Towards a Theory of Packages

Mark Florisson and Alan Mycroft

Computer Laboratory, University of Cambridge
JJ Thomson Avenue, Cambridge CB3 0FD, UK
Firstname.Lastname@cl.cam.ac.uk

Abstract. Package managers are programs that manage the installation of software packages onto the systems of users with the help of *repositories* and package *metadata*. This process is complicated by the fact that packages evolve, and are therefore *versioned*. Traditional package managers cause trouble for users by refusing to install a package, or by failing to compile, load or run programs correctly. We argue that this is the result of inexpressive package metadata and limitations inherent in package *linkers*. We present a package system that ensures installability and type-safety of packages by using interfaces for package metadata, by addressing linker insufficiencies and by imposing restrictions on repositories. The system manages change flexibly and meaningfully using names and nominal subtyping (instead of version ranges) to formalize backwards compatibility. Although the system is similar to module systems, we show how versioning leads to fundamentally different design decisions. The system is applicable to decentralized package systems administered through a centralized but unaudited package repository (e.g. hackage, opam, pypi, npm, etc).

1 Introduction

Large systems are not written monolithically, but are composed of various software components. These components (either library code or executable programs) are usually called *packages*, often comprised internally of many *modules*. Packages are a means of code distribution, so while modules can be modified in-place, packages are versioned to support their evolution. Package metadata expresses how packages *depend* on each other, often through *version ranges*. This metadata is used in the *management* of packages (installation, upgrade or removal of packages). Packages are shared through (central) *repositories*, and dependency *resolution* decides which packages from the repository need to be available on a user's system when installing or upgrading. Due to the inexpressiveness of package metadata, two major problems arise.

Dependency hell is the first problem, and arises because traditional package systems insist on making only a *single version* of a particular package available

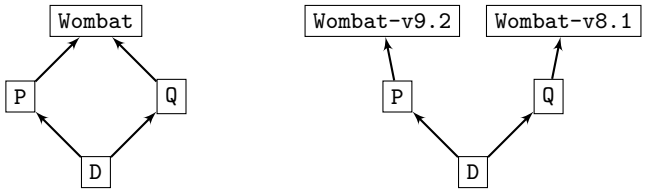

Fig. 1. Diamond dependency with a shared `Wombat` package (left) and with two separate `Wombat` packages (right).

at any given time. This leads to *conflicts*, as different packages on the system may require different versions of a dependency[1].

Systems allow only a single version of a package for two reasons. First, package linkers often cannot link multiple versions into a single executable (and sometimes cannot deal with multiple versions system-wide).[2] Second, linking multiple versions of a package into a program may result in the program passing values from one version of the library into another version, potentially violating *data abstraction* and *type safety*.

To see this, consider a scenario involving a common dependency `Wombat` as shown in Figure 1. As part of their public APIs, `Wombat` *exports* an abstract data type `TWombat`, `P` exports a function **fun** f : `TWombat` $\rightarrow$ `Int` and `Q` exports a function **fun** g : `Int` $\rightarrow$ `TWombat`. Because `TWombat` is part of the function signature we say that `P` and `Q` *expose* `TWombat` from `Wombat`. Due to this exposure, `D` could now pass values of `TWombat` between `P` and `Q` (e.g. $P.f \circ Q.g$). If that happens, safety mandates that `P` and `Q` are linked against the same `Wombat` package (Figure 1, left). On the other hand, if `D` does not wish to share `TWombat` values between `P` and `Q`, we are entirely free to link `P` and `Q` against different `Wombat` packages (Figure 1, right).

Unfortunately, installation problems are unpredictable, as dependency resolution is influenced by both the required software *and* the already available packages on the system; hence a solution to an installation problem may exist for one system but not another. Versions conflicts are common in certain systems, as version ranges may be overly constrained to guard against possible future incompatibilities. Conflicts are a necessary result of the decentralised nature of package systems and the modular definitions of packages, as not every package may be fully "up-to-date" with its dependencies. To compose packages into correct systems we must establish consensus on package versions. [3]

[1] We follow common nomenclature: if package P depends on Q (written $P \rightarrow Q$), then both the arrow and Q are called a "dependency" of P on Q.

[2] Linking multiple versions of native binaries may cause symbol conflicts, and virtual machines often simply link the first package found in a path (e.g. PYTHONPATH).

[3] In centrally audited package systems such as Debian, conflicts may also arise due to genuine system-level incompatibility (e.g. two competing window managers). However, we focus on unaudited systems that typically have no way of expressing such type of conflicts.

Portability is the issue of whether a package developed on one machine can also work correctly on another—even of the same type— and is the second major problem with existing package systems. During development, packages are linked against dependencies as if the dependencies were modules (systems comprised of modules contain only one version of each module). This implicitly assumes a *stable* (non-changing) interface between package versions, which is often not the case. As a result, installing the package on a different system may link it against different versions of dependencies with perhaps different interfaces. This effectively delegates the issue of type-safety to the (language-specific) linking procedure (linkers for languages such as C typically resolve symbols by name, ignoring types completely). Unfortunately, because there are exponentially many combinations of dependency versions, checking for incompatibilities in package metadata beforehand may not be tractable, leaving users with mysterious errors.

A Fresh Approach Existing package managers are pragmatic tools and our purpose is to provide a theoretical basis for packages and their management. For this, we introduce a new package system Π, that is meant to represent a *class* of package systems that solve the aforementioned problems: dependency hell is eliminated by installing package versions *side-by-side*. This is made possible by more expressive package metadata that can express where package versions must be *shared*. Portability and type-safety issues are eliminated by writing packages against interfaces. Packages are type-checked and compiled *modularly* and interfaces are used for dependency resolution and linking.

Finally, we address *change management*, which is especially crucial for packages because of the distributed nature of their development. Packages do not evolve in lockstep, and breaking compatibility with respect to previous versions diminishes the ability to share packages. In the worst case, this can result in the programmer having to write explicit adaptation layers between package versions.

Contributions include a package system Π that:

1. does not suffer from dependency hell, portability or type-safety issues
2. can compile packages separately, and link packages statically or dynamically
3. helps manage change and handles versioning

Further, we show that dependency resolution is decidable in polynomial time, whereas dependency resolution in traditional package managers is NP-complete [10] (§4.1).

There are great similarities between module systems such as that of ML [18] and package systems such as Π. Indeed, recent work on Backpack [20] proposes a package system for Haskell based on *mixin modules* from the MixML module calculus [15]. However, we argue that package systems are different from modules, and we use *abstraction* (such as functors in ML) to express *sharing* of dependencies, rather than *multiple instantiation*. Our presentation differs from Backpack in that we explicitly address *dependency hell* and *change management*. Further differences are detailed in §9.

2 Package System and Examples

Our package system explicitly handles the two scenarios from Figure 1 by distinguishing between *external* (public) dependencies and *internal* (private) dependencies. A dependency is external when we *expose* one of its abstract data types as part of our public API, (e.g. **fun** f : `TWombat` $\rightarrow$ `Int` in package `P`). A dependency is internal when we do not expose any of its abstract data types.

External Dependencies We express external dependencies using *abstraction* (formal parameters), to defer linking of the dependency. A dependent such as `D` from Figure 1 can then choose a suitable version of `Wombat` that is compatible with the needs of both `P` and `Q`. This effectively shifts compatibility constraints from package systems to the package language itself. Package `D` is now assured that it can safely share values of `TWombat` between `P` and `Q`. If `P` and `Q` were to link with `Wombat` individually, `D` would not know whether it could safely share values, as `P` and `Q` could choose different versions of `Wombat`.

Internal Dependencies Internal dependencies are expressed through an **import** operator, which loads *some* compatible version of the dependency. Abstract data types from internal dependencies cannot be exposed by the importing package, but can be hidden behind new abstract data types defined by the importing package. The grammar and scoping rules of our system automatically ensure this, as interfaces have no way of referring to internal dependencies (there is no **import** construct at the interface level).

