Towards a Theory of Packages:
Technical Appendix

Mark Florisson and Alan Mycroft (June 17, 2015)
Computer Laboratory, University of Cambridge
JJ Thomson Avenue, Cambridge CB3 0FD, UK
Firstname.Lastname@cl.cam.ac.uk

A Definitions and Judgements

This section covers the different kind of formalism and judgement forms we make use in our static in dynamic semantics. We do this to give the reader a better understanding of how things fits together.

A.1 Shapes and Substitutions

Shapes and their structure have been covered extensively in §5.1. We repeat the grammar of shapes here:

\[ I ::= \mathcal{I}(\text{type } T, \text{fun } f : \tau) \quad \text{(interface shape)} \]
\[ P ::= \mathcal{P}(\text{type } T = \tau; \text{fun } f : \tau = e) : I \mid \mathcal{P}(\ast) : I \quad \text{(package shape)} \]

We remind the reader that shapes have more or less the same structure as their respective interface or package definitions. However, shapes include labels and bodies, and no longer contain variables \( X \). There are two kinds of substitution that we define on shapes: the first is relabelling, written \( \ell \mapsto \ell' \) and ranged over by \( \phi \) that rewrites labels \( \ell \) to labels \( \ell' \):

\[ \phi(\mathcal{P}(\ell)\{B\}) = \mathcal{P}(\phi(\ell))\{\phi(B)\} \]
\[ \phi(\mathcal{I}(\ell)\{B\}) = \mathcal{I}(\phi(\ell))\{\phi(B)\} \]
\[ \phi(\text{type } T^\ell [= \tau]) = \text{type } T^{\phi(\ell)} [= \phi(\tau)] \]
\[ \phi(\text{fun } f : \tau [= e]) = \text{fun } f : \phi(\tau) [= \phi(e)] \]
\[ \ldots, \ell_1 \mapsto \ell_2, \ldots)(\ell_1) = \ell_2 \]

The second kind of substitution is written \( \ell \mapsto I \) this replaces entire \( \ell \)-labelled shape sub-terms with \( I \) whenever \( \ell \mapsto I \in \sigma \). It also performs relabelling of types that were projected by the (\text{SHAPE-TYPEPROJ}) shaping rule.

\[ \sigma(\mathcal{P}(\sigma(\ell))\{B\}) = \mathcal{P}(\sigma(\ell))(\sigma(B)) \]
\[ \sigma(\mathcal{I}(\sigma(\ell))\{B\}) = \begin{cases} I \quad \text{if } \ell \mapsto I \in \sigma \\ \mathcal{I}(\sigma(\ell))(\sigma(B)) \quad \text{otherwise} \end{cases} \]
\[ \sigma(\text{type } T^\ell [= \tau]) = \text{type } T^{\sigma(\ell)} [= \sigma(\tau)] \]
\[ \sigma(\text{fun } f : \tau [= e]) = \text{fun } f : \sigma(\tau) [= \sigma(e)] \]
\[ \ldots, \ell_1 \mapsto \mathcal{I}(\ell_1)(B), \ldots)(\ell_1) = \ell_2 \]
A.2 Contexts

In our judgements we use the contexts of the forms listed in Figure 1. A repository $\mathcal{R}$ contains all package and interface definitions, and is used in most judgement forms (§A.3). Repositories are often used to retrieve and shape interface definitions. Next we use $\Delta$ in our static semantics (§B.1) to map variables $X$ to shapes (package variables $Pvars$ and interface parameters $Ipars$). Context $\Gamma$ is used for core language judgements, and maps core-language variables $x$ (including functions $f$) to types $\tau$ and keeps track of abstract type definitions $T^\ell$. $\mathcal{E}$ is a sharing context, that keeps track of the correspondence between labels of interface shapes in definitions and interface shapes in a use (package or interface application, i.e. $Pimport$ or $Iexpr$). We use $\Phi$ in our dynamic semantics to resolve $import$ statements (§B.2). Finally we use $\Theta$ to support dependent compatibility for increased flexibility in the compatibility relation (§C, which is entirely optional).

