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 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 packages written in a single programming language, although inter-language dependencies are an important direction for future work.

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

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

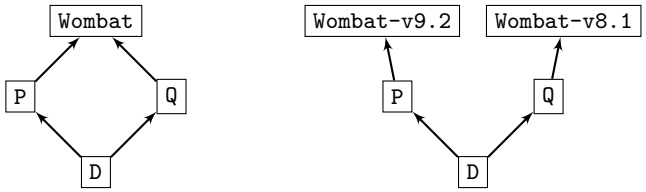

Fig. 1. Diamond dependency with a shared `Wombat` package (left) and with two separate `Wombat` packages (right).

repositories that are not carefully audited, because version ranges may be overly constrained to guard against possible future incompatibilities. Installation problems are unpredictable, as dependency resolution depends on 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.

Systems allow only a single version of a package for two reasons. First, package linkers can often not 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).

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 presumes 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,

[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).

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 $\prod$, 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 $\prod$ that:

1. does not suffer from dependency hell, portability or type-safety issues
2. can compile packages separately, and link packages dynamically or statically
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 [6] (§5.2).

There are great similarities between module systems such as that of ML [14] and package systems such as $\prod$. Indeed, recent work on Backpack [16] proposes a package system for Haskell based on *mixin modules* from the MixML module calculus [10]. However, we argue that package systems are different from modules and that we *abstraction* (such as functors in ML) is used to express *sharing* of dependencies, rather than *multiple instantiaton*. Our presentation differs from Backpack in that we explicitly address *dependency hell* and *change management*. Further differences are detailed in §7.

2 Package System and Examples

Our package system explicitly handles the two scenarios from Figure 1 by distinguishing between *external* dependencies and *internal* 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.

We express external dependencies using *abstraction* (formal parameters), to defer linking of the dependency. A dependant 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`. Package `D` is then 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. Internal dependencies are expressed with an **import** construct, which loads *some* compatible version of the dependency. No abstract data types are exposed from internal dependencies, but they can always be hidden behind a new abstract data type. The grammar and scoping rules of our system automatically ensure this, as interfaces have no way of referring to internal dependencies. In our system packages can be 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 of `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(W)
  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. Because `P` wants to expose `TWombat`, interface `IP` also takes a parameter of type `IW-v1`, allowing it to refer to `W.TWombat`. To show how the `W` parameter to package `P` is related to the `W` parameter of interface `IP`, `P` applies `IP` to `W`: **pkg** `P-v9`$(\ldots)$ **impl** `IP-v1`(W). This allows flexibility in how interfaces are used, to which we return in §4. Package `Q` is similar to `P` so we omit it here. Finally there is package `D`:

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

```
iface ID–v2
  ...
```

`D` first imports `Wombat`, and then *instantiates* (or *links*) `P` and `Q` with `Wombat`, applying both packages (`P` and `Q`) and interfaces (`IP-v1` and `IQ-v6`) to `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. Here 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. For simplicity our packages export only flat namespaces. 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 interfaces contain *declarations* and packages contain *definitions*.

3.1 Package Grammar

In this section we formalise the syntax of package constructs. As mentioned we version both packages (`Wombat-v9.2`) and interfaces (`IW-v2`). 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
PVname	`Wombat-v9.2`	a concrete package in a repository
Pname	`Wombat-any`	a request for some version of a package
Pvar	`P`	a package parameter or local name for an imported package
IVname	`IWombat-v3`	a concrete interface in a repository
Ipar	`I`	an interface parameter name
T	`TWombat`	a core-language type
Uvar	`x`, `f`	a core-language variable or function.

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 package names (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 *PVname* and *IVname* to define backwards compatibility later.

Names in category *Ipar* range over *interface parameters*, which are used to specify external dependencies at the interface level. Similarly, *Pvar* ranges over *package variables*, which represent either internal or external dependencies (i.e. parameters or local imports).

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.

Types and Terms As in the ML module system [19], the (*core-language*) type and term language is largely independent of the package system. However, we assume the language has abstract data types T and functions f over such types.[3] We support functions and corresponding arrow types $\tau_1 \to \tau_2$, and projection of type and term components from *Pvar* and *Ipar* using standard "dot" notation (e.g. `W.TWombat`).

$$
\begin{array}{rll}
e & ::= f \mid \mathit{Pvar}.f \mid \ldots & \textit{(terms)} \\
\tau & ::= T \mid \tau \to \tau \mid \mathit{Pvar}.T \mid \mathit{Ipar}.T \mid \mathit{Unit} \mid \mathit{Int} \mid \mathit{Bool} \mid \mathit{ldots} & \textit{(types)} \\
\mathit{Tdecl} & ::= \textbf{type}\ T & \textit{(abstract data type declaration)} \\
\mathit{Tdef} & ::= \textbf{type}\ T = \tau & \textit{(abstract data type definition)} \\
\mathit{FunDecl} & ::= \textbf{fun}\ f : \tau & \textit{(function declaration)} \\
\mathit{FunDef} & ::= \textbf{fun}\ f : \tau = e & \textit{(function definition)}
\end{array}
$$

Packages and Interfaces Interface and package definitions *Idef* and *Pdef* have largely parallel syntax; their bodies *Ibody* and *Pbody* consist of (type and function) declarations and definitions respectively. Both interfaces and packages may be parameterised—interfaces by named interface parameters *Ipar* and packages by named package parameters *Pvar*. Such parameters are constrained by interface expressions *Iexpr* using a colon. In ML parlance, the "`impl`" clause serves the role of *sealing* when defining the package. Adding interfaces to parameters and imports is what allows separate type-checking or compilation. Adding interfaces to package definitions explicitly allows us to use interfaces as semantically meaningful *contracts* between a defining and importing package. Hence *Pimport* serves to both identify a package and seal it behind an interface (called signatures in ML, which are the structural equivalent of our interfaces). Sealing is a form of *hiding*, exposing only those components listed in the interface definition. Finally, "`<:`" in *Idef* defines a nominal subtype relation (§4)

$$
\begin{array}{rll}
\mathit{Idef} & ::= \textbf{iface}\ \mathit{IVname}\ (\overline{\mathit{Ipar} : \mathit{Iexpr}})\ [\ <: \mathit{Iexpr}\ ]\ \mathit{Ibody} & \textit{(interface def.)} \\
\mathit{Ibody} & ::= \{\overline{\mathit{Tdecl}}\ ;\ \overline{\mathit{FunDecl}}\} & \textit{(interface body)} \\
\mathit{Iexpr} & ::= \mathit{Ipar} \mid \mathit{Pvar} \mid \mathit{IVname}(\overline{\mathit{Iexpr}}) & \textit{(interface expr/app.)} \\
\mathit{Pdef} & ::= \textbf{pkg}\ \mathit{PVname}\ (\ \overline{\mathit{Pvar} : \mathit{Iexpr}}\ )\ \textbf{impl}\ \overline{\mathit{Iexpr}}\ \mathit{Pbody} & \textit{(pkg def.)} \\
\mathit{Pbody} & ::= \{\overline{\mathit{PImport}}; \overline{\mathit{Tdef}}; \overline{\mathit{FunDef}}\} & \textit{(pkg body)} \\
\mathit{Pimport} & ::= \textbf{import}\ (\mathit{Pvar} : \mathit{Iexpr})\ \textbf{from}\ \mathit{Pname}(\overline{\mathit{Pvar}}) & \textit{(pkg import/app.)}
\end{array}
$$

Note that scoping means that *Iexpr* in package definitions cannot contain *Ipar* and that *Iexpr* in interface definitions cannot contain *Pvar*. Further, *Pvar* defined by *Pimport* cannot be used in the **impl** clause, so that no types from internal dependencies can be exposed in the exported interface. The following restrictions apply: the outermost *Iexpr* expression cannot be a *Pvar* or *Ipar*. Second, all *Iexpr* arguments in the **impl** and <: clauses must be respectively *Pvar*s and *Ipar*s (which simplifies our presentation).