Example In our system packages are defined as follows:

```
pkg Wombat-v9.2 impl IW-v2
  type TWombat = ...
  ...
```

```
iface IW-v2 <: IW-v1
  type TWombat
  ...
```

On the left we have a package definition `Wombat` of version `v9.2` implementing an interface `IW-v2` (shown on the right). It further defines an abstract data type `TWombat`, which is declared in `IW-v2`, making it a part of the public API (i.e. visible to other packages). Interface `IW-v2` declares that it is further a subtype of `IW-v1` (not shown here). Next we have a package `P`:

```
pkg P-v9(W : IW-v1) impl IP-v1
  fun f : W.TWombat → Int = ...
```

```
iface IP-v1(W : IW-v1)
  fun f : W.TWombat → Int
```

`P` has a formal parameter `W` of type `IW-v1` to abstract over the `Wombat` dependency (or indeed any package that implements `IW-v1` or a subtype). The package `P` (left) defines f and the interface `IP-v1` (right) declares f and its function signature. The interface must take the same parameters as the package, and functions declared in the interface must have the same type signatures as in the implementing package. This way interfaces fully describe the public API of the package (specifically, the type of the package must be a subtype of the implemented interface, covered in §6). Now `IP-v1` can expose `TWombat` through its `W` parameter. In §7 we shall argue for the use of multiple interfaces and more

flexible management of interface parameters. Package `Q` is similar to `P` so we omit it here. Finally there is package `D`:

```
pkg D–v8 impl ID–v2                    iface ID–v2
  import Wombat–any : IW–v2 as W
  import P–any : IP–v1 as P(W)
  import Q–any : IQ–v6 as Q(W)
  ...                                    ...
```

`D` first imports `Wombat`, and then *instantiates* (or *links*) `P` and `Q` with `Wombat`. Only `D` is in a position to say which `Wombat` `P` and `Q` must be linked with, and it could also choose to link `P` and `Q` with different packages, which would prevent sharing of the `TWombat` type. This is useful whenever there is no version of `Wombat` compatible with both `P` and `Q`. This can happen if for example `P` requires an older version of `Wombat`'s API, but `Q` requires the latest version of the API which is backwards-incompatible with older versions.

Note further that in our example interface `IW-v2` is a subtype (<:) of `IW-v1`, allowing us to exploit subsumption when linking `P` against `Wombat`.

3 Package Language

We now introduce the grammar of our package system. In our system, a package is an opaque unit of code, perhaps internally comprised of modules, but for simplicity our packages export only flat namespaces.

3.1 Naming Convention

In Π both packages and interfaces are versioned (`Wombat-v9.2` and `IW-v2` respectively). Packages are resolved by requesting some version (`Wombat-any`) implementing a particular interface (hence we write package versions when defining a package but not when importing a package). Importing a package *binds* to a local package variable (like a let-binding), and abstraction to a local parameter (like a formal parameter to a λ).

While names are disambiguated by syntax, we find it helpful to define separate syntactic categories and naming conventions for them:

Category	Example	Use
$\mathcal{P}$	`Wombat-v9.2`	a package name in a repository (versioned or unversioned)
$\mathcal{I}$	`IWombat-v3`	a concrete interface in a repository
X, Y	`Wombat`	a package parameter or local name for an imported package
T	`TWombat`	an abstract data type name
f	`f`, `g`	a function name
x	`x`, `y`	a core-language variable

In examples we use lower-case for variable and functions, upper case starting with 'T' for types, upper case beginning with 'I' for interfaces and other upper case

for packages (both concrete and abstract). The format of the 'version' part of a name plays no semantic role other than to distinguish packages and interfaces; however we assume there is a partial order $\leq$ on the string after the '-v' in $\mathcal{P}$ and $\mathcal{I}$ to define backwards compatibility later.

Syntactic meta-notation: given a string ω of terminal and non-terminal symbols we write $\overline{\omega}$ to to mean zero or more repetitions, and $[\omega]$ to mean zero or one repetition of ω. Here, rather abusively, we assume such abstract-syntax repetition includes appropriate concrete separators such as commas in argument lists and semicolons for declarations.

3.2 Grammar

Package definitions *Pdef* and interface definitions *Idef* have largely parallel syntax; their bodies consist of (type and function) definitions and declarations respectively. The **impl** clause serves to *seal* the package body behind the given interface, so that only the contents listed by the interface are exposed from the package. Typing rules defined in §6 ensure that the type of the package body is a subtype of the interface body. Interfaces may optionally declare themselves a subtype of another interface through the <: clause in interface definitions.

Packages may be parameterised over external dependencies, constrained by interfaces—which must be fully applied to appropriate arguments.

$$\begin{array}{lll} \mathit{Pdef} & ::= \mathbf{pkg}\ \mathcal{P}(\overline{X : \mathcal{I}(\overline{Y})})\ \mathbf{impl}\ \mathcal{I}\ \{\overline{\mathit{PImport}}; \overline{\mathit{def}}\} & \textit{(pkg definition)} \\ \mathit{Idef} & ::= \mathbf{iface}\ \mathcal{I}(X : \mathcal{I}(\overline{Y}))\ [\ <: \mathcal{I}\ ]\ \{\overline{\mathit{decl}}\} & \textit{(interface definition)} \end{array}$$

In our core language, code consists of abstract data types and functions that operate over such types. Abstract data types can only be exported abstractly in interfaces, but are concrete within the package (i.e. their definition is visible to the code in the package). Hence packages contain *definitions* (*def*), and interfaces contain corresponding *declarations* (*decl*).

$$\begin{array}{lll} \mathit{def} & ::= \mathbf{type}\ T = \tau \mid \mathbf{fun}\ f : \tau = e & \textit{(definition)} \\ \mathit{decl} & ::= \mathbf{type}\ T \mid \mathbf{fun}\ f : \tau & \textit{(declaration)} \end{array}$$

Terms and types consist of the simply-typed lambda calculus with projections of functions and abstract data type components (using standard "dot" notation) and abstract data types. However, just as the ML module system [23], the package language is largely agnostic of the core language.

$$\begin{array}{lll} e & ::= f \mid X.f \mid \mathit{wrap}_T(e) \mid \mathit{unwrap}_T(e) \mid \lambda x : \tau.e \mid e\ e \mid x & \textit{(terms)} \\ \tau & ::= T \mid X.T \mid \tau \to \tau \mid \mathit{Unit} \mid \mathit{Int} \mid \ldots & \textit{(types)} \end{array}$$

We assume the core language also contains wrap_T and unwrap_T operators, necessary for showing that types are preserved during evaluation. Consider the following example of a function f defined in some package P:

$$\mathbf{fun}\ f : \mathit{Int} \to T = \lambda x : \mathit{Int}.x$$

Here $P.f$ is evaluated by extracting $\lambda x : \mathit{Int}.x$, but this term is of type $\mathit{Int} \to \mathit{Int}$, and not of type $\mathit{Int} \to T$. In other words, the type of the expression depends on which package it is typed in. To handle such scenarios, wrap_T wraps a value into an abstract data type value, and unwrap_T unwraps it (such that $\mathit{unwrap}_T(\mathit{wrap}_T(v)) = v$). We can now update the function definition for f:

$$\textbf{fun } f : \mathit{Int} \to T = \lambda x : \mathit{Int}.\mathit{wrap}_T(x)$$

In package definitions we saw a non-terminal *Pimport* for package imports:

$$\mathit{Pimport} ::= \textbf{import } \mathcal{P}\text{-}\mathit{any} : \mathcal{I} \textbf{ as } X(\overline{Y}) \qquad \textit{(pkg import)}$$

As mentioned in §2, package imports are only allowed in package definitions, and not in interface definitions. This ensures that internal dependencies cannot be referred to at the interface level, and hence abstract data types defined by internal dependencies cannot be exported by the importing package. An import has the effect of loading a package dependency by looking up the dependency from a repository of locally available packages (covered in §4.1). Any package named $\mathcal{P}$ satisfying interface $\mathcal{I}$ can be used for this purpose.

Finally, we need to be able to define programs. We define programs as packages that take no parameters and export a *main* function definition in their interface, of the form **fun** $\mathit{main} : \mathit{Unit} \to \tau$. Programs are then evaluated by first linking the program package (covered in §5.1) and subsequently evaluating the expression $P.\mathit{main}()$ (covered in §5.2).

3.3 Extended Syntax

In this section we extend our syntax with *package expressions* and *package types*. Package expressions are needed in our small-step operational semantics, to represent part-evaluated phrases (cf. using closures when reducing the λ-calculus). Package expressions evaluate to *package values*, which are the ML equivalent of *structures*. Package types are needed to represent interface applications in our static semantics. Packages types are the ML equivalent of *signatures*.