 Repositories $\mathcal{R} ::= PVname \mapsto Pdef \mid TVname \mapsto Idef$
 Package context $\Delta ::= X \mapsto I$
 Core-language context $\Gamma ::= x \mapsto \tau \mid T \mapsto (\tau \mid \text{abs})$
 Label to shape mapping $\mathcal{E} ::= \ell \mapsto I$
 Dependency mapping $\Phi ::= \mathcal{P} \triangleright \triangleright S \mid (P_1, X, P_2) \triangleright \triangleright S$
 Concrete package context $\Theta ::= X \mapsto I$

Fig. 1. Contexts

A.3 Judgements

Judgement forms, along with the sections that use them, are listed in Figure 2. First there are judgements to shape $import$ and $impl$ clauses in package definitions. These judgements compute a package shape $P^\ell(...)\{B\} : I^\ell(...)\{B\}$ that is used for dependence matching, using sub-shaping judgements $\mathcal{R};\mathcal{E} \vdash S_1 <: S_2 \vdash E'$ to match shapes from definitions with shapes from imports. This judgement form relies on a mapping $I_1 \Rightarrow I_2$ between argument positions of interface constructors which are nominal subtypes.

Next are shaping judgements $\mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash E \triangleright S$, which uses sub-shaping of bodies $B_1 <: B_2$ to check for validity of the $impl$ and $<$; clauses, and $\mathcal{R} \vdash I \text{ wf}$ to check whether interface constructors are properly applied (Iexpr). Finally we use typing judgements $\Delta; \Gamma \vdash t : \tau$ to ensure well-typedness of programs. In our dynamic semantics we use a single-step reduction relation $\mathcal{R}; \Phi; \mathcal{P} \vdash e \rightarrow v$ which represents evaluation of programs, packages and core-language terms. Finally, we use judgements $\mathcal{R}; \Delta; \mathcal{I}; \Gamma; \Phi; \Theta \vdash E \triangleright S$ to resolve dependencies more flexibly by dissolving import sealing boundaries in a disciplined way.
A.4 Equivalence

Equivalence checking for shapes and types is needed during shape- and type-checking. Equivalence of labels, types and shapes is defined syntactically and we omit the (trivial) rules.

B Semantics

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

B.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-Idf), does for interfaces what (Shape-Pdef) does for packages. Scoping of parameters is handled in the same way, and $\Delta_i$ 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 $\tau$ and $e$ respectively. Rules for shaping interface applications or variables are identical to the package shaping rules.

---

Import Shape 5.2 $\mathcal{R}; \Delta; \Lambda; \Gamma \vdash E \xrightarrow{\text{import}} I$

Impl Shape 5.2 $\mathcal{R}; \Delta; \Lambda; \Gamma \vdash E \xrightarrow{\text{impl}} I$

Shaping 5.3, B.1 $\mathcal{R}; \Delta; \Lambda; \Gamma \vdash E \triangleright S$

Typing B.1 $\Gamma \vdash t : \tau$

Interface Parameter Permutation B.1 $\mathcal{R} \vdash \mathcal{I}_1 \xrightarrow{\mathcal{M}} \mathcal{I}_2 : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$

Shape Matching B.1 $\mathcal{R}; \mathcal{E} \vdash S_1 <: S_2 \triangleright \mathcal{E}'$

Sub-Shaping B.1 $B_1 <: B_2$

Well-Formedness of Shapes B.1 $\mathcal{R} \vdash I \text{ wf}$

One-Step Reduction Relation B.2 $\mathcal{R}; \Phi; \mathcal{P} \vdash e \rightarrow v$

Static Elaboration of Dependencies C $\mathcal{R}; \Delta; \Lambda; \Gamma; \Phi; \Theta \vdash E \triangleright S$

Fig. 2. Judgements
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 $T = \tau$ 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 Iexprs against the formal parameters computed from Idefs. 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 $E$. Well-formedness of body $\{B\}$ can be ignored, because this is checked during shaping of the interface definition.

$$ \begin{array}{c}
\frac{\mathcal{R} \vdash \text{iface } I(E) [\ldots] \triangleright I^\ell(\bar{I}_i) \{\bar{B}\}}{\mathcal{R} ; \bullet \vdash I_i <: I_i^\ell + E_i \quad \ldots \quad \mathcal{R} ; E_{n-1} \vdash I_i <: I_i^\ell + E_n \quad 1 \leq i \leq n}
\end{array} $$

