMBER` of 10 will generate 60 elements to be compared in total: 10 added, removed and changed functions and types each.

`perftest behavior -t NESTING -f FUNCTIONS` runs the behavior comparison performance test, where `NESTING` is the nesting depth of the generated type and `FUNCTIONS` is the number of functions to be compared. See Section 6.4 for an explanation of the type nesting depth.

`perftest resolution --packages=PKGS` runs the dependency resolution performance test on the specified packages. `PKGS` must be a list of comma-separated package identifiers, i.e., package names and versions separated by hyphens.[1] To run the resolution set, a set of package specifications is needed. The sample data set used in Sec-

[1] Pre-release versions are not supported by `perftest` at the moment.

tion 6.3 is provided in the `cpm-perf-test-data`[2] Git repository. The `packages.term` file from that repository needs to be available in the current directory when `perftest resolution` is run. To reproduce the exact tests from Section 6.3, run `perftest` on the following packages:

```
perftest resolution --packages=express-4.14.0,express-3.9.0,chalk-1.1.3,
    ↪ request-2.74.0,mocha-1.21.5,karma-1.2.0
```

[2] `https://git.ps.informatik.uni-kiel.de/joo/cpm-perf-test-data`

Table C.1.: Performance figures for resolution on KiCS2 0.5.1, all times in msec

Read Package Index	express-4.14.0	express-3.9.0	chalk-1.1.3	request-2.74.0	mocha-1.21.5	karma-1.2.0
12,573	5	240	0	16	4,421	did not finish
12,232	6	236	0	16	4,471	did not finish
12,756	6	243	1	19	5,140	did not finish
13,252	5	241	0	19	4,532	did not finish
12,067	4	241	0	17	4,632	did not finish
$\bar{x} = 12,573$	$\bar{x} = 5$	$\bar{x} = 241$	$\bar{x} = 0$	$\bar{x} = 17$	$\bar{x} = 4,532$	did not finish
$\sigma = 465.51$	$\sigma = 0.83$	$\sigma = 2.58$	$\sigma = 0.44$	$\sigma = 1.51$	$\sigma = 290.79$	did not finish

Table C.2.: Performance figures for resolution on PAKCS 1.14.1, all times in msec

Read Package Index	express-4.14.0	express-3.9.0	chalk-1.1.3	request-2.74.0	mocha-1.21.5	karma-1.2.0
-	330	30,160	30	1,810	did not finish	did not finish
-	370	23,360	10	1,540	did not finish	did not finish
-	340	24,420	10	1,490	did not finish	did not finish
-	350	25,070	20	1,500	did not finish	did not finish
-	350	25,600	10	1,430	did not finish	did not finish
-	$\bar{x} = 350$	$\bar{x} = 25,070$	$\bar{x} = 10$	$\bar{x} = 1,500$	did not finish	did not finish
-	$\sigma = 14.83$	$\sigma = 2,617.73$	$\sigma = 8.94$	$\sigma = 148.43$	did not finish	did not finish

Table C.3.: Performance figures for API diff on KiCS2 0.5.1, all times in msec

No. of functions	Run 1	Run 2	Run 3	Run 4	Run 5	$\bar{x}$	σ
10	2	1	2	1	3	2	0.8366
100	2	2	2	2	3	2	0.4472
1000	2	2	2	2	2	2	0
10000	Did not finish, aborted after 10 minutes						

Table C.4.: Performance figures for API diff on PAKCS 1.14.1, all times in msec

No. of functions	Run 1	Run 2	Run 3	Run 4	Run 5	$\bar{x}$	σ
10	120	50	60	60	70	60	27.7488
100	10	50	50	50	50	50	17.8885
1000	Out of memory while generating sample problem[a]						
10000	Out of memory while generating sample problem[a]						

[a] Aborted after 15 GB of memory usage.