Package Expressions and Package Values The semantics of a package variable X is conceptually the package body to which it is bound—but where any imports have been recursively replaced—in other words a *Pbody* of the form $\{\overline{\mathit{def}}\}$. In our small-step semantics §5.2 we require a syntax for intermediate forms during reduction—package *expressions*, which are like *Pbody* terms, but may contain other package expressions representing part-complete resolution of imports. Package expressions have the following syntax:

$$\mathit{Pexpr} ::= \{\overline{\mathit{Pexpr}};\ \overline{\mathit{Pimport}};\ \overline{\mathit{def}}\}^X_{\mathcal{I}} \qquad \textit{(package expressions)}$$

The annotations $\mathcal{I}$ and X are needed to assign package types to package expressions: the interface name $\mathcal{I}$ is used to properly define a nominal subtyping relation (§6.3), and the package variable X is used to reason about type equivalence (§6.2). This is covered in more detail below.

Our semantics uses a depth-first left-right traversal strategy for import resolution, meaning that a *Pexpr* contains at most one internal *Pexpr* representing a part-resolved import. *Package values* are now the special case of *Pexpr* of the form $\{\overline{def}\}^X_{\mathcal{I}}$.

Our semantics substitutes package values (and not arbitrary package expressions) for package variables. There are three syntactic forms where package variables may occur: type projections, term projections, and package applications in import statements. To allow for this substitution we augment the syntax of expressions e, types τ and imports *Pimport* to allow their package name components X to be package expressions. In particular, during reduction type projections $X.T$ may be of the form $\{\overline{def}\}^X_{\mathcal{I}}.T$ and term projections $X.f$ may be of the form $\{\overline{def}\}^X_{\mathcal{I}}.f$. During reduction imports may be of the form

$$\textbf{import}\ \mathcal{P}\text{-}any : \mathcal{I}'\ \textbf{as}\ Y(\overline{\{\overline{def}\}^X_{\mathcal{I}}}).$$

Package Types To represent interface applications, and to assign types to package expressions, we use *package types*:

$$Ptype ::= \{\overline{decl}\}^X_{\mathcal{I}} \qquad \textit{(package types)}$$

Note that package types, like package values, include not only the signature $\overline{decl}$, but also the package name X being being used and the declared interface $\mathcal{I}$ it implements.

The interface annotation $\mathcal{I}$ is required by the definition of our nominal subtyping relation. We use this in particular check whether interface and package applications are properly typed.

The package variable label X is needed to preserve the access paths to the abstract data types defined in the package. Such paths are important in our definition of type equivalence (written $\approx$): $X.T \approx Y.T$ iff $X = Y$. Hence access paths help us distinguish abstract data types with the same name, but defined by different imported packages. Imports in Π are essentially *generative*, which means that each imported package contains fresh abstract data types — separate from any other abstract data types in the system. Distinguishing abstract data types by access path is a technique known as *manifest types* [22]. However, because we do not support hierarchical packages or modules (e.g. $P_1.P_2$) our access paths are simpler and consist of a single package variable name (e.g. P_1).

4 Dependency Resolution and Linking

In this section we cover how repositories are constructed, and the restrictions that make dependency resolution a simple and efficient (polynomial-time) depth-first traversal. This is an improvement over existing package systems, which often do not have polynomial-time algorithms for dependency resolution [10]. The restrictions further ensure that dependency hell (unsatisfiability errors) cannot occur. Repositories are also an essential part of our small-step operational semantics for packages, developed in §5.1: repositories ensure that all package dependencies are available and that linking terminates by restricting cyclic dependencies.

4.1 Satisfiability

Repositories are a collection of packages and interfaces, subject to certain restrictions (to make this explicit, we sometimes call such repositories *well-formed*). We define repositories $\mathcal{R}$ to be a sequence of package definitions *Pdef* and interface definitions *Idef*:

$$\mathcal{R} ::= \overline{Pdef} \quad \overline{Idef}$$

We further say that $\mathcal{P} : \mathcal{I} \in \mathcal{R}$ (for some versioned or unversioned $\mathcal{P}$) if $\textbf{pkg}\ \mathcal{P}\ \textbf{impl}\ \mathcal{I}\ \{\ldots\} \in \mathcal{R}$, or if $\mathcal{R} \vdash \mathcal{I}' <:_{nom} \mathcal{I}$ and $\mathcal{P} : \mathcal{I}' \in \mathcal{R}$. That is, either $\mathcal{P}$ directly implements $\mathcal{I}$, or $\mathcal{P}$ implements some nominal subtype $\mathcal{I}'$ of $\mathcal{I}$ (see §6.3 for our definition of subtyping). Finally, we say that $\mathcal{I} \in \mathcal{R}$ whenever $\textbf{iface}\ \mathcal{I}\ [<: \mathcal{I}']\ \{\ldots\} \in \mathcal{R}$.

The main restriction on package repositories is the *satisfiability* restriction, which states that our dependency resolution algorithm can determine a suitable set of well-typed package dependency versions for *any* package in the repository. For users of the package systems this means any package in $\mathcal{R}$ can be installed onto their system. We formalise this with inference rules that define well-formedness. We start by defining well-formedness for repository extension, and for simplicity assume that each package takes a single parameter and has a single import:

$$\frac{\mathcal{R}\ \texttt{WF} \qquad \mathcal{I} \in \mathcal{R} \qquad \mathcal{I}_X \in \mathcal{R} \qquad \mathcal{P}_Y : \mathcal{I}_Y \in \mathcal{R} \qquad \mathcal{R} \vdash \mathcal{P}\ \texttt{WF}}{\mathcal{R}[\textbf{pkg}\ \mathcal{P}(X : I_X)\ \textbf{impl}\ \mathcal{I}\ \{\textbf{import}\ \mathcal{P}_Y : \mathcal{I}_Y\ \textbf{as}\ Y(X); \mathit{body}\}]\ \texttt{WF}} \text{ (Repo-Pkg)}$$

The rule ensures that any interfaces mentioned by the package are available in $\mathcal{R}$, and further than any of its internal dependencies (imports) are satisfiable in $\mathcal{R}$. Finally, the rule requires that the package itself is well-formed (see below).

Adding interfaces to a repository is similar: any interfaces that are mentioned must already be available in the repository, and the interface itself must be well-formed with respect to $\mathcal{R}$:

$$\frac{\mathcal{R}\ \texttt{WF} \qquad \mathcal{I}_X \in \mathcal{R} \qquad \mathcal{R} \vdash \mathcal{I}\ \texttt{WF}}{R[\textbf{iface}\ \mathcal{I}(X : I_X) <: \mathcal{I}'\ \{\overline{decl}\}]\ \texttt{WF}} \qquad \text{(Repo-IFace)}$$

Well-formedness of packages and interfaces is covered in §6.4. Briefly, for packages it requires that they are well-typed, and a subtype of the interface they claim to implement. Well-formedness of interfaces also requires that the nominal subtyping relation holds structurally. An additional restriction is that package dependencies may not be circular, as this poses some problems for our operational semantics and proofs. For example, if a package `Foo` imports a package `Bar`, then `Bar` may not also import `Foo`. We reject such cycles *conservatively* with the help of an conservative package dependency graph $G = (V, E)$, where:

$$V = \{\mathcal{P} : \mathcal{I} \mid \mathcal{P} : \mathcal{I} \in \mathcal{R}\}$$

$$E = \{(A : \mathcal{I}_A, B : \mathcal{I}_B) \mid A\texttt{-vX} : \mathcal{I}_A \in \mathcal{R} \wedge A \text{ imports } B : \mathcal{I}_B\}$$

Cycles within this graph then indicate that there is a particular combination of package versions which would contain a dependency cycle, and we treat all such cycles as errors (indicating ill-formedness of the repository). There are similar restrictions on circularity between interfaces, where the nodes are interface constructors $\mathcal{I}$, and there is an edge between two nodes $\mathcal{I}_1$ and $\mathcal{I}_2$ whenever $\mathcal{I}_1$ mentions $\mathcal{I}_2$.

We can now prove that dependency hell does not exist in our system, by proving that any $\mathcal{P} : \mathcal{I} \in \mathcal{R}$ has all its dependencies recursively satisfied whenever $\mathcal{R}$ `WF` (Theorem 1).