[3] Providing non-function values is simple, but is omitted as it adds no expressive power here.

In examples we relax the syntax for clarity in the following ways: *Iexpr* follows a first-order expression syntax and we shorten (e.g.) $IVname()$ to $IVname$. Similarly we write "**fun** $f(\overline{x}) = e$" instead of the more formal "**fun** $f : \tau = \lambda\overline{x}.e$" and omit braces { } around package and interface bodies in favour of indentation.

4 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" [20]. 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 [7].

4.1 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 ($IVname$).

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

[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.

nominal (i.e. explicitly defined between interface names). This allows us to reason about backwards compatibility of interfaces (Definition 1), and by extension packages (Definition 2).

Definition 1. *An interface* $\texttt{I-v}N(\overline{Y})$ *is* backwards compatible *with an interface* $\texttt{I-v}M(\overline{X})$ *if* $N \geq M$ *and* $\texttt{I-v}N <: \texttt{I-v}M$ *and* $\overline{Y <: X}$.

Definition 2. *A package* $\texttt{P-v}B(\overline{T})$ *implementing interface* $\texttt{I-v}N(\overline{Y})$ *is* backwards compatible *with package* $\texttt{P-v}A(\overline{S})$ *implementing interface* $\texttt{I-v}M(\overline{X})$ *if* $B \geq A$, $\overline{S <: T}$ *and* $\texttt{I-v}N(\overline{Y})$ *is backwards compatible with* $\texttt{I-v}M(\overline{X})$.

Definition 2 states that packages are contravariant in their parameters and covariant in the implemented interface. This corresponds to subtyping for functors in Standard ML [14] and standard subtyping for function types [20]. Imported packages can be ignored, because they are not part of the exposed interface of a package. Subtyping has more flexibility than shown here, since it also has to deal with sharing and the correspondence between parameters, for example **iface** $\texttt{IX-v2}(W : \texttt{IW-v1}) <: \texttt{IX-v1}(W, W)$ is valid. The full details of subtyping are in Appendix A.1.

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

5 Formal Development and Guarantees

In this section we address the safety claims we have made in the introduction. We introduce *shapes* in §5.1, the basis of our type system, adopting terminology from

Backpack [16]. Building on this, we show how "dependency hell" is eliminated in §5.2, by formalizing package repositories and dependency resolution. Although we do not prove all aspects of the system, dependency resolution is a simple and efficient depth-first traversal. We then show that packages are modularly type-checkable (and compilable) (§5.3) by defining a static semantics that consults a package repository $\mathcal{R}$ for interface but not package definitions. We defer a dynamic semantics to §A.2 where we also state soundness (but leave a proof to future work).

5.1 Shape and Identity

We have argued for interfaces as a means of package *specification* or *type*; sharing of dependencies and abstract data types is expressed through variable names. For example in **pkg** `A-v1`$(W$: `IW-v1`$, P$: `IP-v1(`W`)`$)$ the package for P must have been linked with the package for W. Although all required information is present to treat packages as terms and interfaces as types, we find it convenient to enrich packages and interfaces with information that explicitly expresses equalities of packages such as W. We do this by assigning *labels* $\ell \in \mathit{Label}$ that represent *package identity* (assigned to a particular imported or linked package). Versions bear no relation to labels, as they bear no relation to their identity. We label interface and package constructors (*IVname* and *PVname*), written $\mathcal{I}^\ell$ and $\mathcal{P}^\ell$, as well as (abstract data) type names, written T^ℓ. Note that textual equality of types now includes labels on their type names; this ensures that $P.f \circ Q.g$ is valid only if the `TWombat` types exposed by f and g are identically labelled.

Labels further make it easy to resolve **import**s and to verify the validity of *Iexpr*s. We call these enriched structures *shapes*. One way of handling type identity is *manifest types* [18] and substitution of type names in signatures. However, we support nominal subtyping, which extends to interface parameters and not just to the body of the interface (or the *signature*). For example, both `IW-v1` and `IW-v2` may export a type `TWombat`, but if W_1 : `IW-v1` and W_2 : `IW-v2`, then `IP-v1`(W_1) and `IP-v2`(W_2) represent different interfaces (with `IP-v1`(W_2) <: `IP-v1`(W_1) if `IW-v2` <: `IW-v1`). Hence we model our equality constraints with interface names instead of type names.

Type-checking and compilation of packages is the process of *shape checking*, which given a *Idef* or *Pdef* produces an interface or package shape respectively:

$$I \ ::= \mathcal{I}^\ell(\overline{I})\{\overline{\textbf{type } T^\ell}; \overline{\textbf{fun } f : \tau}\} \qquad \textit{(interface shape)}$$

$$P \ ::= \mathcal{P}^\ell(\overline{I})\{\overline{\textbf{type } T^\ell = \tau}; \overline{\textbf{fun } f : \tau = e}\} : I \mid \mathcal{P}^\ell(\overline{I})\{*\} : I \qquad \textit{(package shape)}$$

Shapes have more or less the same structure as their respective interface or package definitions. Most notable is the addition of labels ($\mathcal{I}^\ell$, $\mathcal{P}^\ell$ and T^ℓ) and the addition of *shape bodies* $\{\ldots\}$ for interfaces and external package dependencies (internal dependences use $\{*\}$ as their bodies are unknown during shaping). Bodies include labelled types T^ℓ, with appropriate extensions to τ and e. Moreover, variables *Ipar* and *Pvar* are resolved to interface shapes I using a context Δ (defined below), so an *Iexpr* `IP-v1`(W_2) becomes an interface

shape $\texttt{IP-v1}^{\ell_1}(\texttt{IW-v2}^{\ell_2}()\{B_2\})\{B_1\}$ for some shape bodies B_1, B_2 and labels ℓ_1, ℓ_2. Type projections $W_2.\texttt{TWombat}$ are now of the form $\texttt{TWombat}^{\ell}$, while term projections are now on interface shapes I, e.g. $\texttt{IW-v2}()\{\ldots\}.f$.

To keep the size of inference rules manageable, we use $\mathcal{I}$ instead of *IVname* for interface constructors and $\mathcal{P}$ instead of package constructors *PVname* and *Pname*. We use B to range over shaped bodies (corresponding to *Ibody* and *Pbody*). We use S to range over either I, P or B, and X to range over both *Ipar* and *Pvar*. We use the following contexts in our shape-checking rules:

$$\begin{array}{ll} \mathcal{R} ::= & \overline{\mathcal{I} \mapsto \mathit{Idef}} \quad \overline{\mathcal{P} \mapsto \mathit{Pdef}} \\ \Delta ::= & \overline{X \mapsto I} \\ \Gamma ::= & \overline{x \mapsto \tau} \quad \overline{T \mapsto (\tau \mid \mathbf{abs})} \end{array}$$

Contexts $\mathcal{R}$, Δ and Γ are each finite partial maps: $\mathcal{R}$ maps constructors $\mathcal{I}$ and $\mathcal{P}$ (*IVname* and *PVname*) to their respective interface and package definitions (*Idef* and *Pdef*), Δ maps variables X (*Pvar* and *Ipar*) to interface shapes. Core-language context Γ maps variable and function names to types, and abstract types names either to types if locally defined or to **abs** otherwise. We further use $\mathcal{L} = [\ell]$ as an optional label (ℓ or $\bullet$), in order to label type declarations and definitions **type** $T[= \tau]$. Finally, we write $\Delta[X \mapsto I]$ for context extension, $\mathit{dom}(\Delta)$ for all X and $\mathit{range}(\Delta)$ for all I where $X \mapsto I \in \Delta$, and $\bullet$ for the empty context. This applies similarly to $\mathcal{R}$ and Γ. We use judgements of the following form during shape-checking

$$\mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash E \rhd S$$

to translate a source-level term E (package, interface or core-language) into a shape S. We also use judgements

$$\begin{array}{l} \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash E \overset{import}{\rightsquigarrow} I \\ \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash E \overset{impl}{\rightsquigarrow} I \end{array}$$

to compute shape information for the purpose of dependency resolution. Finally, we presume the core-language type system has judgements of the form:

$$\Gamma \vdash t : \tau$$

Finally, we abbreviate $\bullet; \bullet$ as just $\bullet$.