Table C.5.: Performance figures for behavior diff on KiCS2 0.5.1, all times in msec

Type nesting depth	No. of functions	Run 1	Run 2	Run 3	Run 4	Run 5	$\bar{x}$	σ
10	10	290	303	287	286	299	290	7.5828
10	100	525	528	561	530	530	530	14.7885
10	1000	5093	5260	5428	5232	5256	5256	119.0806
100	10	454	463	457	460	466	460	4.7434
100	100	722	706	718	722	716	718	6.5726
100	1000	5636	5823	5637	5789	5666	5666	89.0937
1000	10	8768	9179	7929	8141	7955	8141	554.3138
1000	100	8454	8696	8692	8611	8564	8611	193.9961
1000	1000	17595	17835	17398	18261	17627	17627	328.3187

Table C.6.: Performance figures for behavior diff on PAKCS 1.14.1, all times in msec

Type nesting depth	No. of functions	Run 1	Run 2	Run 3	Run 4	Run 5	$\bar{x}$	σ
10	10	1490	1450	1400	1370	1520	1450	61.8869
10	100	6930	7460	9090	8290	7210	7460	883.8438
10	1000	Out of memory[a]						
100	10	16590	9780	10960	9160	13950	10960	3118.7930
100	100	36130	17490	18990	16470	24410	18990	8108.7927
100	1000	Out of memory[a]						
1000	10	Out of memory[a]						
1000	100	Out of memory[a]						
1000	1000	Out of memory[a]						

[a] Aborted after 15GB of memory usage.

D

User's Manual

The user's manual for the Curry package manager is printed verbatim on the following pages.

This document describes the Curry package manager (CPM), an application for distributing and installing Curry libraries.

D.1 Installing the Curry Package Manager

To install and use CPM, a working installation of either the PAKCS[1] compiler in version 1.14.1 or greater, or the KiCS2[2] compiler in version 0.5.1 or greater is required. Additionally, CPM requires *Git*[3], *curl*[4] and *unzip* to be available on the `PATH` during installation and operation. You also need to ensure that your Haskell installations reads files using UTF-8 encoding by default. Haskell uses the system locale charmap for its default encoding. You can check the current value using `System.IO.localeEncoding` inside a `ghci` session.

With your Curry compiler's `bin` directory on the `PATH`, enter the root directory of the CPM source distribution and type `make`. The application will be compiled using the Curry compiler on the `PATH` and a binary called `cpm` will be created in the `bin` subdirectory. Put this binary somewhere on your path.

Afterwards, run `cpm update` to pull down a copy of the central package index to your system.

D.2 Package Basics

Essentially, a Curry package is nothing more than a directory structure containing a `package.json` file and a `src` directory at its root. The `package.json` file is a JSON file containing package metadata, the `src` directory contains the Curry modules that make up the package.

We assume familiarity with the JSON file format. A good introduction can be found at http://json.org. The package specification file must contain a top-level JSON object with at least the keys `name`, `author`, `version`, `synopsis` and `dependencies`. More possible fields are described in section D.8. A package's name may contain any ASCII alphanumeric character as well as dashes (-) and underscores (_). It

[1] https://www.informatik.uni-kiel.de/ pakcs/
[2] https://www-ps.informatik.uni-kiel.de/kics2/
[3] http://www.git-scm.com
[4] https://curl.haxx.se

must start with an alphanumeric character. The author field is a free-form field, but the suggested format is either a name (`John Doe`), or a name followed by an email address in angle brackets (`John Doe <john.doe@goldenstate.gov>`). Separate multiple authors with commas.

Versions must be specified in the format laid out in the semantic versioning standard[5]: each version number consists of numeric major, minor and patch versions separated by dot characters as well as an optional pre-release specifier consisting of ASCII alphanumerics and hyphens, e.g. `1.2.3` and `1.2.3-beta5`. Please note that build metadata as specified in the standard is not supported.

The synopsis should be a short summary of what the package does. Use the `description` field for longer form explanations.

Dependencies are specified as a nested JSON object with package names as keys and dependency constraints as values. A dependency constraint restricts the range of versions of the dependency that a package is compatible to. Constraints consist of elementary comparisons that can be combined into conjunctions, which can then be combined into one large disjunction – essentially a disjunctive normal form. The supported comparison operators are $<, \leq, >, \geq, =$ and $\sim>$. The first four are interpreted according to the rules for comparing version numbers laid out in the semantic versioning standard. $\sim>$ is called the *semantic versioning arrow*. It requires that the package version be at least as large as its argument, but still within the same minor version, i.e. $\sim>$ 1.2.3 would match 1.2.3, 1.2.9 and 1.2.55, but not 1.2.2 or 1.3.0.