Theorem 1. *Let P be a package **pkg** $\mathcal{P}$ **impl** $\mathcal{I}$ $\{\ldots\}$ in a repository $\mathcal{R}$. Then P is satisfiable in $\mathcal{R}$ whenever $\mathcal{R}$ `WF`.*

Proof. Since $\mathcal{P} : \mathcal{I} \in \mathcal{R}$, we know from (Repo-Pkg) that there exists a well-formed repository $\mathcal{R}'$ containing a subset of package and interface definitions such that $\mathcal{R}'[P]$ `WF`. From (Repo-Pkg) we also know that $P' \in \mathcal{R}'$ for any immediate dependee P' of P. By assumption each P' is itself satisfiable in $\mathcal{R}'$, and hence P is satisfiable in $\mathcal{R}'[P]$. Therefore P is also satisfiable in $\mathcal{R}$. □

The above theorem is our argument that dependency hell does not arise in our system. Intuitively, this makes sense, because we have expressed dependencies in terms of interfaces, and have expressed sharing of dependencies through abstraction. So when we see an **import** we are free to choose *any* compatible version from $\mathcal{R}$, without regard for other packages or the current state of the system. Conflicts, and therefore dependency hell, are simply not possible. When conflicts are not possible, satisfiability checking and dependency resolution become trivial operations.

Next we turn to the idea of *installation plans* that shows more practically how users can install packages onto their system.

Plans Repositories are used for sharing packages with others, and may consist of many packages. The point of an *installation plan* is to install some $P \in \mathcal{R}$ and all of its dependencies. Such installation plans are always computable (from Theorem 1 we know that we can always choose $\mathcal{R}$ itself), but $\mathcal{R}$ may be very large, so we wish to compute some repository $\mathcal{R}'$ that is preferably smaller than $\mathcal{R}$:

Definition 1. *An installation plan for a package $\mathcal{P}$, with respect to a package repository $\mathcal{R}$, is some well-formed repository $\mathcal{R}'[P]$ containing a subset of the package and interface definitions of $\mathcal{R}$.*

First, we note that nominal subtyping is used for package dependency resolution. Nominal subtyping can be decided in polynomial time as it is just a relation between names defined explicitly by the interface definitions in the repository.

Theorem 2. *A (best-effort) installation plan $\mathcal{R}'$ can be computed in polynomial time for any package $P \in \mathcal{R}$.*

Proof. We can compute a plan by taking the transitive closure of our package dependence relation, starting with only the desired package. Finding a package is a linear scan over $\mathcal{R}$ at worst, and interface subtyping consists of a membership check over a simple relation between interface names. Since cycles are not permitted, this process halts in polynomial time. □

Our installation theorem and its PTIME complexity trivially follow from the removal of version conflicts, a fact well known for version-based systems [7]. However, as we showed in §2, version-based package managers generally do not know when it is safe to install multiple versions, because their metadata cannot distinguish shared from non-shared dependencies. Hence our contribution is not stating a well-known complexity, but rather unlocking it with more expressive metadata.

Heuristics and Installation We can further minimize the size of the plan through a simple heuristic: we can reuse packages from the (partial) solution $\mathcal{R}'$ when resolving a dependence. With this heuristic we can in fact compute a plan for any $P \in \mathcal{R}$ by taking the initial solution to be the repository $\mathcal{S}$ that represents the user system. The result is then a new system $\mathcal{S}'$ that contains P.

So far our rules have relied on a well-formedness condition, which relies on type-checking packages (and hence on the complexity and soundness of our rules). We state soundness in §6.7, but do not provide a proof that type-checking is decidable in polynomial time. But since we have made no assumption on the core language besides the use of abstract data types and manifest types (both employed by OCaml), we think this is a reasonable assumption.

5 Dynamic Semantics

In this section we develop our dynamic semantics, for package linking and program evaluation. We use two forms of reduction— the first are the *linking steps*, written $\xrightarrow{\text{L}}$, which transform package definitions into *package values* without any remaining imports. Second are the *evaluation steps*, written $\xrightarrow{\text{E}}$, which evaluate projections on package values and core-language expressions.

5.1 Linking rules

In *Π* linking can be done statically (at compile time) or dynamically (at runtime). There is a further choice between resolving packages eagerly or lazily at runtime, which is analogous to the difference between call-by-value and call-by-need (conceptually call-by-name is also possible, but would break for nondeterministic package loaders or when side-effects or mutable state are permitted at the package level). For simplicity we assume call-by-value.

Below we give a small-step linking semantics. As mentioned in §3.3, during linking packages reduce to package expressions, which in turn reduce to package values until no more imports are left. Intermediate package expressions may contain further imports, which are evaluated before proceeding to evaluate the remainder of the package expression that contained the import. This means that packages are recursively linked in a depth-first order.

We represent linking with the relation $\xrightarrow{\text{L}}$, and we start evaluating the main program package to a package expression:

$$\frac{X \textit{ fresh}}{\mathcal{R} \vdash \textbf{pkg } \mathcal{P} \textbf{ impl } \mathcal{I} \ \{body\} \xrightarrow{\text{L}} \{body\}^X_{\mathcal{I}}} \quad \text{(Link-Prog)}$$

Next we need to resolve imports within package expressions. For simplicity of presentation we assume that package abstractions (and hence imports) have only a single argument. Imports proceed by looking up a suitable version of the dependency, and then applying the package to its argument by substituting the argument package value for the formal parameter of the package abstraction. The result of package application is a package expression, perhaps containing further import statements. The rule for imports is shown below (Link-Import):

$$\frac{\textbf{pkg } \mathcal{P}_1(Z : I_Z) \textbf{ impl } \mathcal{I}'_1 \ \{body_2\} \in \mathcal{R} \qquad \mathcal{I}'_1 <:_{nom} \mathcal{I}_1}{\mathcal{R} \vdash \{\textbf{import } \mathcal{P}_1 : \mathcal{I}_1 \textbf{ as } Y(P);\ body_1\}^X_{\mathcal{I}} \xrightarrow{\text{L}} \{[Z \mapsto P]\{body_2\}^Y_{\mathcal{I}_1}; body_1\}^X_{\mathcal{I}}}$$

Since imports reduce to package expressions, we now need to reduce package expressions to package values before we can evaluate the remaining body:

$$\frac{\mathcal{R} \vdash \textit{pexpr} \xrightarrow{\text{L}} \textit{pexpr}'}{\mathcal{R} \vdash \{\textit{pexpr}; body\}^X_{\mathcal{I}} \xrightarrow{\text{L}} \{\textit{pexpr}'; body\}^X_{\mathcal{I}}} \text{ (Link-Pkg-Expr)}$$

$$\frac{}{\mathcal{R} \vdash \{\{\overline{def}\}^Y_{\mathcal{I}_Y};\quad body\}^X_{\mathcal{I}} \xrightarrow{\text{L}} \{[Y \mapsto \{\overline{def}\}^Y_{\mathcal{I}_Y}]body\}^X_{\mathcal{I}}} \text{ (Link-Pkg-Value)}$$

Rule (Link-Pkg-Expr) reduces package expressions to packages values, while (Link-Pkg-Value) substitutes the package expression when it has been fully reduced to a package value (i.e. when it has no more imports).

Linking thus has the effect of "inlining" all dependencies, which proceeds recursively by first looking up the package, substituting the argument package value for its formal parameter, and evaluating any remaining imports in the package before substituting it for the package variable name from the import. This type of linking can be done statically (for example after type-checking), or dynamically (i.e. before program evaluation).

5.2 Evaluation rules

In our evaluation rules we assume that linking has already been performed. The linking rules reduce package definitions to package values, without any remaining imports or package applications. All that is left are then projections and (extended) core-language expressions. Evaluation starts with an expression of the form $(\{body\}^X_{\mathcal{I}}).main\ ()$, and the first thing we evaluate is the projection of the *main* function:

$$\frac{}{\begin{array}{l}(\{body_1;\ \mathbf{fun}\ f : \tau = e; body_2;\ \}^X_{\mathcal{I}}).f \\ \quad \xrightarrow{\mathrm{E}} \quad [\xi \mapsto (\{\ldots\}^X_{\mathcal{I}}).\xi \mid \xi \in dom(body_1)]\ e\end{array}} \text{(E-Proj-Fun)}$$

When we project a function we need to handle any references to abstract data types and functions defined in the same package as f — otherwise we may end up with an ill-formed term. For example we may have $e = \lambda x : T.g\ x$, in which case we want transform T and g into projections on the package value, i.e. $\lambda(x : \{\ldots\}^X_{\mathcal{I}}.T).\{\ldots\}^X_{\mathcal{I}}.g\ x$.