(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 ($\S 5.2$), 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 $E$. 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 $E$ through the rules. Rule (SubSHAPE-PKG) handles sub-shaping of package shapes, used to match im-
ports 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 contravari-
ant in their arguments we first check that \( R; E \vdash I'_1 <: I_i \vdash E_i \) using sharing
context \( E_i = [\text{lab}(I_1) \mapsto I'_1, \ldots, \text{lab}(I_n) \mapsto I'_n] \). Package shapes are covariant
in the exported interface, so we need to check that \( I'_1 <: I_i \vdash E_i \). We
do this by computing a correspondence \( M \) between the arguments to \( I_1 \) and \( I_2 \).
We then use substitution with \( E_n \) (which is valid because we know that \( I'_i <: I_i \))
to check for an exact sharing correspondence between the two interfaces. Substi-
tution is necessary to handle the dependence that exists between the parameters
to \( P_{\ell_1} \) and the arguments to \( I_1 \). Without substitution subtyping would not work,
as the arguments are contravariant and the result covariant. For example, the
relation would not hold for \( R; E \vdash P_{\ell_1}(T) <: P_{\ell_1'}(S) \vdash E' \) whenever
\( R; E \vdash S <: T \vdash E' \) with \( S \neq T \). However, we want the relation to
hold, because the shape of the imported package (left) is really \( I'_1(S) \), which is
trivially a sub-shape of \( I_i(S) \) (the required shape on the right).

Next are rules (SubShape-Iface-1) and (SubShape-Iface-2), which update \( E \)
whenever they see a new label \( \ell \); 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
\( I'_2(T)\{B'_1\} \), 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 inter-
face 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.
\( I_2(T_2)\{B_2\} \) \( I_1(T_1)\{B_1\} \)). To define sub-shaping, we have to understand how
the parameters of \( I_2(T_2) \) correspond to those of \( I_1(T_1) \). For example, consider
the following interface definitions:

```plaintext
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 \( \mathcal{M} : I_2 \rightarrow I_1 \) 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:

\[ [1 \mapsto x_1, \ldots, m \mapsto x_m]T_1 = T_2 \]

The rules for computing these functions are listed in Figure 8.
B.2 Dynamic Semantics

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

```
pkg PVname {Pbody}
```

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

```
fun main : Unit → Int = λ().e
```

Package values are (closed) package bodies of the form

```
PkgVal ::= {type T^ℓ = τ_val; fun f : τ_val = v}
```

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

```
R; Φ; P ⊢ e → v
```

where `R` is a well-formed repository, `Φ` is a mapping to connect internal dependencies with package implementations (covered next), and `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 `⇝` 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:

```
Φ ::= P impl ⇝ S | (P_1, X, P_2) import ⇝ S
```

In particular, rule (E-Import) uses `Φ` to look up an appropriate package implementation for an `import` statement. Here `P` and `P_1` are versioned `PVname` package constructors. However, `P_2` is an unversioned `Pname`, leaving the evaluation rules to determine an appropriate version. Here `P_1` is the dependant (the importing package, for which a concrete version number is known), and `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 `P` to range over package constructors that are appropriate versioned or unversioned depending on context. Rules for core-language term reduction `e → 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 $R$ is a well-formed repository, and $\Phi$ is a shape mapping for all package definitions (\texttt{impl}) and internal dependencies (\texttt{import}). Theorem 1 states progress of package evaluation, and Theorem 2 states preservation of shape of package terms under evaluation.

**Theorem 1 (Package Progress).** Let $p$ be a label-annotated package body that is well-shaped with respect to repository $R$. Then either $p$ is a value, or there is some $p'$ such that $R; \Phi; P \vdash p \rightarrow p'$.

**Theorem 2 (Package Preservation).** Let $p$ be a label-annotated package body with $R; \bullet \vdash p \triangleright B$ and $R; \Phi; P \vdash p \rightarrow p'$, then also $R; \bullet \vdash p' \triangleright B$.

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

C  Expressivity and Dependent Compatibility

In §6 of the paper we alluded to a form of dependent compatibility, where compatibility of one dependency is dependent on the implemented interface of another (rather than the required interface). Consider the following package definitions:

```
pkg Wombat-v1  impl IWombat-v1
   ...
pkg Park-v1 (W : IWombat-v1)  impl IPark-v1 (W)
   ...
pkg WombatPark-v1  impl IWombatPark-v1
    import W : IWombat-v1 from Wombat-any
    import P : IPark-v1 (W) from Park-any
   ...
```

Here we see that $\text{WombatPark-v1}$ has an internal dependence on $\text{Wombat}$ and $\text{Park}$. We can further see that $\text{Park-v1}$ has an external dependence on a package implementing $\text{IWombat-v1}$. Now consider an upgrade to each of the packages and respective interfaces:

```
pkg Wombat-v2  impl IWombat-v2
   ...
pkg Park-v2 (W : IWombat-v2)  impl IPark-v2 (W)
   ...
pkg WombatPark-v2  impl IWombatPark-v2
    import W : IWombat-v2 from Wombat-any
    import P : IPark-v2 (W) from Park-any
   ...
```

where the newer interfaces are backwards compatible with the older ones:
iface IWombat-v2 <: IWombat-v1
.

iface IPark-v2 (W : IWombat-v2) <: IPark-v1 (W)
.

Here we see that again WombatPark-v2 has an internal dependence on Wombat and Park. As before, we can further see that Park-v2 has an external dependence on a package implementing IWombat-v2. Here we see that the older version WombatPark-v1 is compatible with Wombat-v2, since Wombat-v2 is backwards compatible with Wombat-v1. However, WombatPark-v1 is not compatible with Park-v2, as Park-v2 is not backwards compatible with Park-v1. This rejected compatibility is good, since WombatPark-v1 may otherwise provide Park-v2 with something that implements IWombat-v1, whereas it requires something of IWombat-v2. However, as we had just seen, WombatPark-v1 can in fact sometimes provide something of IWombat-v2, and so should be compatible with Park-v2 under the condition that its import of Wombat implements something compatible with IWombat-v2.

We call this kind of compatibility dependent compatibility. We can support such compatibility if we track the implemented rather than the required interface while resolving dependencies. This essentially breaks the import sealing boundary for the purpose of dependency resolution (but never for the purpose of shaping the package itself!).

To account for dependent compatibility in our rules, we can add an additional finite partial map $\Theta$ which we call a concrete package context from variables $X$ to their actual interface shapes:

$$\Theta ::= \frac{}{X \mapsto I}$$

Then, when an import is resolved we can use use this new mapping as follows:

$$\frac{}{\mathcal{R}}; \Delta; L; \Gamma; \Theta \vdash \{\text{import } X : \mathcal{I}(Y) \text{ from } \mathcal{P}(\overline{Z}); \text{body} \} \triangleright B \quad \text{(SHAPE-PIMPORT)}}$$

With appropriate modifications to shaping rules $\triangleright$ and $\implies$ to use $\Theta$. Although the rule looks somewhat complicated, it is conceptually simple: $\Theta$ contains the actually implemented interfaces of imported packages, instead of the required interface. We need to do three things: first we need to propagate shape information from $\Theta$ into imported packages, second we need to use match dependencies using $\Theta$ and third we need to update $\Theta$ with the imported result.

First, when we import a package we need to find a suitable implementation. We now need to propagate information from $\Theta$ about the external dependencies
of the package:

\[ Z' \mapsto \Theta(Z) \]

Second, we also do matching using the new shape information:

\[ \mathcal{R} : \bullet \vdash P' \{ \mathcal{I}_2 \} \{ \ast \} : I_2 : P(\Theta(Z)) \{ \ast \} : I_1(\Theta(Y)) \vdash E \]

The right-hand side here is the importing shape, and the left-hand side the imported shape. By using \( \Theta \) we would for example make clear that we are applying package \( \text{Park} \) with a package implementing \( \text{IWombat-v2} \) instead of \( \text{IWombat-v1} \). We can now shape the remainder of the package body with the implemented interface from the import:

\[ \Delta \{ X \mapsto I_1 \}; \mathcal{L} ; \Gamma ; \Theta \{ X \mapsto I''_2 \} \vdash \{ \text{body} \} \triangleright B \]

Here \( \Delta \) contains the required interface, and \( \Theta \) the implemented one. We cannot just update \( \Delta \) with the implemented interface, as shaping and type-checking need to be independent of the actual packages used.

We have gained some flexibility in how packages may be composed, and now provide users with a notion of “maximal compatibility”, which is achieved by requiring as little as possible (import \( \text{Wombat} \) with \( \text{IWombat-v1} \)) and providing as much as possible (impl \( \text{IWombat-v2} \) where possible). Flexibility is useful because it enables greater sharing of packages, reducing the number of simultaneous package versions required on user systems. For example, even if we use an old version \( \text{WombatPark-v1} \), we can still use up-to-date versions of \( \text{Wombat} \) and \( \text{Park} \).

## D 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 \( P \mapsto P_{\text{def}} \in \mathcal{R} \) 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 (§B.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 \( \text{Foo} \) requires a \( \text{Bar} \) with fresh mutable state, and \( \text{Bar} \) requires a \( \text{Foo} \) 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 \( \mathcal{P}(T)^{\ell} : \mathcal{I}(T)^{\ell} \{ B \} \) for
all packages in \( R \), as defined by rule (Dep-Pimpl) (Figure 2 from §5.2). The
edges \( E \) are constructed from the package dependencies arising from imports,
as defined by rule (Dep-Import) 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 \( I(IVname) \), and there is an edge between two nodes
\( I_1 \) and \( I_2 \) whenever \( I_1 \) occurs in an \( Iexpr \) of \( I_2 \).

To handle restrictions on repositories that cover dependent dependencies
(see §C), we need to refine our subtype algorithm for package shapes to con-
sider the possibility that the argument shapes \( T \) in \( \mathcal{P}(T) : \mathcal{I}(T) \) from an import
statement, may actually be sub-shapes \( T' : I \). Hence we need to match
shapes generated by (Dep-Import) both co- and contravariantly against shapes
generated by (Dep-Pimpl).
\( \mathcal{R}; \Delta_0; \bullet \vdash E_1 \triangleright I_1 \quad \ldots \quad \mathcal{R}; \Delta_{n-1}; \bullet \vdash E_n \triangleright I_n \quad (\Delta_i \equiv [X_1 \mapsto I_1, \ldots, X_i \mapsto I_i]) \)

\( \mathcal{R}; \Delta_n; \ell; \bullet \vdash \{ \text{body} \} \triangleright \{ B \} \quad \ell \) fresh

\( \mathcal{R}; \Delta_n; \ell; \bullet \vdash E \triangleright \mathcal{T}(\ell)([\ell \mapsto \ell']B') \quad \{ B \} < \{ B' \} \)

\( \mathcal{R}; \bullet \vdash \text{pkg } \mathcal{P}(X_1 : E_1, \ldots, X_n : E_n) \implies E \{ \text{body} \} \triangleright \mathcal{P}(T_i)(B) : \mathcal{T}(T_i)(B') \) (SHAPE-PDEF)