To combine multiple comparisons into a conjunction, separate them by commas, e.g. $\geq$ 2.0.0, $<$ 3.0.0 would match all versions with major version 2. Note that it would not match *2.1.3-beta5* for example, since pre-release versions are only matched if the comparison is explicitly made to a pre-release version, e.g. $=$ 2.1.3-beta5 or $\geq$ 2.1.3-beta2.

Conjunctions can be combined into a disjunction via the || characters, e.g. $\geq$ 2.0.0, $<$ 3.0.0|| $\geq$ 4.0.0 would match any version within major version 2 and from major version 4 onwards, but no version within major version 3.

[5] http://www.semver.org

D.3 Using Packages

Curry packages can only be used as dependencies of other Curry packages. Luckily creating a Curry package is easy, as we have seen in the previous section. So, to use a Curry package in your project, create a `package.json` file in the root, fill it with the minimum amount of information discussed in the previous session, and move your Curry code to a `src` directory inside your project's directory. Alternatively, if you are starting a new project, use the `cpm new <project-name>` command, which will ask you a few questions and then create a new project directory with a `package.json` file for you. Declare a dependency inside the new `package.json` file, e.g.:

```
{
  ...,
  "dependencies": {
    "json": "~> 1.1.0"
  }
}
```

Then run `cpm install` to install all dependencies of the current package and start your interactive Curry environment with `cpm curry`. You will be able to load the JSON package's modules.

Installing and Updating Dependencies

To install the current package's dependencies, run `cpm install`. This will install the most recent version of all dependencies that are compatible to the package's dependency constraints. Note that a subsequent run of `cpm install` will always prefer the versions it installed on a previous run, if they are still compatible to the package's dependencies. If you want to explicitly install the newest compatible version regardless of what was installed on previous runs of `cpm install`, you can use the `cpm upgrade` command to upgrade all dependencies to their newest compatible versions, or `cpm upgrade <package>` to update a specific package and all its transitive dependencies to the newest compatible version.

Note that there is also a `cpm update` command, which will update your copy of the central package index to the newest version. You can search the central package index via the `cpm search` command. See section D.7 for a reference of all commands.

Executing the Compiler

To use the dependencies of a package, the Curry compiler needs to be started via CPM, so that the compiler will know where to search for modules. You can use the `cpm curry` command to start the current Curry compiler (the `curry` command that is on the path). Any parameters given to `cpm curry` will be passed along verbatim to the Curry compiler, for example the following will start the Curry compiler, print `Hello, World` and then quit.

```
cpm curry :eval "1+1" :quit
```

To execute other Curry commands such as `currycheck` with the package's dependencies available, you can use the `cpm exec` command. `cpm exec` will set the `CURRYPATH` environment variable and then execute the command it is given.

Replacing Dependencies with Local Versions

During development of your applications, situations may arise in which you want to temporarily replace one of your package's dependencies with a local copy, without having to publish a copy of that dependency somewhere or increasing the dependency's version number. One such situation is a bug in a dependency not controlled by you: if your own package depends on package *A* and *A*'s current version is 1.0.3 and you encounter a bug in this version, then you might be able to investigate, find and fix the bug. Since you are not the the author of *A*, however, you cannot release a new version with the bug fixed. So you send off your patch to *A*'s maintainer and wait for 1.0.4 to be released. In the meantime, you want to use your local, fixed copy of version 1.0.3 from your package. The `cpm link` command allows you to replace a dependency with your own local copy.

`cpm link` takes a directory containing a copy of one of the current package's dependencies as its argument. It creates a symbolic link from that directory the the current package's local package cache. If you had a copy of *A-1.0.3* in the */src/A* − 1.0.3 directory, you could use `cpm link ~/src/A-1.0.3` to ensure that any time *A-1.0.3* is used from the current package, your local copy is used instead of the one from the global package cache. To remove any links, use `cpm upgrade` without any arguments, which will clear the local package cache. See section D.6 for more information on the global and local package caches.