To do this we use ξ to range over both functions f and type names T, and we write $dom(body)$ for the set of names ξ bound within *body*. This way we can rewrite any free variables in the function body e of f to projections on the package value.

Now our entire program consists of a closed expression, which may contain applications, further projections and wrappings and unwrappings. The (standard) rules for function application are shown below.

$$\frac{e_1 \xrightarrow{\mathrm{E}} e_1'}{e_1\ e_2 \xrightarrow{\mathrm{E}} e_1'\ e_2} \text{(E-App-Left)}$$

$$\frac{e_2 \xrightarrow{\mathrm{E}} e_2'}{(\lambda x.e_1)\ e_2 \xrightarrow{\mathrm{E}} (\lambda x.e_1)\ e_2'} \text{(E-App-Right)}$$

$$\frac{}{(\lambda x.e)\ v \xrightarrow{\mathrm{E}} [x \mapsto v]e} \text{(E-App)}$$

Finally, we must be able to handle wrappings and unwrappings of abstract data type terms:

$$\frac{e \xrightarrow{\mathrm{E}} e'}{wrap_T(e) \xrightarrow{\mathrm{E}} wrap_T(e')} \text{(E-Wrap)}$$

$$\frac{e \xrightarrow{\mathrm{E}} e'}{unwrap_T(e) \xrightarrow{\mathrm{E}} unwrap_T(e')} \text{(E-Unwrap)}$$

$$\frac{}{unwrap_T(wrap_T(e)) \xrightarrow{\mathrm{E}} e} \qquad \text{(E-UnwrapWrap)}$$

The fact that values of abstract data types can only be wrapped (introduced) and unwrapped (eliminated) inside the package they are defined is our argument that evaluation preserves data abstraction, as there are no evaluation rules that work on wrapped terms other than the (E-UnwrapWrap).

6 Static Semantics

Now that we have covered linking and evaluation, we present our static semantics, inspired by manifest types [22].

6.1 Typing

In our typing rules we use a context Γ that keeps track of package and core-language types, as well of type definitions:

$$\Gamma ::= \overline{X \mapsto \{\overline{decl}\}^X_{\mathcal{I}}} \quad \overline{x \mapsto \tau} \quad \overline{T = \tau}$$

We have previously used package expressions in our linking rules. During type-checking, we use corresponding package types of the form $\{\overline{decl}\}^X_{\mathcal{I}}$, which are constructed from interface applications. While package application substitutes package values for package variables, interface application has the effect of substituting access paths (i.e. package variable names). For clarity of presentation we use a separate interface application rule:

$$\frac{\textbf{iface}\ \mathcal{I}(Z : I_Z)\ \{\overline{decl}\} \in \mathcal{R} \qquad \mathcal{R}; \Gamma \vdash Y : I_Y \qquad \mathcal{R}; \Gamma \vdash I_Y <: I_Z}{\mathcal{R}; \Gamma \vdash \mathcal{I}(Y)^X \rightsquigarrow \{[Z \mapsto Y]\overline{decl}\}^X_{\mathcal{I}}} \qquad \text{(T-IfaceApp)}$$

That is, we substitute access path Y (the path of the argument) for the access path Z (the path of the parameter). Besides looking up the interface name in the repository and performing substitution, the rule also checks that the argument package has the type required for the interface parameter ($I_Y <: I_Z$). Resolving interface application is a recursive process, which terminates because of restrictions on circularity imposed on repositories (both for packages and interfaces).

Interface application is perhaps the most interesting part of our typing rules, and is used in particular to reduce interface applications used to define package parameters, package imports and interface parameters. We start with package definitions (which for simplicity of presentation take only a single argument):

$$\frac{X\ \mathit{fresh} \qquad \mathcal{R}; [X : I] \vdash \mathit{body} : B}{\mathcal{R}; \bullet \vdash \textbf{pkg}\ \mathcal{P}(X : I)\ \textbf{impl}\ \mathcal{I}\ \{\mathit{body}\} : \{B\}^X_{\mathcal{I}}} \qquad \text{(T-Pdef)}$$

We require that the body implemented by the package is a (structural) subtype of the body of the implemented interface, which is checked by the well-formedness rule for packages (see §6.4). Packages are used by importing them, in which case context Γ is augmented with the package types corresponding to the import:

$$\frac{\mathcal{P} : \mathcal{I} \in \mathcal{R} \qquad \mathcal{R}; \Gamma \vdash \mathcal{I}(Y)^X \rightsquigarrow I_X \qquad \mathcal{R}; \Gamma[X \mapsto I_X] \vdash \mathit{body} : B}{\mathcal{R}; \Gamma \vdash (\textbf{import } \mathcal{P} : \mathcal{I} \textbf{ as } X(Y); \mathit{body}) : B} \text{(T-PIMPORT)}$$

First we check that there exists a suitable package definition for the import (which may implement a subtype $\mathcal{I}'$ of $\mathcal{I}$). We then add the package definition to Γ and type-check the package body. The resulting package types are useful in particular for resolving projections of functions from package variables:

$$\frac{\Gamma \vdash X : \{\textbf{type } T \ \ldots \ \textbf{fun } f : \tau \ \ldots\}^X_{\mathcal{I}}}{\Gamma \vdash X.f : [T \mapsto X.T]\tau} \text{(T-PROJ)}$$

Although we have already rewritten the paths of interface parameters with the paths of their arguments, we also need to handle references to abstract data types within the interface itself. We do this by prefixing such references with the package variable name under which the package was imported. For example, in a projection $X.f$ where $f : \tau \to T$, we have $X.f : \tau \to X.T$.

Next we need to assign types to definitions, which are simply transformed to corresponding declarations. When typing a type definition we check that T is not already defined in the package body, and then add it to core-language context Γ:

$$\frac{T \notin \mathrm{dom}(\Gamma) \qquad \mathcal{R}; \Gamma[T = \tau] \vdash \mathit{body} : B}{\mathcal{R}; \Gamma \vdash (\textbf{type } T = \tau; \mathit{body}) : (\textbf{type } T = \tau; B)} \text{(T-TYPE-DEF)}$$

Function definitions are similar: we check that f is not already defined in the package body, and then add it to core-language context Γ:

$$\frac{f \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash e : \tau \qquad \mathcal{R}; \Gamma[f : \tau] \vdash \mathit{body} : B}{\mathcal{R}; \Gamma \vdash (\textbf{fun } f : \tau = e; \mathit{body}) : (\textbf{fun } f : \tau = e; B)} \text{(T-FUN-DEF)}$$

Rules for type and function declarations are nearly identical, except for the omission of the type definition τ and the term definition e.

Core-language rules are the standard typing rules for the simply-typed lambda calculus, and are omitted. We also have to type wrap and unwrap operations:

$$\frac{T = \tau \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \mathit{wrap}_T(e) : T} \text{(T-WRAP)}$$

$$\frac{T = \tau \in \Gamma \qquad \Gamma \vdash e : T}{\Gamma \vdash \mathit{unwrap}_T(e) : \tau} \qquad \text{(T-Unwrap)}$$

The side condition $T = \tau \in \Gamma$ ensures that wrap and unwrap operations are only possible within the context of the packages that defines T. This is our argument that our semantics properly enforces data abstraction.

Typing the Extended Grammar The above rules suffice to type-check programs and packages. We also need additional rules to type-check the extended syntax, in order to argue that linking reduction steps preserve types. We show this informally without proving soundness of the linking operations.

To start, we need to assign types to package values. Fortunately, this is straightforward, as we can first check the package body and use this to assign the package type:

$$\frac{\mathcal{R}, \Gamma \vdash \mathit{body} : B}{\mathcal{R}, \Gamma \vdash \{\mathit{body}\}^X_{\mathcal{I}} : \{B\}^X_{\mathcal{I}}} \qquad \text{(T-PkgVal)}$$

Since linking has substituted package values for term-level package variables, we need to update our typing rule for imports and for projections. The rules are essentially the same as their variable-based counterparts, and involve extracting the access path:

$$\frac{\mathcal{R}; \Gamma \vdash \{\mathit{body}\}^X_{\mathcal{I}} : \{\textbf{type } T \ \ldots \ \textbf{fun } f : \tau \ \ldots\}^X_{\mathcal{I}}}{\mathcal{R}; \Gamma \vdash (\{\mathit{body}\}^X_{\mathcal{I}}).f : [T \mapsto X.T]\tau} \text{ (T-ProjPkgVal)}$$

The rule above is a straightforward adjustment of (T-Proj), employing package values $\{\mathit{body}\}^X_{\mathcal{I}}$ instead of package variables X. The rule for imports is a similar adjustment of the (T-Import), and omitted.