\( \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash E_1 \triangleright I \)

\( \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash \{ \text{import } X : E_1 \text{ from } E_2; \ldots \} \triangleright \{ B \} \) (SHAPE-PIMPORT)

\( \ell \) fresh

\( \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash \{ \text{iface } I \ldots \triangleright \mathcal{T}(T_i)(B') \quad \mathcal{T}(T_i)\} \text{ wf} \) (SHAPE-IEXPR)

\( \mathcal{T}^\ell \not\in \text{dom}(\Gamma) \quad \mathcal{R}; \Delta; \ell; \Gamma \vdash \tau \triangleright \tau' \)

\( \mathcal{R}; \Delta; \ell; \Gamma \vdash \tau^s \downarrow \{ \text{body} \} \triangleright \{ B \} \) (SHAPE-TYPE-DEF)

\( f \not\in \text{dom}(\Gamma) \quad \mathcal{R}; \Delta; \ell; \Gamma \vdash \tau \triangleright \tau' \quad \mathcal{R}; \Delta; \ell; \Gamma \vdash e \triangleright e' \)

\( \mathcal{R}; \Delta; \ell; \Gamma \vdash \{ \text{fun } f : \tau = e, \text{body} \} \triangleright \{ \text{fun } f : \tau' = e; B \} \) (SHAPE-FUN-DEF)

\( \mathcal{R}; \Delta; \ell; \Gamma \vdash \tau_1 \triangleright \tau'_1 \quad \mathcal{R}; \Delta; \ell; \Gamma \vdash \tau_2 \triangleright \tau'_2 \)

\( \mathcal{R}; \Delta; \ell; \Gamma \vdash \tau_1 \triangleright \tau_2 \triangleright \tau'_1 \rightarrow \tau'_2 \) (SHAPE-FUN-TYPE)

\( X \mapsto \mathcal{T}(T_i)[\ldots; \text{type } \tau^s; \ldots] \in \Delta \)

\( \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash X, T \triangleright T^s \) (SHAPE-PROJTYPE)

\( T^s \in \text{dom}(\Gamma) \quad \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash T \triangleright T^s \) (SHAPE-ABS-TYPE)

\( X \mapsto I \in \Delta \quad \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash X \triangleright I \) (SHAPE-VAR)

\( \mathcal{R}; \Delta; \mathcal{L}; \Gamma \vdash \{ \} \triangleright \{ \} \) (SHAPE-EMPTY-BODY)

Fig. 3. Shaping of Packages
\[
R; \Delta_0; \vdash E_1 \triangleright I_1 \quad \ldots \quad R; \Delta_{n-1}; \vdash E_n \triangleright I_n \quad (\Delta_i \equiv [X_1 \mapsto I_1, \ldots, X_i \mapsto I_i])
\]
\[
R; \Delta_n; \ell; \vdash \{\text{body}\} \triangleright \{B\} \quad \ell \text{ fresh}
\]
\[
[r; \Delta_n; \ell; \vdash E \triangleright T^\ell(\ldots)([\ell \mapsto \ell']B') \quad \{B\} \ll \{B'\}]
\]
\[
R; \vdash \text{iface } I(X_1 : E_1, \ldots, X_n : E_n) \ll [E] \{\text{body}\} \triangleright T^\ell(I)(B)
\]

(Fig. 4. Shaping of Interfaces)

\[
T^\ell \not\in \text{dom}(\Gamma) \quad R; \Delta; \ell; \Gamma[T^\ell \mapsto \text{abs}] \vdash \{\text{body}\} \triangleright \{B\}
\]

\[
R; \Delta; \ell; \Gamma \triangleright \{\text{type } T; \text{body}\} \triangleright \{\text{type } T^\ell; B\}
\]

(Shape-Type-Decl)

\[
f \not\in \text{dom}(\Gamma) \quad R; \Delta; \ell; \Gamma \vdash \{\text{fun } f : \tau; \text{body}\} \triangleright \{\text{fun } f : \tau'; B\}
\]

(Shape-Fun-Decl)

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

\[
\Gamma \vdash T^\ell(\ldots\text{fun } f : \tau; \ldots)f : \tau
\]

(TYPE-PROJTERM)

\[
T^\ell \mapsto \tau \in \Gamma \quad \Gamma \vdash e : T^\ell
\]

\[
\Gamma \vdash e : \tau
\]

(TYPE-REFIFY)

\[
T^\ell \mapsto \tau \in \Gamma \quad \Gamma \vdash e : \tau
\]

\[
\Gamma \vdash e : T^\ell
\]

(TYPE-ABSTRACT)

\[
\Gamma \vdash x : \tau
\]

\[
\Gamma \vdash x : \tau
\]

(TYPE-COREVAR)

Fig. 6. Sub-Shaping Rules for Bodies.
\[ \mathcal{R} \vdash I_2 \xrightarrow{\text{iface}} I_1 : M \]