D.4 Authoring Packages

If you want to create packages for other people to use, you should consider filling out more metadata fields than the bare minimum. See section D.8 for a reference of all available fields.

Semantic Versioning

The versions of published packages should adhere to the semantic versioning standard, which lays out rules for which components of a version number must change if the public API of a package changes. Recall that a semantic versioning version number consists of a major, minor and patch version as well as an optional pre-release specifier. In short, semantic versioning defines the following rules:

- If the type of any public API is changed or removed or the expected behavior of a public API is changed, you must increase the major version number and reset the minor and patch version numbers to 0.
- If a public API is added, you must increase at least the minor version number and reset the patch version number to 0.
- If only bug fixes are introduced, i.e. nothing is added or removed and behavior is only changed to removed deviations from the expected behavior, then it is sufficient to increase the patch version number.
- Once a version is published, it must not be changed.
- For pre-releases, sticking to these rules is encouraged but not required.
- If the major version number is 0, the package is still considered under development and thus unstable. In this case, the rules do not apply, although following them as much as possible as still encouraged. Release 1.0.0 is considered to be the first stable version.

To aid you in following these rules, CPM provides the `diff` command. `diff` can be used to compare the types and behavior of a package's public API between two versions of that package. If it finds any differences, it checks whether they are acceptable under semantic versioning for the difference in version numbers between

the two package versions. To use `diff`, you need to be in the directory of one of the versions, i.e. your copy for development, and have the other version installed in CPM's global package cache (see the `cpm install` command). For example, if you are developing version 1.3.0 of the JSON package and want to make sure you have not introduced any breaking changes when compared to the previous version 1.2.6, you can use the `cpm diff 1.2.6` command while in the directory of version 1.3.0.

CPM will then check the types of all public functions and data types in all exported modules of both versions (see the `exportedModules` field of the package specification) and report any differences and whether they violate semantic versioning. It will also generate a CurryCheck program that will compare the behavior of all exported functions in all exported modules whose types have not changed and execute that program. Note that not all functions can be compared via CurryCheck. In particular, functions taking other functions as arguments can not be checked, as well as functions taking `Float` arguments. There are a few other minor restrictions. Also note that the generated program may not terminate if one of the versions of the function does not terminate, for example if it generates an infinite list. In this case, you can mark those functions with the compiler pragma `{-# NOCOMPARE -#}` and CPM will not generate tests for them, e.g.

```
{-# NOCOMPARE #-}
ones :: [Int]
ones = 1 : ones
```

Publishing a Package

There are three things that need to be done to publish a package: make the package accessible somewhere, add the location to the package specification, and add the package specification to the central package index.

CPM supports ZIP files accessible over HTTP as well as Git repositories as package sources. You are free to choose one of those, but a publicly accessible Git repository is preferred. To add the location to the package specification, use the `source` key. For a HTTP source, use:

```
{
  ...,
  "source": {
    "http": "http://example.com/package-1.0.3.zip"
```

```
  }
}
```

For a Git source, you have to specify both the repository as well as the revision that represents the version:

```
{
  ...,
  "source": {
    "git": "git+ssh://git@github.com:john-doe/package.git",
    "tag": "v1.2.3"
  }
}
```

There is also a shorthand, `$version`, available to automatically use a tag consisting of the letter `v` followed by the current version number, as in the example above. Specifying `$version` as the tag and then tagging each version in the format `v1.2.3` is preferred, since it does not require changing the source location in the `package.json` file every time a new version is released.

After you have published the files for your new package version, you have to add the corresponding package specification to the central package index. The central package index is just a Git repository containing a directory for each package, which contain subdirectories for all versions of that package which in turn contain the package specification files. So the specification for version 1.0.5 of the `json` package would be located in `json/1.0.5/package.json`. If you have access to the Git repository containing the central package index, then you can add the package specification yourself. If you do not have access, then send the file to someone who does.