6.2 Type Equivalence

Crucial to our system is preserving data abstraction, for which we need to reason about equality of abstract data types using a generative type semantics. To do this, we need to distinguish abstract data types with the same name, that originate from different packages or package instantiations.

Following [22] we use *manifest types* and base type equality, written $\approx$, on *paths*. Paths are manipulated by interface application, by updating the access path for type projections. Two abstract type names are equivalent if and only if they have the same path:

$$\frac{X = Y}{X.T \approx Y.T} \qquad \text{(Eq-Abs-Type)}$$

In our extended syntax we have also substituted package values for package variables, so we need an addition type equality rule for type projections:

$$\frac{}{X.T \approx (\{\overline{def}\}^X_{\mathcal{I}}).T} \qquad \text{(Eq-Abs-Proj)}$$

This finally reveals why tracking package variables as part of package values was useful.

6.3 Subtyping

So far we have used two forms of subtyping rules, namely a nominal relation $<:_{nom} \subseteq \mathcal{I} \times \mathcal{I}$, and a more traditional relation $<: \ \subseteq \tau \times \tau$. Nominal subtyping is used to look up package definitions in a repository, either to link a dependency into a program or during dependency resolution to compute installation plans (see §4.1). The more traditional subtyping relation is used to check that packages properly implement their interfaces (T-Pdef), for type-checking package applications (R-TyApp) and for checking interface well-formedness (§6.4).

Nominal Subtyping is defined explicitly in interface definitions, and the nominal subtyping relation is reflexive transitive closure of this explicit declaration:

$$\frac{\textbf{iface}\ \mathcal{I}(X : I_X) <: \mathcal{I}'\ \{\overline{decl}\} \in \mathcal{R}}{\mathcal{R} \vdash \mathcal{I} <:_{nom} \mathcal{I}'} \qquad \text{(NomSub-IFace)}$$

Traditional Subtyping is defined as the reflexive transitive closure of the rules for package types, package type bodies and declarations below. Subtyping for package types relies only on the nominal subtyping relation. This works because interface and repository well-formedness have already ensured us that the interface body is also a subtype (according to this relation).

$$\frac{\mathcal{R} \vdash \mathcal{I} <:_{nom} \mathcal{I}'}{\mathcal{R}; \Gamma \vdash \{B_1\}_{\mathcal{I}} <: \{B_2\}_{\mathcal{I}'}} \qquad \text{(Sub-IFace)}$$

In addition to simply deferring to the nominal subtype relation, we need to handle subtyping of package type bodies and declarations, in order to define well-formedness. Subtyping for bodies includes width, depth and permutation subtyping:

$$\frac{\mathcal{R}; \Gamma \vdash B_{\sigma(i)} <: B'_i \ \ (1 \le i \le m) \qquad \sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \qquad m \le n}{\mathcal{R}; \Gamma \vdash \{B_1; \ldots; B_n\} <: \{B'_1; \ldots; B'_m\}} \qquad \text{(Sub-Body)}$$

We define subtyping of declarations in terms of syntactic equality. However, in a serious implementation we may wish to enrich the definition, for example if subtyping is defined on core language types, or to allow scenarios such as the following:

$$\mathcal{R}; \Gamma[T = \tau] \vdash \textbf{fun}\ f : Int \to \tau <: \textbf{fun}\ f : Int \to T$$

6.4 Well-Formedness

Well-formedness is used by repositories to check for correctness of package and interface definitions. We say packages are well-formed if they are well-typed (first line) and a proper subtype of their interface (second line):

$$\frac{\begin{array}{c}\mathcal{R};\bullet \vdash \textbf{pkg}\ \mathcal{P}(X:I)\ \textbf{impl}\ \mathcal{I}\ \{body\} : \{B\}^{Y}_{\mathcal{I}} \qquad \text{for some } Y \\ \mathcal{R};\bullet \vdash \mathcal{I}(X) \rightsquigarrow \{B'\}_{\mathcal{I}} \qquad \mathcal{R};[X:I] \vdash B <: B'\end{array}}{\mathcal{R};\bullet \vdash \textbf{pkg}\ \mathcal{P}(X:I)\ \textbf{impl}\ \mathcal{I}\ \{body\}\ \texttt{WF}} \text{ (WF-PDEF)}$$

We say interfaces are well-typed if they are well-scoped and a proper subtype of their supertype interface. We omit scoping rules for interfaces, as they provide no useful insights, but in brief it checks that any referenced abstract data types and any projections are valid. To check that an interface is a proper subtype of its declared supertype, we check that it is contra-variant in its parameter types (line 1) and covariant in its return type/body (line 2):

$$\frac{\begin{array}{c}\mathcal{R};\bullet \vdash \textbf{iface}\ \mathcal{I}'(Y:I_Y)\ \{B'\} \in \mathcal{R} \qquad \mathcal{R};\bullet \vdash I_Y <: I_X \\ \mathcal{R};[X \mapsto I_Y] \vdash \{B\} <: \{[Y \mapsto X]B'\}\end{array}}{\mathcal{R};\bullet \vdash (\textbf{iface}\ \mathcal{I}(X:I_X) <: \mathcal{I}'\ \{B\})\ \texttt{WF}} \text{ (WF-IFACE)}$$

6.5 Separate Type-Checking and Compilation

Our system supports separate type-checking, as the type-checker looks up interfaces in repository $\mathcal{R}$, but not package implementations $\mathcal{P}$. We would argue that our system could also support separate compilation of packages. One way of doing this would be to treat package values as records of function pointers (type definitions were needed in our operational semantics only to show type preservation, but not not needed in a real implementation). Then imports could be compiled into code that finds a suitable package in $\mathcal{R}$, and evaluates it recursively (e.g. through a package initialisation routine). This initialisation routine could further take package values (i.e. records of function pointers) as arguments, which initialises local package variable pointers with the supplied arguments. Projection of functions is then compiled as accesses to members of the package record values.

6.6 Implementation

We have implemented our package system in Haskell, which can be accessed from [3]. Repositories are specified within a single file as a sequence of interface and package definitions. Each new definition "extends" the repository and is type-checked with respect to previous definitions. The repository may contain a program (a package taking no arguments and defining a *main* function), which can be evaluated. During evaluation imports are resolved dynamically by finding a suitable definition in $\mathcal{R}$. Our implementation comes with various example repositories, accessible here: [2] [5].

6.7 Soundness

We have covered linking ($\xrightarrow{\text{L}}$), evaluation ($\xrightarrow{\text{E}}$) and typing. We are now in a position to state progress and preservation theorems. We use the relation $\longrightarrow$ to stand for either linking or core-language evaluation. That is, whenever we write $\mathcal{R} \vdash e \longrightarrow e'$ we mean either $\mathcal{R} \vdash e \xrightarrow{\text{L}} e'$ or $e \xrightarrow{\text{E}} e'$ (core-language evaluation does not need a repository $\mathcal{R}$). The purpose is to show that both linking and evaluation preserve types, and that neither gets stuck on well-typed terms. We use e to range over all terms (package or core-language), and τ to range over all types (interface or core-language):

Theorem 3 (Type Preservation)**.** *If $\mathcal{R}; \Gamma \vdash e : \tau$ and $\mathcal{R} \vdash e \longrightarrow e'$, then $\mathcal{R}; \Gamma \vdash e' : \tau$.*

Theorem 4 (Progress)**.** *If $\mathcal{R}; \bullet \vdash e : \tau$ (e is closed and well-typed), then either e is a value or there is some e' such that $\mathcal{R} \vdash e \longrightarrow e'$.*

Values include package values and core-language values (lambdas, primitive integers, etc).

7 Design and Change Management

In the design of a package system there are a number of considerations that are different from module systems, which arise because of versioning. Evolution of packages must be managed in order to remain *backwards compatible* as much as possible with importing packages. This is useful because breaking changes (changes that are backwards incompatible) force others to adapt to those changes in newer versions of their code, and may further prevent sharing of dependencies due to conflicting interface requirements (e.g. `P` and `Q` from Figure 1 may have conflicting requirements of `Wombat`).