5.2 Satisfiability

In this section we formalize our *satisfiability* guarantee, by which we mean the guarantee that any $\mathcal{P} \mapsto \mathit{Pdef} \in \mathcal{R}$ can be installed onto a user system. To do so we restrict repositories $\mathcal{R}$. The first restriction is that of *well-formedness*, which states that a repository must have all information available for any package or interface to be judged by the shaping rules from Figure 3 (§5.3) and Figure 4

$$\frac{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \mathcal{I}(\overline{E}) \rhd \mathcal{I}^{\ell}(\overline{I}) \qquad \overline{X \mapsto I'} \in \Delta}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \textbf{import } Y : \mathcal{I}(\overline{E}) \textbf{ from } \mathcal{P}(\overline{X}) \overset{import}{\rightsquigarrow} \mathcal{P}^{\ell}(\overline{I'})\{*\} : \mathcal{I}^{\ell}(\overline{I})} \text{ (Dep-Pimport)}$$

$$\frac{\mathcal{R};\bullet \vdash \textbf{pkg } \mathcal{P}(X_1 : E_1, \ldots, X_n : E_n) \textbf{ impl } \mathcal{I}(\overline{E'})\ \{\ldots\} \rhd \mathcal{P}^{\ell}(\overline{I})\{\ldots\} : \mathcal{I}^{\ell}(\overline{I'})}{\mathcal{R};\bullet \vdash \textbf{pkg } \mathcal{P}(X_1 : E_1, \ldots, X_n : E_n) \textbf{ impl } \mathcal{I}(\overline{E'})\ \{\ldots\} \overset{impl}{\rightsquigarrow} \mathcal{P}^{\ell}(\overline{I})\{*\} : \mathcal{I}^{\ell}(\overline{I'})} \text{ (Dep-Pimpl)}$$

Fig. 2. Shaping Rules for Package Imports and Interface Exports

(Appendix A.1). These rules succeed only if all *dependencies* (Definition 3) are *satisfied* in $\mathcal{R}$ (Definition 6, Definition 7), and if all dependencies are themselves well-formed (i.e. shape- and type-correct) according to the rules. There are further restrictions on circularity, covered in Appendix B.

Definition 3. *A **dependency** is either a package dependence (Definition 4), or an interface dependence (Definition 5).*

Definition 4. *Package `P` **depends** on package `Q` if `P` contains some `import` with a shape S_1, determined by* (Dep-Pimport)*, that is a supertype (or supershape) of shape S_2 exported by `Q` according to* (Dep-Pimpl) *from Figure 2 ($S_2 <: S_1$, using rule* (SubShape-Pkg) *from Figure 7 in Appendix A.1).*

Definition 5. *A package or interface **depends** on another interface $\mathcal{I}$ whenever, during the shaping of the package or interface, the rule* (Shape-Iexpr) *(from Figure 4 in Appendix A.1) is applied for an Iexpr with interface constructor $\mathcal{I}$.*

Definition 6. *A package dependence arising from a package P is **satisfied** in a repository $\mathcal{R}$ whenever there is some package Q in $\mathcal{R}$ on which P depends (i.e. when there is at least one matching package for every package dependency).*

Definition 7. *A dependence on an interface $\mathcal{I}$ is **satisfied** in a repository $\mathcal{R}$ whenever $\mathcal{I} \mapsto Idef \in \mathcal{R}$.*

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 $\mathcal{P} \mapsto Pdef \in \mathcal{R}$ and all of its dependencies. Such installation plans are always computable (by definition we can always choose $\mathcal{R}' = \mathcal{R}$), but $\mathcal{R}$ may be very large, so we wish to compute some repository $\mathcal{R}'$ that is preferably smaller than $\mathcal{R}$ (Definition 8).

Definition 8. *An **installation plan** for a package $\mathcal{P}$, with regard to a (shared, central) package repository $\mathcal{R}$, is some repository $\mathcal{R}' \subseteq \mathcal{R}$ such that $\mathcal{P} \mapsto Pdef \in \mathcal{R}'$ where $\mathcal{P}$ is satisfied in $\mathcal{R}'$.*

In our system we have expressed dependencies in terms of interfaces, and express sharing of dependencies through abstraction. Hence when we see an **import** we are free to choose *any* compatible version, without regard for other

packages or the current state of the system. Conflicts are therefore not possible, and we free ourselves of dependency hell. Moreover, checking for satisfiability of a package, or computing an installation plan $\mathcal{R}'$ for some package $\mathcal{P}$ is both efficient and almost trivial in our system. First, we assume shaping (§5.3) and subtyping (Appendix A.1) are both decidable in polynomial time.[5] With the help of our subtype algorithm we can readily compute an installation plan $\mathcal{R}'$ from a given repository $\mathcal{R}$ (Theorem 1).

Theorem 1. *A (best-effort) installation plan can be computed in polynomial time for any package $\mathcal{P} \mapsto Pdef \in \mathcal{R}$.*

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 shape matching is decidable in polynomial time by assumption. □

Our installation theorem and its PTIME complexity trivially follow from the removal of version conflicts, a fact well known for version-based systems [3]. 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 $\mathcal{P} \mapsto Pdef \in \mathcal{R}$ 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 $\mathcal{P}$.

5.3 Portability

To argue that packages can be compiled separately and reasoned about modularly, we show shaping rules of packages that look up interface but not package definitions in $\mathcal{R}$. We explain package shaping rules in turn, which can also be found in Figure 3 in Appendix A.1.

We start by shaping package definitions:

$$\frac{\begin{array}{c}\mathcal{R};\Delta_0;\bullet \vdash E_1 \rhd I_1 \quad \ldots \quad \mathcal{R};\Delta_{n-1};\bullet \vdash E_n \rhd I_n \quad (\Delta_i \equiv [X_1 \mapsto I_1, \ldots, X_i \mapsto I_i]) \\ \mathcal{R};\Delta_n;\ell;\bullet \vdash \{body\} \rhd \{B\} \qquad \ell \text{ fresh} \\ \mathcal{R};\Delta_n;\ell;\bullet \vdash E \rhd \mathcal{I}^{\ell'}(\overline{I_j})\{[\ell \mapsto \ell']B'\} \qquad \{B\} <: \{B'\}\end{array}}{\mathcal{R};\bullet \vdash \mathbf{pkg}\ \mathcal{P}(X_1 : E_1, \ldots, X_n : E_n)\ \mathbf{impl}\ E\ \{body\} \rhd \mathcal{P}^{\ell}(\overline{I_i})\{B\} : \mathcal{I}^{\ell}(\overline{I_j})\{B'\}}\ (\textsc{Shape-Pdef})$$

[5] Although interface shapes may grow exponentially due to substitution during shaping packages or interfaces, sub-terms are also shared exponentially often in such scenarios. Our matching algorithm (Appendix A.1) copes well with this, as checking for label equality suffices on the second and subsequent encounters of some sub-term.