D.5 Configuration

CPM can be configured via the `$HOME/.cpmrc` configuration file. The following list shows all configuration options and their default values.

`package_install_path` The path to the global package cache. This is where all downloaded packages are stored. Default value: `$HOME/.cpm/packages`

`repository_path` The path to the index repository. Default value: `$HOME/.cpm/index`.

D.6 Some CPM Internals

CPM's central package index is a Git repository containing package specification files. A copy of this Git repository is stored on your local system in the `$HOME/.cpm/index` directory, unless you changed the location using the `repository_path` setting. CPM uses the package index when searching for and installing packages and during dependency resolution.

When a package is installed on the system, it is stored in the global package cache. By default, the global package cache is located in `$HOME/.cpm/packages`. When a package *foo*, stored in directory `foo`, depends on a package *bar*, a link to *bar's* directory in the global package cache is added to *foo's* local package cache when dependencies are resolved for *foo*. The local package cache is stored in `foo/.cpm/package_cache`. Whenever dependencies are resolved, package versions already in the local package cache are preferred over those from the central package index or the global package cache.

When a module inside a package is compiled, packages are first copied from the local package cache to the runtime cache, which is stored in `foo/.cpm/packages`. Ultimately, the Curry compiler only sees the package copies in the runtime cache, and never those from the local or global package caches.

D.7 Command Reference

This section gives a short description of all available CPM commands. In addition to the commands listed here, there is a global `--`*verbosity* parameter which defaults to *info* but can be increased to *debug* for more output.

`info` Gives information on the current package, e.g. the package's name, author, synopsis and its dependency specifications.

`info` *package* Gives information on the newest known version of the given package.

`info` *package version* Gives information on the given package version.

`search` *query* Searches the names and synopses of all packages in the central package index for a term.

`update` Updates the local copy of the central package index to the newest available version.

`install` Installs all dependencies of the current package.

`install` *package* `[--`*pre*`]` Installs the newest version of a package to the global package cache. `--`*pre* enables the installation of pre-release versions.

`install` *package version* Installs a specific version of a package to the global package cache.

`install` *package*`.zip` Installs a package from a ZIP file to the global package cache.

`uninstall` *package version* Uninstalls a specific version of a package from the global package cache.

`upgrade` Upgrades all dependencies of the current package to the newest compatible version.

`upgrade` *package* Upgrades a specific dependency of the current package and all its transitive dependencies to their newest compatible versions.

`deps` Does a dependency resolution run for the current package and prints out the results. The result is either a list of all package versions chosen or a description of the conflict encountered during dependency resolution.

`diff` Test

`exec` *command* Executes an arbitrary command with the `CURRYPATH` environment variable set to the paths of all dependencies of the current package. Can be used to execute `currycheck` with dependencies available, for example.

`curry` *args* Executes the Curry compiler with the dependencies of the current package available. Any arguments are passed verbatim to the compiler.

`link` *source* Can be used to replace a dependency of the current package using a local copy, see D.3.3 for details.

`new` Asks a few questions and creates a new package.

D.8 Package Specification Reference

This section describes all metadata fields available in a CPM package specification. Mandatory fields are marked with a * character.

`name*` The name of the package. Must only contain ASCII letters, digits, hyphens and underscores. Must start with a letter.

`version*` The version of the package. Must follow the format for semantic versioning version numbers.

`author*` The package's author. This is a free-form field, the suggested format is either a name or a name followed by an email address in angle brackets – e.g. `John Doe <john@doe.com>`. Multiple authors should be separated by commas.

`maintainer` The current maintainers of the package, if different from the original authors. This field allows the current maintainers to indicate the best person or persons to contact about the package while attributing the original authors.

`synopsis` A longer form description of what the package does.

`license` The license under which the package is distributed. This is a free-form field. In case of a well-known license such as the GNU General Public License[6], the SPDX license identifier[7] should be specified. If a custom license is used, this field should be left blank in favor of the license file field.

`licenseFile` The name of a file in the root directory of the package containing explanations regarding the license of the package or the full text of the license. The suggested name for this file is `LICENSE`.

`copyright` Copyright information regarding the package.