Flexibility in package compatibility is desired, but only in a semantically meaningful way, as opposed to a structurally meaningful way, which would allow for "spurious subsumption" [24]. First, we argue that interfaces should be *named* and *reusable* to this end. Second we can exploit these meaningful names to reason formally (and explicitly) about backwards compatibility, so that more caution can be taken when changing APIs. Third, we argue that semantically named and reusable interfaces are incompatible with *structural typing*, because one cannot resolve a dependency with a name only to then treat the package as its structural contents. Finally, we argue that interfaces capture *concerns* in packages that deal with specific aspects on the public API, which are best separated according to the "separation of concerns" principle [12].

7.1 Change Management

Change in the interface of a package can be either *semantic* or *structural*. A semantic change is a change in the higher-level semantics of an interface, while

a structural change is a change to the structure (or *signature*) of the interface (e.g. removing a type, changing a function signature, etc). While incompatible changes to structure are detected automatically by a type-checker, the higher-level semantics are meaningful to humans but not the type-checker. To distinguish between an (incompatible) change of semantics we use interface names $\mathcal{I}$.

While removing or changing components in interfaces are breaking changes, adding components is not. To support compatibility with packages that have more components exposed in their public interface, we add subtyping to our system.[4] Because interface names carry meaning, our subtyping relation is also *nominal* (i.e. explicitly defined between interface names). This allows us to reason about backwards compatibility of interfaces by relying on subtyping:

Definition 2. *An interface* `I-vN` *is* backwards compatible *with an interface* `I-vM` *if* $N \geq M$ *and* `I-vN` $<:$ `I-vM`*.*

Similarly, we can say a package is backwards compatible with respect to a previous version whenever its implemented interface is backwards compatible with the interface of the previous version. Subtyping is defined in such a way that packages/interfaces are contravariant in their parameters and covariant in the package/interface body. This corresponds to subtyping for functors in Standard ML [18] and standard subtyping for function types [24]. Imported packages can be ignored, because they are not part of the exposed interface of a package.

Structural Typing and Names Standard ML and OCaml (among others) use structural typing for *signatures* (the structural equivalent of our interfaces). We have argued for the use of interface names to import packages, which is incompatible with structural typing because we can import a package with a name, say `I-v2`, but subsequently use the package as a `I-v1` where `I-v2` $<:$ `I-v1`. However, the package may implement `I-v2` but not `I-v1`, and the relation between `I-v2` and `I-v1` is not explicitly defined. Hence a system based on structural subtyping that wishes to have separately defined and named interfaces must treat them purely in a structural fashion both for type-checking and dependency resolution.

Multiple Interfaces Packages are more heavyweight than modules (they are often comprised of many modules), and typically handle multiple *concerns*. For example, we could have a package **pkg** `Wombat-v1` **impl** `IFeedAndGroom-v1` that handles both feeding and grooming of wombats. Any incompatible change to wombat grooming results in a **pkg** `Wombat-v2` **impl** `IFeedAndGroom-v2`, breaking backwards compatibility even with packages looking to feed their wombats. Concerns should be separated by implementing *multiple interfaces*, isolating change in different interfaces. Although we could argue that packages should

[4] Note that superficially this can be modelled by implementing multiple interfaces. However, in dependency structures such as Figure 1 the bottom of the diamond `D` would have to import `Wombat` twice. This then raises the question of whether the type `TWombat` is shared between those two imports.

address a single concern, this may be incompatible with their course-grained nature (leading users to ignore the separation principle and write monolithic interfaces anyway).

8 Expressiveness

Since our system is meant to address problems in decentralized package systems based on version ranges (for example cabal, opam or pypi), we compare the expressiveness our Π with that of traditional systems. We do this by (informally) exploring the expressiveness through possible *encodings* of our system into version-based systems, and vice-versa. Although encodings are always possible through (whole-program) restructurings, as discussed below such restructurings may not be computationally tractable.

8.1 Encoding Π into Version-Based Systems

Dependency Hell Encoding Π into a system based on versions may be non-trivial if we want to avoid dependency hell, as version-based systems would falsely assume sharing of our internal dependencies. This assumption is based on package names, and avoiding this may require renaming internal dependencies and duplicating them in the repository. However, duplication has to proceed recursively, as a duplicated/copied package will have the same imports as the original, and may otherwise generate a conflict on one of *its* internal dependencies. However, such duplication grows exponentially in the depth of the dependency graph, and may generate unnecessarily large package repositories.

Further, our abstraction mechanism allows for packages to be linked against different packages or package versions in different contexts, a construct not normally supported by many programming languages. Emulating such constructs in a version-based system may require further duplications and renamings of package definitions.

Interfaces and Semantic Versioning Packages in Π can express dependencies on future versions of a dependency if that dependency implements (a subtype of) the required interface. In some decentralized package system this is modelled by *semantic versioning*, which places compatibility assumptions on future package versions encoded within the versioning name *MAJOR.MINOR.PATCH* [4]:

- *MAJOR* is bumped whenever the API is changed in a backwards-*incompatible* way
- *MINOR* is bumped when the API is changed in a backwards-*compatible*
- *PATCH* is bumped for changes that don't affect the API (e.g. bug fixes)

However, this cannot (directly) deal with backwards compatibility with respect to different subsets of a public API (i.e. packages implementing multiple interfaces).

8.2 Encoding Versions into Π

Decentralized package system are often developed for a particular programming language, to manage the packages written in that language. Package systems such as Debian on the other hand have a community that centrally audits packages and maintains package metadata for entire operating systems. Such systems have to support a variety of dependency types, such as dependency on a package written in different language, dependency on configuration data, command-line programs or dependency on a service. So far we have only looked at *API-dependencies* of program libraries written in the same language. [5]

For our system to be applicable to such type of package systems we believe a long list of extensions are in order. This list is necessarily long because we provide strong guarantees: installation always works and package linking does not give errors, and preserves type-safety and data abstraction. Further, the system can efficiently check that new package submissions to the repository do not violate any of these conditions. Verifying such properties would require ways of expressing types in the first place. For example, to express dependence on a communicating service may require the use of session types [11] or a similar typing discipline.

Although this is an important direction for future work, we believe that a system of this variety will be hard or impractical to apply to existing ecosystems such as Debian. Instead, we think our system is particularly applicable to decentralized package systems for new, clean-slate, programming languages, or perhaps for existing statically-typed programming languages for API dependencies. Below we compare Π to similar package systems based on versions.

Version Restrictions Traditional package systems based on version ranges can restrict package versions, which is especially useful to rule out buggy or vulnerable implementations. We can incorporate such functionality into Π by adding version restrictions on `imports` in addition to the interface restriction. This then restricts import resolution without sacrificing any guarantees or simplicity of dependency resolution. This is because we are free to choose any compatible version of the package without restrictions from outside (i.e. the constraints are local). Adding version restrictions to external dependencies (formal parameters) would however require non-trivial extensions to our system, with a question around whether the version restrictions on packages should be part of the interface annotations.

Conflicts Traditional package systems can often express *conflicts* between packages [10], e.g. because they both provide a service that binds to the same port. We can extend our system to deal with such cases (e.g. **pkg** `FastHTTP` **impl** `IHTTP-v1` **conflicts** `HTTP`), with corresponding changes to the dependency resolution algorithm. Conflicts might be enforced (efficiently) in a similar way as our

[5] Many languages already have Foreign Function Interfaces (FFIs) for calling into other languages. Perhaps foreign functions declarations could be defined as as part of "foreign interfaces", assuming appropriate compiler extensions.

restrictions on package circularity, covered in §4.1. Such an extension of course does mean that we can no longer guarantee to install any package from a repository. However, we may ask ourselves if such conflicts are not better expressed at a "whole systems"-level, rather than at the level of individual packages.

Platforms Some packages may be *platform-specific*, which we could support through an import "disjunction": **import** `P : IP-v1` **from** `PLinux` **or** `PWindows`. Disjunctions that depend on user preferences or configuration data may prove useful in general.