Rule (Shape-Pdef) handles scoping of package parameters, computing a shape I_i of the form $\mathcal{I}^{\ell_i}(\ldots)\{B\}$ for each package argument $X_i : E_i$. We add $X_i \mapsto I_i$ to Δ for any subsequent arguments $X_{i+1} : E_{i+1}$, as well as the **impl** clause and the package body. A fresh label ℓ is allocated that represents the package itself, used when shaping the package body (for (Shape-Type-Def)). The shape body must be a sub-shape of the shapes from the **impl** clause $\{B\} <: \{B'\}$, with a proper relabelling of ℓ' to ℓ. Here ℓ' is a fresh label allocated by the (Shape-Idef) rule. Subtyping, or sub-shaping, is covered in detail in Appendix A.1. Next is (Shape-Pimport), which handles scoping of imports:

$$\frac{\begin{array}{c}\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash E_1 \rhd I \\ \mathcal{R};\Delta[X \mapsto I];\mathcal{L};\Gamma \vdash \{\ldots\} \rhd \{B\}\end{array}}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \{\textbf{import } X : E_1 \textbf{ from } E_2;\ldots\} \rhd \{B\}} \text{ (Shape-Pimport)}$$

The rule computes an interface shape I (which contains a fresh label, allocated by (Shape-Iexpr) below) from the interface expression E_1 and adds $X \mapsto I$ to Δ for the remainder of the body. Package application E_2 is left untouched, which is our argument for modular type-checking. However, it is processed by (Dep-Pimport) from Figure 2 for dependency resolution and (static or dynamic) linking. Interface applications *Iexpr* occur in both package and interface definitions, shown below:

$$\frac{\begin{array}{c}\ell \text{ fresh} \qquad \overline{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash E \rhd I} \\ \mathcal{R};\bullet \vdash \textbf{iface } \mathcal{I}\ldots \rhd \mathcal{I}'^{\ell'}(\overline{I'})\{B'\} \qquad \mathcal{I}^{\ell}(\overline{I})\{\} \text{ wf}\end{array}}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \mathcal{I}(\overline{E}) \rhd \mathcal{I}^{\ell}(\overline{I})\{[\ell' \mapsto \ell][\overline{lab(I') \mapsto I}]B'\}} \text{ (Shape-Iexpr)}$$

The rule computes a shape $\mathcal{I}^{\ell}(\overline{I})\{\phi\sigma B'\}$ with a fresh label ℓ, arguments $\overline{I}$ and a body $\phi\sigma B'$. To compute the body we look up the actual definition of $\mathcal{I} \mapsto \textbf{iface } \mathcal{I}\ldots \in \mathcal{R}$ and assign a new shape $\mathcal{I}'^{\ell'}(\overline{I'})\{B'\}$ to the interface definition. This new shape has distinct labels from the package (or interface) that we are shaping, so we use a renaming $\phi = [\ell \mapsto \ell']$ where we define $lab(\mathcal{I}^{\ell}(\ldots)\{\ldots\}) = \ell$. This effectively relabels abstract data type names with labels from the arguments instead of the labels from the formal parameters. Relabelling is formally defined in the Technical Appendix, and rewrites types $T^{\ell'}$ to T^{ℓ} whenever $\ell' \mapsto \ell \in \phi$. We further need to substitute interface shapes I, which we do with $\sigma = [\overline{lab(I') \mapsto I}]$, and essentially replaces a shape (sub-)term $\mathcal{I}'^{\ell'}(\ldots)\{\ldots\}$ with I whenever $\ell' \mapsto I \in \sigma$. Substitution of the arguments, as opposed to relabelling, is necessary to properly replace interface shapes I ocurring in the interface body as part of term projections. Interface constructors are a form of dependent function types, since the resulting shape depends on the values of the arguments, and (Shape-Iexpr) is the elimination rule.

The new shape $\mathcal{I}^{\ell}(\overline{I})\{\}$ is checked for well-formedness (Appendix A.1), to ensure that the *Iexpr* is actually valid, using a nominal subtype relation that respects sharing (Appendix A.1). This relation is defined purely on interface names and parameters, and ignores bodies (so we leave it empty).

Rule (Shape-Type-Def) handles type definitions by labelling types T:

$$\frac{T^\ell \notin \mathrm{dom}(\Gamma) \qquad \Delta; \Gamma; \ell; \Gamma \vdash \tau \rhd \tau' \qquad \mathcal{R}; \Delta; \ell; \Gamma[T^\ell = \tau'] \vdash \{body\} \rhd \{B\}}{\mathcal{R}; \Delta; \ell; \Gamma \vdash \{\mathbf{type}\ T = \tau; body\} \rhd \{\mathbf{type}\ T^\ell = \tau'; B\}} \text{ (Shape-Type-Def)}$$

The rule also guards against duplications with $T^\ell \notin dom(\Gamma)$ and processes τ to label any uses of type names T' and handle type projections. The rule for function definitions is similar:

$$\frac{f \notin \mathrm{dom}(\Gamma) \qquad \Delta; \Gamma; \mathcal{L}; \Gamma \vdash \tau \rhd \tau' \qquad \Delta; \Gamma; \mathcal{L}; \Gamma \vdash e \rhd e' \qquad \Gamma \vdash e' : \tau' \qquad \mathcal{R}; \Delta; \mathcal{L}; \Gamma[f : \tau'] \vdash \{body\} \rhd \{B\}}{\mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash \{\mathbf{fun}\ f : \tau = e; body\} \rhd \{\mathbf{fun}\ f : \tau' = e; B\}} \text{ (Shape-Fun-Def)}$$

Rules (Shape-Type-Def) and (Shape-Fun-Def) each perform shaping of types, in order to resolve type projections. To do this we need to define shaping on arrow types:

$$\frac{\mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash \tau_1 \rhd \tau_1' \qquad \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash \tau_2 \rhd \tau_2'}{\mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash \tau_1 \to \tau_2 \rhd \tau_1' \to \tau_2'} \text{ (Shape-Fun-Type)}$$

We assume a similar rule for terms e to handle variables X. Type projection labels abstract data type names by looking up the shape for variable X in Δ and finding a definition $\mathbf{type}\ T^\ell$:

$$\frac{X \mapsto \mathcal{I}^\ell(\overline{I})\{\ldots; \mathbf{type}\ T^\ell; \ldots\} \in \Delta}{\mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash X.T \rhd T^\ell} \text{ (Shape-ProjType)}$$

We also need to resolve and label bare type names T, and check that they are properly declared or defined:

$$\frac{T^\ell \in \mathrm{dom}(\Gamma)}{\mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash T \rhd T^\ell} \text{ (Shape-Abs-Type)}$$

We do not handle term projections directly during shaping, which is instead done by (Type-TermProj) in §A.1 (since we cannot directly inline the function body in terms e). However, we do need to resolve variables X in term projections $X.f$:

$$\frac{X \mapsto I \in \Delta}{\mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash X \rhd I} \text{ (Shape-Var)}$$

Shaping of interfaces and other core-language typing rules are in Figure 4 and Figure 5 in Appendix A.1. Finally, we have to handle the empty body:

$$\frac{}{\mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash \{\} \rhd \{\}} \text{ (Shape-Empty-Body)}$$

6 Expressiveness

It seems that our system and traditional version-based systems are not equally expressive, and comparing them is not entirely straightforward. For example, encoding a version-based system into $\prod$ automatically would be non-trivial, as the interface between packages may vary between versions and sharing is implicit. Conversely, encoding $\prod$ into a system based on versions may also be non-trivial if we want to avoid dependency hell, as version based systems would falsely assume sharing of our internal dependencies.

However, our system is more expressive than traditional systems in that packages in $\prod$ can express dependencies on future versions of a dependency if that dependency implements (a subtype of) the required interface. Further, by mechanically removing import sealing boundaries during dependency resolution we can express a form a *dependent compatibility*, where compatibility with one package depends on the interface implemented by a dependency, rather than the interface required by the importing package. This gives a form of disjunction that is present in some version-based package systems. We cover this further in the Technical Appendix [13].

However, traditional package systems (such as Debian) are more expressive in that they support dependencies between packages written in different languages, dependencies on services, configuration data or command-line programs. To somewhat narrow the expressivity gap we introduce some extensions below.