\[ \mathcal{R}; \bullet \vdash I'_1 <: I_1 \vdash \mathcal{E}_1, \ldots, \mathcal{R}; \mathcal{E}_{n-1} \vdash I'_n <: I_n \vdash \mathcal{E}_n \]

\[ \mathcal{E}_n(M(I)) \equiv \top \]

\[ \mathcal{R}; \mathcal{E}_0 \vdash P_{i_1}(I_1, \ldots, I_n)(B_1) : I_1^{i_1}(\overline{I})\{B_1\} \iff P_{i_2}(I_1, \ldots, I_n)(B_2) : I_2^{i_2}(\overline{I})\{B_2\} \dashv \mathcal{E}_n \]

(SubShape-Pkg)

\[ \mathcal{R}; \bullet \vdash I_{M(1)} <: I'_1 \vdash \mathcal{E}_1, \ldots, \mathcal{R}; \mathcal{E}_{n-1} <: I'_n \vdash \mathcal{E}_n \]

\[ \ell_2 \notin \text{dom}(\mathcal{E}_0) \]

\[ \mathcal{E}_{\text{out}} = \mathcal{E}_n[\ell_2 \mapsto I_1^{i_1}(I_1, \ldots, I_n)(B_1)] \]

\[ \mathcal{R}; \mathcal{E}_0 \vdash I_1^{i_1}(I_1, \ldots, I_n)(B_1) <: I_2^{i_2}(I_1, \ldots, I_n)(B_2) \dashv \mathcal{E}_{\text{out}} \]

(SubShape-Iface-1)

\[ \ell_2 \mapsto I_1^{i_1}(\overline{I})\{B_1\} \in \mathcal{E} \]

(SubShape-Iface-2)

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

\[ I_2 \mapsto \text{iface} \ I_2(p_1 : E_1, \ldots, p_n : E_n) <: I_1(p'_1, \ldots, p'_m)\{\ldots\} \in \mathcal{R} \]

(Map-SubShape)

\[ \mathcal{R} \vdash I_1 \xrightarrow{\text{iface}} I_2 : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \]

(Map-Refl)

\[ \mathcal{R} \vdash I_1 \xrightarrow{\text{iface}} I : [1 \mapsto 1, \ldots, n \mapsto n] \]

\[ \mathcal{R} \vdash I_1 \xrightarrow{\text{iface}} I_2 : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \mapsto \{1, \ldots, k\} \]

(Map-Trans)

Fig. 8. Formal parameter mappings
$$
\begin{align*}
R; \Phi; \mathcal{P} \vdash \{\text{body}\} & \longrightarrow \{\ldots; \text{fun main : Unit} \to \text{Int} = \lambda() : \text{Unit.e}\} \\
R; \Phi; \bullet \vdash \text{pkg} \mathcal{P} \{\text{body}\} & \longrightarrow e
\end{align*}
$$

(E-PROG)

$$
(P, X, \mathcal{P}') \vdash \text{import } S \in \Phi \quad \mathcal{P}' \vdash \text{impl } S' \in \Phi \quad R; \bullet \vdash S' <: S \vdash \mathcal{E} \\
R; \Phi; \mathcal{P}' \vdash \text{pkg} \mathcal{P}'(\text{par : E}) \text{ impl } E' \{\text{body}'\} \in R \\
R; \Phi; \mathcal{P}' \vdash \text{par } \vdash \text{my}\{\text{body}'\} \longrightarrow v
$$

(E-IMPORT)

$$
R; \Phi; \mathcal{P} \vdash \tau \longrightarrow \tau' \quad R; \Phi; \mathcal{P} \vdash \{\text{body}'\} \longrightarrow \{\text{body}'\} \\
R; \Phi; \mathcal{P} \vdash \{\text{type } T = \tau; \text{body}\} \longrightarrow \{\text{type } T = \tau'; \text{body}'\}
$$

(E-TYPEDEF)

$$
R; \Phi; \mathcal{P} \vdash \tau \longrightarrow \tau' \quad R; \Phi; \mathcal{P} \vdash \{f \mapsto v\}\{\text{body}\} \longrightarrow \{\text{body}'\} \\
R; \Phi; \mathcal{P} \vdash \{\text{fun } f : \tau = v; \text{body}\} \longrightarrow \{\text{fun } f : \tau' = v; \text{body}'\}
$$

(E-FUNDEF)

$$
R; \Phi; \mathcal{P} \vdash \{\ldots; \text{type } T^l = \tau; \ldots\}.T \longrightarrow T^l
$$

(E-PROJ-TYPE)

$$
R; \Phi; \mathcal{P} \vdash \{\ldots; \text{fun } f : \tau = v; \ldots\}.f \longrightarrow v
$$

(E-PROJ-FUN)

$$
R; \Phi; \mathcal{P} \vdash \tau_1 \longrightarrow \tau'_1 \quad R; \Phi; \mathcal{P} \vdash \tau_2 \longrightarrow \tau'_2 \\
R; \Phi; \mathcal{P} \vdash \tau_1 \rightarrow \tau_2 \longrightarrow \tau'_1 \rightarrow \tau'_2
$$

(E-ARROW-TYPE)

$$
R; \Phi; \mathcal{P} \vdash T^l \longrightarrow T^l
$$

(E-ABS-TYPE)

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

14