Circularity Our system does not support *circular dependencies* or recursive linking. However, our system can in principle be extended to include *fixpoints* of package applications, similar to the *functor fixpoints* based on the ML module system detailed in [19]. For example, we could write:

import Foo−*any* : IFoo−v1 as F(B)
import Bar−*any* : IBar−v1 as B(F)

Recursive modules, and the interplay with mutable variables and circular definitions have been extensively studied in [14][13][25][19][16] and others.

9 Discussion and Related Work

Module Systems As alluded to earlier, our package system is much like module systems such as that of Standard ML [18] or OCaml, MixML [15], Units [17] or component-based systems [21]. This is because the purpose of module systems is to support modular development and a way of linking modular code units into larger ones. All these systems have an abstraction mechanism in common that uses types or interfaces to express how modules may be composed. Our presentation differs in that we focus on packages instead of modules, which have different dependency relations than modules (hence our **import** operator). For example, both module systems and package systems need a (typed) abstraction mechanism, but their motivation is rather different: module abstraction is used for *multiple instantiation* (as well as separate compilation), while package abstraction is used primarily to reason about package equality for common dependencies, necessary because of versioning.

We further support separate compilation of packages, which is not possible in module systems which use bare names M to identify modules. We argue that these differences are substantial enough to re-evaluate common design decisions, in particular with regard to how packages and interface are identified, how subtyping is done, and when sealing occurs.

Although component systems often feature repositories of components and sometimes automatic matching (similar to what our import operator does), we are not aware of any component system that covers component versioning and dependency hell.

Backpack is a package system for Haskell [20], based on the module calculus MixML. The main difference in our presentation is that we handle versioning, a

defining aspect that sets packages apart from modules. With versioning in place, we can reason about dependency hell and the constructs required to absolve it. This leads us to think about change management and backwards compatibility, where we place increased importance on names and the use of nominal instead of structural typing.

Backpack is applied to an existing language, and is therefore more practically motivated. Backpack, building on MixML, further supports recursive and more flexible linking with an applicative semantics. Applicativity is not of great concern in the design of our system for two reasons: first, packages cannot export internal dependencies, which means that applicativity of imports would only work locally within the package body. Second, our package abstractions are first-order, so we cannot export types from an abstraction passed in as an argument.

Functional Package Managers such as Nix [9] manage packages *functionally* by never removing or mutating them explicitly during upgrade or installation. Instead packages are only ever added to the system, allowing seamless rollback to earlier versions of the system. Packages are then *garbage collected* to free diskspace. In Nix the user describes a complete system, making systems reproducible. Our system differs in that we compose packages modularly, without requiring a description of a complete system. Systems can then be automatically generated by a dependency resolution procedure. Further, we use interfaces for modular compilation, while Nix can only verify a package composition once all dependencies have been satisfied (requiring intricate knowledge of package compatibility). We also support type-safety at the package level, and support runtime linking. The concept of functional package management is a powerful one, which implementations of our system almost invariably would adopt. Reproducibility in our system would be supported by generating a list of package versions for a particular user system $\mathcal{S}$.

Virtual Package Environments such as virtualenv [8] or Cabal sandboxes provide a fresh environment for packages to be installed (often including core components such as a compiler or interpreter). Although this can help reduce version conflicts, the same problem still exists within the individual environments. Tools such as *Vagrant* [6] or *Docker* [1] solve woes around code deployment, by *isolating* the entire operating system (or large parts of it) from the host system by *sandboxing* a virtual image. Although powerful for code deployment, creating an image still relies on traditional package systems and type-safety is not addressed.

Traditional Package Systems We have already covered traditional package systems based on version ranges. Often such package managers employ advanced SAT-solvers to compute which package versions are required. By now we hope that the advantage of Π over version-based systems have become clearer. These systems have seen great advances over time, to a point where packaging problems may seem like a problem on systems such as Debian. However, we would like to point out once more that this is to a large extent because of the involvement of

communities that work hard to make sure that upgrades for user systems work properly. In decentralized systems this model often breaks: Cabal frequently fails with compile errors due to Haskell's strong type system, and in Python or Javascript runtime exceptions (when lucky) or "mysterious runtime behaviour" (when unlucky) are not uncommon. For example, although the `npm` package manager for Javascript installs packages side-by-side, it has no notion of when it is safe to do so. Hence mutable state that must be shared may be erroneously duplicated, and data abstraction may be violated.

We note however that our internal dependencies aren't suitable for all scenarios. For example, for security reasons we may wish to only link in a single TLS library into our programs — instead of an old version with known security vulnerabilities. Although an interesting direction for future work, enforcing such properties system-wide (perhaps influenced by user-defined policies), is beyond the scope of this work.

10 Conclusions

We have introduced a package system based on interfaces with strong safety guarantees: any package contained in a shared repository of packages can be installed onto a user system, and the resulting libraries and programs defined by those packages are type-safe. A defining aspect has been change management, so that packages can be flexibly updated with minimal effects to the package ecosystem (and the need for others to adapt to change in other packages).

We hope to have shown that there are real problems in package management, which are not a mere result of buggy or incapable package managers. Dependency hell is a direct result arising from the limitations of linkers and inexpressiveness of package metadata. Package systems for Linux and other systems work as well as they do because of extensive community involvement, which includes testing and auditing of a centralized package repository. With our approach packages can be developed in a distributed way, and composed in a type-safe way.

An important direction for future work is to apply package systems such as ours to existing languages, and study the interplay with different language features such as subtyping, dynamic typing, unsafe language features, and so on. To be practically applicable to new programming languages, it may make sense to incorporate more features from module systems (such as more sophisticated type sharing constraints), to create a unified system for both packages and modules. Other directions of future work may include increasing the expressiveness of the language to include other forms of dependency (communicating programs, cross-language support, command-line tools, and so on).

Acknowledgements Omitted for review version.

References

1. Build, ship, and run any app, anywhere, https://www.docker.com/

2. Evaluation examples of the package system, https://github.com/markflorisson/packages/blob/master/Test/Eval/EvalDiamond.pkg
3. Implementation of the package system, https://github.com/markflorisson/packages
4. Semantic versioning, http://semver.org/
5. Type-checking examples of the package system, https://github.com/markflorisson/packages/tree/master/Test/TypeCheck
6. Vagrant, https://www.vagrantup.com/
7. Abate, P., Cosmo, R.D., Treinen, R., Zacchiroli, S.: Dependency solving: A separate concern in component evolution management. Journal of Systems and Software 85(10), 2228–2240 (2012)
8. Bicking, I.: Virtualenv, https://virtualenv.pypa.io/en/latest/
9. van der Burg, S.: A Reference Architecture for Distributed Software Deployment. Ph.D. thesis (2013)
10. di Cosmo, R.: Report on formal management of software dependencies (05 2012)
11. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and session types: An overview. Lecture Notes in Computer Science pp. 1–28 (2010)
12. Dijkstra, E.W.: On the role of scientific thought. Selected Writings on Computing: A personal Perspective pp. 60–66 (1982)
13. Dreyer, D.: A type system for well-founded recursion. ACM SIGPLAN Notices 39(1), 293–305 (2004)
14. Dreyer, D.: Understanding and evolving the ML module system (05 2005)
15. Dreyer, D., Rossberg, A.: Mixin' up the ML module system (09 2008), http://dx.doi.org/10.1145/1411203.1411248
16. Duggan, D.: Type-safe linking with recursive DLLs and shared libraries. ACM Transactions on Programming Languages and Systems 24(6), 711–804 (2002)
17. Flatt, M., Felleisen, M.: Units: Cool modules for hot languages. ACM SIGPLAN Notices 33(5), 236–248 (1998)
18. Harper, R.: Programming in Standard ML (2011)
19. Im, H., Nakata, K., Garrigue, J., Park, S.: A syntactic type system for recursive modules. Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications – OOPSLA '11 (2011)
20. Kilpatrick, S., Dreyer, D., Jones, S.P., Marlow, S.: Backpack: Retrofitting haskell with interfaces. Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages – POPL '14 (2014)
21. Lau, K.K., Wang, Z.: Software component models. Proceeding of the 28th international conference on Software engineering – ICSE '06 (2006)
22. Leroy, X.: Manifest types, modules, and separate compilation. Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94 (1994)
23. Leroy, X.: Applicative functors and fully transparent higher-order modules. Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages – POPL '95 (1995)
24. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA (12 2002)
25. Russo, C.V.: Recursive structures for Standard ML. ACM SIGPLAN Notices 36(10) (2001)