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 $\prod$ 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 ranges are part of interfaces.

Conflicts Traditional package systems can often express *conflicts* between packages [6], 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. Such an extension means 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 [15]. For example, we could write:

```
import F : IFoo-v1(B) from Foo-any(B)
import B : IBar-v1(F) from Bar-any(F)
```

Recursive modules, and the interplay with mutable variables and circular definitions have been extensively studied in [9][8][21][15][11] and others.

7 Discussion and Related Work

Module Systems As alluded to earlier, our package system is rather much like module systems such as that of Standard ML [14] or OCaml, MixML [10], Units [12] or component-based systems [17]. 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 importing and imported 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.

Backpack Backpack is a package system for Haskell [16], 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 Functional package managers such as Nix [5] 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 Environments Virtual package environments such as virtualenv [4] 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* [2] 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.

8 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 in-expressiveness 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, 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. Vagrant, https://www.vagrantup.com/
3. 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)
4. Bicking, I.: Virtualenv, https://virtualenv.pypa.io/en/latest/
5. van der Burg, S.: A Reference Architecture for Distributed Software Deployment. Ph.D. thesis (2013)
6. di Cosmo, R.: Report on formal management of software dependencies (05 2012)
7. Dijkstra, E.W.: On the role of scientific thought. Selected Writings on Computing: A personal Perspective pp. 60–66 (1982)
8. Dreyer, D.: A type system for well-founded recursion. ACM SIGPLAN Notices 39(1), 293–305 (2004)
9. Dreyer, D.: Understanding and evolving the ML module system (05 2005)
10. Dreyer, D., Rossberg, A.: Mixin' up the ML module system (09 2008), http://dx.doi.org/10.1145/1411203.1411248
11. Duggan, D.: Type-safe linking with recursive DLLs and shared libraries. ACM Transactions on Programming Languages and Systems 24(6), 711–804 (2002)
12. Flatt, M., Felleisen, M.: Units: Cool modules for hot languages. ACM SIGPLAN Notices 33(5), 236–248 (1998)
13. Florisson, M., Mycroft, A.: Towards a theory of packages: Technical appendix, http://www.cl.cam.ac.uk/∼mbf24/PackageAppendix.pdf
14. Harper, R.: Programming in Standard ML (2011)
15. 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)
16. 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)
17. Lau, K.K., Wang, Z.: Software component models. Proceeding of the 28th international conference on Software engineering – ICSE '06 (2006)
18. Leroy, X.: Manifest types, modules, and separate compilation. Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94 (1994)
19. 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)
20. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA (12 2002)
21. Russo, C.V.: Recursive structures for Standard ML. ACM SIGPLAN Notices 36(10) (2001)

A Semantics

We define a static (§A.1) and dynamic semantics (§A.2) in order to state soundness (which we do not prove). Soundness is our argument for type-safety of package linking.

A.1 Static Semantics

In §5.3 of the paper we covered package shaping in detail. We now also define shaping for interface definitions and include well-formedness constraints, which in turn relies on nominal sub-shaping rules, analogous to subtyping for shapes.

Package Shaping Package shaping rules are repeated from §5.3 in a single Figure 3.

Interface Shaping The shaping rules for interface definitions are shown in Figure 4. The first rule, (Shape-Idef), does for interfaces what (Shape-Pdef) does for packages. Scoping of parameters is handled in the same way, and Δ_i is again a macro for $X_1 \mapsto I_1, \ldots, X_i \mapsto I_i$. Since $<:$ is optional the sub-shape check is also optional. Rules (Shape-Type-Decl) and (Shape-Fun-Decl) are analogous to the package shaping rules (Shape-Type-Def) and (Shape-Fun-Def), but with omitted definitions τ and e respectively. Rules for shaping interface applications or variables are identical to the package shaping rules.

Core-Level Typing Figure 5 shows core-level typing judgements. The first rule (Type-ProjTerm) handles projection of term components (functions f) from variables X. Rules (Type-Reify) and (Type-Abstract) are inverses of each other. The first removes abstraction boundaries of abstract data types within a package. That is, when it is known that **type** $\mathrm{T} = \tau$ and further $e : T$, then we can conclude $e : \tau$. The latter goes the other way, i.e. given a typing $e : \tau$, it concludes $e : T^\ell$ if $T^\ell = \tau$. Rule (Shape-CoreVar) handles term and function variables from the core language.

Sub-Shaping of Package and Interface Bodies This sub-section defines subtyping for shape bodies. These rules are used to check whether an package/interface body is a sub-shape of an implemented interface/super-shape interface, by rules (Shape-Pdef) and (Shape-Idef) respectively. The rules are simple because both bodies are assumed to have identical labels and interface shapes, as (Shape-Iexpr) substitutes shape arguments in an *Iexpr* for formal parameters from an *Idef*. We again use B to range over shape bodies. Rule (Sub-Body) defines width, depth and permutation subtyping for bodies. The remaining rules require definitions to match declarations. Rules for matching declarations with equivalent declarations are handled by the reflexivity rule (Sub-Refl).

Well-Formedness We use the rule (WF-App-Iface) during shaping, to ensure well-formedness of interface shapes constructed from interface expressions *Iexpr*. The rule validates the shape arguments computed from *Iexpr*s against the formal parameters computed from *Idef*s. Validation checks whether the argument shape is a sub-shape of the parameter shape, and ensures sharing constraints are respected with the help of a sharing context $\mathcal{E}$. Well-formedness of body $\{B\}$ can be ignored, because this is checked during shaping of the interface definition.

$$\frac{\mathcal{R} \vdash \textbf{iface}\ \mathcal{I}(\overline{E})\ [<:\ E'\ ]\ \{\ldots\} \rhd \mathcal{I}^{\ell'}(\overline{I'_i})\{B'\} \qquad \mathcal{R};\bullet \vdash I_1 <: I'_1 \dashv \mathcal{E}_1 \quad \ldots \quad \mathcal{R};\mathcal{E}_{n-1} \vdash I_1 <: I'_1 \dashv \mathcal{E}_n \qquad 1 \leq i \leq n}{\mathcal{R} \vdash \mathcal{I}^{\ell}(\overline{I_i})\{B\}\ \text{wf}}$$

(WF-App-Iface)

Sub-Shape Checking of Package and Interface Shapes Here we define sub-shaping (subtyping for shapes) of package and interface shapes. Sub-shaping for package shapes is used during dependency resolution (§5.2), while sub-shaping for interface shapes happens during shaping, when checking for well-formedness of interface shapes in (WF-App-Iface). The key consideration is to handle sharing, for which we use a sharing context $\mathcal{E}$. Further, we can ignore interface bodies, and instead rely on a purely nominal sub-shape relation. This is because the bodies are guaranteed to be sub-shapes by the definition of our shaping rules. Sub-shaping rules are shown in Figure 7.

The rules handle sharing by threading a sharing context $\mathcal{E}$ through the rules. Rule (SubShape-Pkg) handles sub-shaping of package shapes, used to match imports with package definitions. The shape of the imported package must be a sub-shape of the shape of the importing package. Since package shapes are contravariant in their arguments we first check that $\mathcal{R};\mathcal{E}_{i-1} \vdash I'_i <: I_i \dashv \mathcal{E}_i$ using sharing context $\mathcal{E}_i = [lab(I_1) \mapsto I'_1, \ldots, lab(I_n) \mapsto I'_i]$. Package shapes are covariant in the exported interface, so we need to check that $\mathcal{I}_1^{\ell_1}(\overline{I})\{B'_1\} <: \mathcal{I}_2^{\ell_2}(\overline{I'})\{B'_2\}$. We do this by computing a correspondence M between the arguments to $\mathcal{I}_1$ and $\mathcal{I}_2$. We then use substitution with $\mathcal{E}_n$ (which is valid because we know that $I'_i <: I$) to check for an exact sharing correspondence between the two interfaces. Substitution is necesary to handle the *dependence* that exists between the parameters to $\mathcal{P}^{\ell_1}$ and the arguments to $\mathcal{I}_1$. Without substitution subtyping would not work, as the arguments are contravariant and the result covariant. For example, the relation would not hold for

$$\mathcal{R};\mathcal{E} \vdash \mathcal{P}^{\ell}(T)\{B\} : \mathcal{I}^{\ell}(T) <: \mathcal{P}^{\ell'}(S)\{B'\} : \mathcal{I}^{\ell'}(S) \dashv \mathcal{E}'$$

whenever $\mathcal{R};\mathcal{E} \vdash S <: T \dashv \mathcal{E}'$ with $S \neq T$. However, we want the relation to hold, because the shape of the imported package (left) is really $\mathcal{I}^{\ell}(S)$, which is trivally a sub-shape of $\mathcal{I}^{\ell'}(S)$ (the required shape on the right).