`homepage` The package's web site. This field should contain a valid URL.

`bugReports` A place to report bugs found int he package. The suggested formats are either a valid URL to a bug tracker or an email address.

`repository` The location of a SCM repository containing the package's source code. Should be a valid URL to either a repository (e.g. a Git URL), or a website repre-

[6]https://www.gnu.org/licenses/gpl-3.0.en.html
[7]https://spdx.org/licenses/

senting the repository.

`dependencies*` The package's dependencies. This must be JSON object where the keys are package names and the values are version constraints. See section D.2.

`compilerCompatibility` The package's compatibility to different Curry compilers. Expects a JSON object where the keys are compiler names and the values are version constraints. Currently, the supported compiler names are `pakcs` and `kics2`. If this field is missing or contains an empty JSON object, the package is assumed to be compatible to all compilers in all versions.

`source` A JSON object specifying where the version of the package described in the specification ca be obtained. See section D.4.2 for details.

`exportedModules` A list of modules intended for use by consumers of the package. These are the modules compared by the `cpm diff` command. Note that modules not in this list are still accessible to consumers of the package.

Bibliography

[Bra+11] Bernd Braßel et al. "KiCS2: A New Compiler from Curry to Haskell". In: *Functional and Constraint Logic Programming* 6816 (2011), pp. 1–18. DOI: http://dx.doi.org/10.1007/978-3-642-22531-4_1.

[Bra14] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. Internet Engineering Task Force, 2014. URL: https://tools.ietf.org/html/rfc7159.

[Bur05] Daniel Burrows. "Modelling and Resolving Software Dependencies". In: (2005), pp. 1–16.

[DLP10] Eelco Dolstra, Andres Löh, and Nicolas Pierron. "NixOS: A purely functional Linux distribution". In: *Journal of Functional Programming* 20.5-6 (2010), pp. 577–615. ISSN: 0956-7968. DOI: 10.1017/S0956796810000195.

[Han+16a] M. Hanus et al. *Curry, An Integrated Functional Logic Language, Version 0.9.0*. 2016.

[Han+16b] M. Hanus et al. *The Kiel Curry System (Version 2), Version 0.5.1*. 2016.

[Han02] M Hanus. "CurryDoc: A Documentation Tool for Declarative Programs". In: *Proceedings WFLP '02* 2002 (2002), pp. 225–228.

[Han16] Michael Hanus. "CurryCheck : Checking Properties of Curry Programs". In: *Pre-proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*. 2016. DOI: http://arxiv.org/abs/1608.05617.

[Jon05] Isaac Jones. "The Haskell Cabal, a common architecture for building applications and libraries". In: (2005), pp. 1–22.

[Kat10] Yehuda Katz. *Some of the Problems Bundler Solves*. 2010. URL: http://yehudakatz.com/2010/04/12/some-of-the-problems-bundler-solves/ (visited on 07/25/2016).

[Les13] Joe Leslie-Hurd. "Maintaining verified software". In: *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell - Haskell '13* (2013), p. 71. ISSN: 15232867. DOI: 10.1145/2503778.2503787. URL: http://dl.acm.org/citation.cfm?doid=2503778.2503787.

[Löh11] Andres Löh. *A New Dependency Solver for cabal-install*. 2011. (Visited on 09/01/2016).

[NT01] Thomas Nordin and Andrew Tolmach. "Modular lazy search for Constraint Satisfaction Problems". In: *Journal of Functional Programming* 11.05 (2001), pp. 1–16. ISSN: 0956-7968. DOI: 10.1017/S0956796801004051.

[Pey03] Simon Peyton. "Haskell 98 Language and Libraries The Revised Report". In: *Language* 13 (2003), pp. 1–277. ISSN: 0956-7968. DOI: 10.1017/S0956796803000315.

[Pre16] T. Preston-Werner. *Semantic Versioning 2.0.0*. 2016. URL: http://semver.org.

[RN03] S J Russell and P Norvig. *Artificial Intelligence: A Modern Approach*. 2003.

[Sip12] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012. ISBN: 9781133187790. URL: https://books.google.de/books?id=ekE2ngEACAAJ.