Next are rules (SubShape-Iface-1) and (SubShape-Iface-2), which update $\mathcal{E}$ whenever they see a new label ℓ: if the label has not been seen before, the rules apply recursively (SubShape-Iface-1). If the label has been seen before, the rules

require that the left-hand shape is equivalent to what was seen before (SubShape-Iface-2). This last rule ignores the remainder of the shape on the right-hand side $\mathcal{I}_2^{\ell_2}(\overline{I'})\{B'\}$, because it has been processed before by (SubShape-Iface-1).

Formal Parameter Mapping When sub-shaping package and interface shapes, we covered how sharing is handled. We also used a mapping M to make arguments from a sub-shape correspond to the arguments of a super-shape interface constructor. In particular, when we want to know whether $I_2 <: I_1$ for some interface shapes I_2 and I_1, I_2 may have as outermost term an interface constructor that is a nominal subtype of the interface constructor in I_1 (i.e. $\mathcal{I}_2(\overline{I_2})\{B_2\} <: \mathcal{I}_1(\overline{I_1})\{B_1\}$). To define sub-shaping, we have to understand how the parameters of $\mathcal{I}_2(\overline{I_2})$ correspond to those of $\mathcal{I}_1(\overline{I_1})$. For example, consider the following interface definitions:

```
iface IWombat-v1(S : IString-v1)
    ...
iface IWombat-v2(S : IString-v1, G : IGrass-v1) <: IWombat-v1(S)
    ...
```

To define sub-shaping, we have to relate arguments of `IWombat-v2` to arguments of `IWombat-v1`. To do so we define a *mapping* $\mathcal{I}_2 \overset{\mathcal{M}}{\to} \mathcal{I}_2$ for example to map from m argument positions from `IWombat-v1` to n argument positions from `IWombat-v2`. We use these mapping functions to transform arguments vectors as follows:

$$[1 \mapsto x_1, \ldots, m \mapsto x_m]\overline{I_i} = \overline{I_{x_i}}$$

The rules for computing these functions are listed in Figure 8.

A.2 Dynamic Semantics

Here we give an operational semantics that performs dynamic linking. We first extend our grammar with *programs* of the form

$$\textbf{pkg}\ PVname\ \{Pbody\}$$

We require the last function definition in *Pbody* to be

$$\textbf{fun}\ main : Unit \to Int = \lambda().e$$

Package *values* are (closed) package bodies of the form

$$\begin{aligned} PkgVal &::= \{\textbf{type}\ T^\ell = \tau_{val}; \textbf{fun}\ f : \tau_{val} = v\} \\ \tau_{val} &::= \tau_1 \to \tau_2 \mid T^\ell \end{aligned}$$

where v is a core-language λ-value. There is no real operational need to keep type components, their labels or indeed any type as part of package values. However, since packages include the type components of a package in the shape body, we also include type components and types as part of package values in

order to state our preservation theorem. Further, in order to construct package values that include labels, we assume that packages have been annotated with the labels from their shapes. We call such packages *label annotated*.

Reduction rules are of the form

$$\mathcal{R}; \Phi; \mathcal{P} \vdash e \longrightarrow v$$

where R is a well-formed repository, Φ is a mapping to connect internal dependencies with package implementations (covered next), and $\mathcal{P}$ keeps track of the package constructor during package body evaluation, in order to find shapes for internal dependencies using Φ.

We assume a dependency mapping Φ from package implementations and imports to (respectively) **import** and **impl** package shapes, constructed during shaping with the $\rightsquigarrow$ rules from Figure 2 in §5.2 (we omit modified shaping rules that construct this mapping as they are trivial). The mapping is of the following form:

$$\Phi ::= \overline{\mathcal{P} \overset{impl}{\rightsquigarrow} S} \quad \overline{(\mathcal{P}_1, X, \mathcal{P}_2) \overset{import}{\rightsquigarrow} S}$$

In particular, rule (E-Import) uses Φ to look up an appropriate package implementation for an **import** statement. Here $\mathcal{P}$ and $\mathcal{P}_1$ are versioned $PVname$ package constructors. However, $\mathcal{P}_2$ is an unversioned $Pname$, leaving the evaluation rules to determine an appropriate version. Here $\mathcal{P}_1$ is the dependant (the importing package, for which a concrete version number is known), and $\mathcal{P}_2$ the dependee (the imported package, for which we still need to find an appropriate version). In either case S is a package shape corresponding to the **impl** and **import** shapes respectively.

We now define a call-by-value small-step operational semantics for programs and packages, shown in Figure 9. In the rules we use $\mathcal{P}$ to range over package constructors that are appropriate versioned or unversioned depending on context. Rules for core-language term reduction $e \longrightarrow e'$ are omitted, which take place after all packages have been elaborated.

Soundness We are now in a position to state soundness, although we do not provide a proof. Suppose $\mathcal{R}$ is a well-formed repository, and Φ is a shape mapping for all package definitions (**impl**) and internal dependencies (**import**). Theorem 2 states progress of package evaluation, and Theorem 3 states preservation of shape of package terms under evaluation.

Theorem 2 (Package Progress)**.** *Let p be a label-annotated package* body *that is well-shaped with respect to repository $\mathcal{R}$. Then either p is a value, or there is some p' such that $\mathcal{R}; \Phi; \mathcal{P} \vdash p \longrightarrow p'$.*

Theorem 3 (Package Preservation)**.** *Let p be a label-annotated package* body *with $\mathcal{R}; \bullet \vdash p \rhd B$ and $\mathcal{R}; \Phi; \mathcal{P} \vdash p \longrightarrow p'$, then also $\mathcal{R}; \bullet \vdash p' \rhd B$.*

We leave progress and preservation theorems of programs and terms to future work.

B Repositories and Restrictions

In §5.2 we covered how dependencies can be automatically resolved from a repository of packages $\mathcal{R}$. We said that all packages and interfaces in the repository must be well-typed and satisfied in $\mathcal{R}$, so that any $\mathcal{P} \mapsto \mathit{Pdef} \in \mathcal{R}$ could be installed onto a any user system $\mathcal{S}$. In this section we cover additional restrictions that are necessary to ensure that packages will in fact install and link correctly (§A.2). Firstly, there must be no *dependency cycles* in repositories because this may lead to non-termination of package evaluation in our operational semantics. We might imagine changing the operational semantics and having the linker deal with cycles, this would not be compatible with mutable state: if `Foo` requires a `Bar` with fresh mutable state, and `Bar` requires a `Foo` with fresh mutable state, there is no finite package instantiation that satisfies such demands.

For these reasons we impose the restriction that no cycle is allowed between packages. We can do this in two ways: we could impose the restriction that a package is rejected if there is *always* a dependency cycle, or if there is *some* dependency cycle. While the former is more expressive, it undoes the simplicity of our dependency resolution algorithm, as resolution can no longer just pick any suitable package version to resolve an import. To maintain simplicity and efficiency we reject cycles *conservatively*. We do this by constructing a graph $G = (V, E)$ where V consists of all exported shapes $\mathcal{P}^\ell(\overline{I})\{*\} : \mathcal{I}^\ell(\overline{I})\{B\}$ for all packages in $\mathcal{R}$, as defined by rule (Dep-Pimpl) (Figure 2 from §5.2). The edges E are constructed from the package dependencies arising from **import**s, as defined by rule (Dep-Pimport) from §5.2. That is, whenever a package `A-vX` has a dependence on package `B-vY`, we add an edge between *all* shapes exported by `A-vX` to the matching shape for the import exported by `B-vY`. Any cycles in the resulting graph indicate errors (indicating ill-formedness of the repository). There are similar restrictions on circularity between interfaces, where the nodes are interface constructors $\mathcal{I}$ (IVname), and there is an edge between two nodes $\mathcal{I}_1$ and $\mathcal{I}_2$ whenever $\mathcal{I}_1$ occurs in an Iexpr of $\mathcal{I}_2$.

$$\frac{\mathcal{R};\Delta_0;\bullet \vdash E_1 \rhd I_1 \quad \ldots \quad \mathcal{R};\Delta_{n-1};\bullet \vdash E_n \rhd I_n \quad (\Delta_i \equiv [X_1 \mapsto I_1, \ldots, X_i \mapsto I_i]) \\ \mathcal{R};\Delta_n;\ell;\bullet \vdash \{body\} \rhd \{B\} \qquad \ell \text{ fresh} \\ \mathcal{R};\Delta_n;\ell;\bullet \vdash E \rhd \mathcal{I}^{\ell'}(\overline{I_j})\{[\ell \mapsto \ell']B'\} \qquad \{B\} <: \{B'\}}{\mathcal{R};\bullet \vdash \textbf{pkg } \mathcal{P}(X_1 : E_1, \ldots, X_n : E_n) \textbf{ impl } E\ \{body\} \rhd \mathcal{P}^{\ell}(\overline{I_i})\{B\} : \mathcal{I}^{\ell}(\overline{I_j})\{B'\}} \text{ (Shape-Pdef)}$$

$$\frac{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash E_1 \rhd I \\ \mathcal{R};\Delta[X \mapsto I];\mathcal{L};\Gamma \vdash \{\ldots\} \rhd \{B\}}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \{\textbf{import } X : E_1 \textbf{ from } E_2; \ldots\} \rhd \{B\}} \text{ (Shape-Pimport)}$$

$$\frac{\ell \text{ fresh} \qquad \overline{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash E \rhd I} \\ \mathcal{R};\bullet \vdash \textbf{iface } \mathcal{I} \ldots \rhd \mathcal{I}^{\ell'}(\overline{I'})\{B'\} \qquad \mathcal{I}^{\ell}(\overline{I})\{\} \text{ wf}}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \mathcal{I}(\overline{E}) \rhd \mathcal{I}^{\ell}(\overline{I})\{[\ell' \mapsto \ell][\overline{lab(I') \mapsto I}]B'\}} \text{ (Shape-Iexpr)}$$

$$\frac{T^{\ell} \notin \text{dom}(\Gamma) \qquad \Delta;\Gamma;\ell;\Gamma \vdash \tau \rhd \tau' \\ \mathcal{R};\Delta;\ell;\Gamma[T^{\ell} = \tau'] \vdash \{body\} \rhd \{B\}}{\mathcal{R};\Delta;\ell;\Gamma \vdash \{\textbf{type } T = \tau; body\} \rhd \{\textbf{type } T^{\ell} = \tau'; B\}} \text{ (Shape-Type-Def)}$$

$$\frac{f \notin \text{dom}(\Gamma) \qquad \Delta;\Gamma;\mathcal{L};\Gamma \vdash \tau \rhd \tau' \qquad \Delta;\Gamma;\mathcal{L};\Gamma \vdash e \rhd e' \\ \Gamma \vdash e' : \tau' \qquad \mathcal{R};\Delta;\mathcal{L};\Gamma[f : \tau'] \vdash \{body\} \rhd \{B\}}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \{\textbf{fun } f : \tau = e; body\} \rhd \{\textbf{fun } f : \tau' = e; B\}} \text{ (Shape-Fun-Def)}$$

$$\frac{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \tau_1 \rhd \tau_1' \qquad \mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \tau_2 \rhd \tau_2'}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \tau_1 \to \tau_2 \rhd \tau_1' \to \tau_2'} \text{ (Shape-Fun-Type)}$$

$$\frac{X \mapsto \mathcal{I}^{\ell}(\overline{I})\{\ldots;\textbf{type } T^{\ell};\ldots\} \in \Delta}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash X.T \rhd T^{\ell}} \text{ (Shape-ProjType)}$$

$$\frac{T^{\ell} \in \text{dom}(\Gamma)}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash T \rhd T^{\ell}} \text{ (Shape-Abs-Type)}$$

$$\frac{X \mapsto I \in \Delta}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash X \rhd I} \text{ (Shape-Var)}$$

$$\frac{}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \{\} \rhd \{\}} \text{ (Shape-Empty-Body)}$$

Fig. 3. Shaping of Packages

$$\frac{\mathcal{R};\Delta_0;\bullet \vdash E_1 \rhd I_1 \quad \ldots \quad \mathcal{R};\Delta_{n-1};\bullet \vdash E_n \rhd I_n \quad (\Delta_i \equiv [X_1 \mapsto I_1, \ldots, X_i \mapsto I_i]) \quad \mathcal{R};\Delta_n;\ell;\bullet \vdash \{body\} \rhd \{B\} \quad \ell \text{ fresh} \quad [\mathcal{R};\Delta_n;\ell;\bullet \vdash E \rhd \mathcal{I}^{\ell'}(\ldots)\{[\ell \mapsto \ell']B'\} \quad \{B\} <: \{B'\}]}{\mathcal{R};\bullet \vdash \textbf{iface } \mathcal{I}(X_1 : E_1, \ldots, X_n : E_n)\ [<: E]\ \{body\} \rhd \mathcal{I}^{\ell}(\overline{I_i})\{B\}} \text{ (Shape-Idef)}$$

$$\frac{T^{\ell} \notin \mathrm{dom}(\Gamma) \quad \mathcal{R};\Delta;\ell;\Gamma[T^{\ell} \mapsto \textbf{abs}] \vdash \{body\} \rhd \{B\}}{\mathcal{R};\Delta;\ell;\Gamma \vdash \{\textbf{type } T; body\} \rhd \{\textbf{type } T^{\ell}; B\}} \text{ (Shape-Type-Decl)}$$

$$\frac{f \notin \mathrm{dom}(\Gamma) \quad \mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \tau \rhd \tau' \quad \mathcal{R};\Delta;\ell;\Gamma[f : \tau'] \vdash \{body\} \rhd \{B\}}{\mathcal{R};\Delta;\mathcal{L};\Gamma \vdash \{\textbf{fun } f : \tau; body\} \rhd \{\textbf{fun } f : \tau'; B\}} \text{ (Shape-Fun-Decl)}$$

Fig. 4. Shaping of Interfaces

$$\frac{}{\Gamma \vdash \mathcal{I}^{\ell}(\overline{I})\{\ldots \textbf{fun } f : \tau; \ldots\}.f : \tau} \text{ (Type-ProjTerm)}$$

$$\frac{T^{\ell} \mapsto \tau \in \Gamma \quad \Gamma \vdash e : T^{\ell}}{\Gamma \vdash e : \tau} \text{ (Type-Reify)}$$

$$\frac{T^{\ell} \mapsto \tau \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash e : T^{\ell}} \text{ (Type-Abstract)}$$

$$\frac{x \mapsto \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ (Type-CoreVar)}$$

Fig. 5. Core-level Typing (omitting other term typing rules)

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

$$\frac{}{\textbf{type } T^{\ell} = \tau <: \textbf{type } T^{\ell}} \text{ (Sub-Type-Def-Decl)}$$

$$\frac{}{\textbf{fun } f : \tau = \mathrm{e} <: \textbf{fun } f : \tau} \text{ (Sub-Fun-Decl-Decl)}$$

$$\frac{}{S <: S} \text{ (Sub-Refl)}$$

Fig. 6. Sub-Shaping Rules for Bodies.

$$\frac{\begin{array}{c}\mathcal{R} \vdash \mathcal{I}_2 \overset{M}{\to} \mathcal{I}_1 : M \\ \mathcal{R}; \bullet \vdash I'_1 <: I_1 \dashv \mathcal{E}_1 \qquad \ldots \qquad \mathcal{R}; \mathcal{E}_{n-1} \vdash I'_n <: I_n \dashv \mathcal{E}_n \\ \overline{\mathcal{E}_n(M(I)) \equiv I'}\end{array}}{\mathcal{R}; \mathcal{E}_0 \vdash \mathcal{P}^{\ell_1}(I_1, \ldots, I_n)\{B_1\} : \mathcal{I}_1^{\ell_1}(\overline{I})\{B'_1\} <: \mathcal{P}^{\ell_2}(I'_1, \ldots, I'_n)\{B_2\} : \mathcal{I}_2^{\ell_2}(\overline{I'})\{B'_2\} \dashv \mathcal{E}'} \text{ (SubShape-Pkg)}$$

$$\frac{\begin{array}{c}\mathcal{R} \vdash \mathcal{I}_2 \overset{M}{\to} \mathcal{I}_1 : M \\ \mathcal{R}; \bullet \vdash I_{M(1)} <: I'_1 \dashv \mathcal{E}_1 \qquad \ldots \qquad \mathcal{R}; \mathcal{E}_{n-1} \vdash I_{M(n)} <: I'_n \dashv \mathcal{E}_n \\ \ell_2 \notin \mathrm{dom}(\mathcal{E}_0) \qquad \mathcal{E}_{out} = \mathcal{E}_m[\ell_2 \mapsto \mathcal{I}_1^{\ell_1}(I_1, \ldots, I_n)\{B_1\}]\end{array}}{\mathcal{R}; \mathcal{E}_0 \vdash \mathcal{I}_1^{\ell_1}(I_1, \ldots, I_n)\{B_1\} <: \mathcal{I}_2^{\ell_2}(I'_1, \ldots, I'_m)\{B_2\} \dashv \mathcal{E}_{out}} \text{ (SubShape-Iface-1)}$$

$$\frac{\ell_2 \mapsto \mathcal{I}_1^{\ell_1}(\overline{I})\{B_1\} \in \mathcal{E}}{\mathcal{R}; \mathcal{E} \vdash \mathcal{I}_1^{\ell_1}(\overline{I})\{B_1\} <: \mathcal{I}_2^{\ell_2}(\overline{I'})\{B_2\} \dashv \mathcal{E}} \text{ (SubShape-Iface-2)}$$

Fig. 7. Sub-Shaping Rules that Handle Sharing.

$$\frac{\mathcal{I}_2 \mapsto \textbf{iface } \mathcal{I}_2(p_1 : E_1, \ldots, p_n : E_n) <: \mathcal{I}_1(p'_1, \ldots, p'_m)\{\ldots\} \in \mathcal{R}}{\mathcal{R} \vdash \mathcal{I}_1 \overset{M}{\to} \mathcal{I}_2 : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}} \text{ (Map-SubShape)}$$

$$\frac{\mathcal{I} \mapsto \textbf{iface } \mathcal{I}(p_1 : E_1, \ldots, p_n : E_n)\{\ldots\} \in \mathcal{R}}{\mathcal{R} \vdash \mathcal{I} \overset{M}{\to} \mathcal{I} : [1 \mapsto 1, \ldots, n \mapsto n]} \text{ (Map-Refl)}$$

$$\frac{\mathcal{R} \vdash \mathcal{I}_1 \overset{M}{\to} \mathcal{I}_2 : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \qquad \mathcal{R} \vdash \mathcal{I}_2 \overset{M}{\to} \mathcal{I}_3 : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}}{\mathcal{R} \vdash \mathcal{I}_1 \overset{M}{\to} \mathcal{I}_3 : \{1, \ldots, m\} \mapsto \{1, \ldots, k\}} \text{ (Map-Trans)}$$

Fig. 8. Formal parameter mappings

$$\frac{\mathcal{R};\Phi;\mathcal{P} \vdash \{body\} \longrightarrow \{\ldots; \textbf{fun } main : Unit \to Int = \lambda() : Unit.e\}}{\mathcal{R};\Phi;\bullet \vdash \textbf{pkg } \mathcal{P}\ \{body\} \longrightarrow e} \text{ (E-Prog)}$$

$$\frac{\begin{array}{c}(\mathcal{P}, X, \mathcal{P}') \overset{import}{\rightsquigarrow} S \in \Phi \qquad \mathcal{P}' \overset{impl}{\rightsquigarrow} S' \in \Phi \qquad \mathcal{R};\bullet \vdash S' <: S \dashv \mathcal{E} \\ \mathcal{P}' \mapsto \textbf{pkg } \mathcal{P}'(\overline{par : E}) \textbf{ impl } E'\ \{body'\} \in \mathcal{R} \\ \mathcal{R};\Phi;\mathcal{P}' \vdash [\overline{par \mapsto arg}]\{body'\} \longrightarrow v\end{array}}{\mathcal{R};\Phi;\mathcal{P} \vdash \{\textbf{import } X : \ldots \textbf{ from } \mathcal{P}'(\overline{arg}); body\} \longrightarrow [X \mapsto v]\{body\}} \text{ (E-Import)}$$

$$\frac{\mathcal{R};\Phi;\mathcal{P} \vdash \tau \longrightarrow \tau' \qquad \mathcal{R};\Phi;\mathcal{P} \vdash \{body\} \longrightarrow \{body'\}}{\mathcal{R};\Phi;\mathcal{P} \vdash \{\textbf{type } T = \tau; body\} \longrightarrow \{\textbf{type } T = \tau'; body'\}} \text{ (E-TypeDef)}$$

$$\frac{\mathcal{R};\Phi;\mathcal{P} \vdash \tau \longrightarrow \tau' \qquad \mathcal{R};\Phi;\mathcal{P} \vdash [f \mapsto v]\{body\} \longrightarrow \{body'\}}{\mathcal{R};\Phi;\mathcal{P} \vdash \{\textbf{fun } f : \tau = v; body\} \longrightarrow \{\textbf{fun } f : \tau' = v; body'\}} \text{ (E-FunDef)}$$

$$\frac{}{\mathcal{R};\Phi;\mathcal{P} \vdash \{\ldots; \textbf{type } T^\ell = \tau; \ldots\}.T \longrightarrow T^\ell} \text{ (E-Proj-Type)}$$

$$\frac{}{\mathcal{R};\Phi;\mathcal{P} \vdash \{\ldots; \textbf{fun } f : \tau = v; \ldots\}.f \longrightarrow v} \text{ (E-Proj-Fun)}$$

$$\frac{\mathcal{R};\Phi;\mathcal{P} \vdash \tau_1 \longrightarrow \tau_1' \qquad \mathcal{R};\Phi;\mathcal{P} \vdash \tau_2 \longrightarrow \tau_2'}{\mathcal{R};\Phi;\mathcal{P} \vdash \tau_1 \to \tau_2 \longrightarrow \tau_1' \to \tau_2'} \text{ (E-Arrow-Type)}$$

$$\frac{}{\mathcal{R};\Phi;\mathcal{P} \vdash T^\ell \longrightarrow T^\ell} \text{ (E-Abs-Type)}$$

Fig. 9. Operational Semantics for $\prod$.