The Backpack algorithm

May 8, 2015

This document describes the Backpack shaping and typechecking passes, as we intend to implement it.

1 Changelog

April 28, 2015 A signature declaration no longer provides a signature in the technical shaping sense; the motivation for this change is explained in **In-scope signatures are not provisions**. The simplest consequence of this is that all requirements are importable (Derek has stated that he doesn't think this will be too much of a problem in practice); it is also possible to extend shape with a `signatures` field, although some work has to be done specifying coherence conditions between `signatures` and `requirements`.

2 Front-end syntax

p, q, r		Package names	
m, n		Module names	
Packages			
pkg	::=	`package` p [$provreq$] `where {`d_1`;` ... `;`d_n`}`	
Declarations			
d	::=	`module` m [$exports$] `where` $body$	
	|	`signature` m [$exports$] `where` $body$	
	|	`include` p [$provreq$]	
Provides/requires specification			
$provreq$	::=	`(` rns `)` [`requires(` rns `)`]	
rns	::=	rn_0 `,` ... `,` rn_n[`,`]	Renamings
rn	::=	m `as` n	Renaming
Haskell code			
$exports$		A Haskell module export list	
$body$		A Haskell module body	

Figure 1: Syntax of Backpack

The syntax of Backpack is given in Figure 1. See the "Backpack manual" for more explanation about the syntax. It is slightly simplified here by removing any constructs which are easily implemented as syntactic sugar (e.g., a bare m in a renaming is simply m `as` m.)

3 Shaping

$$
\begin{array}{rcll}
\mathit{Shape} & ::= & \texttt{provides:}\ m\ \texttt{->}\ \mathit{Module}\ \{\,\mathit{Name}\texttt{,}\ \ldots\,\}\texttt{;}\ \ldots \\
 & & \texttt{requires:}\ m\ \texttt{->}\ \qquad\qquad \{\,\mathit{Name}\texttt{,}\ \ldots\,\}\texttt{;}\ \ldots \\
\mathit{PkgKey} & ::= & p\texttt{(}\,m\ \texttt{->}\ \mathit{Module}\texttt{,}\ \ldots\,\texttt{)} \\
\mathit{Module} & ::= & \mathit{PkgKey}\texttt{:}m \\
\mathit{Name} & ::= & \mathit{Module}\texttt{.}\mathit{OccName} \\
\mathit{OccName} & & \text{Unqualified name in a namespace}
\end{array}
$$

Figure 2: Semantic entities in Backpack

Shaping computes a *Shape*, whose form is described in Figure 2. Initializing the shape context to the empty shape, we incrementally build the context as follows:

1. Calculate the shape of a declaration, with respect to the current shape context. (e.g., by renaming a module/signature, or using the shape from an included package.)
2. Merge this shape into the shape context.

The final shape context is the shape of the package as a whole. Optionally, we can also compute the renamed syntax trees of modules and signatures.

In the description below, we'll assume `THIS` is the package key of the package being processed.

3.1 `module M`

A module declaration provides a module `THIS:M` at module name `M`. It has the shape:

```
provides: M -> THIS:M { exports of renamed M }
requires: (nothing)
```

Example:

```
module A(T) where
    data T = T

-- provides: A -> THIS:A { THIS:A.T }
-- requires: (nothing)
```

`OccName` is implied by `Name`. In Haskell, the following is not valid syntax:

```
import A (foobar as baz)
```

In particular, a `Name` which is in scope will always have the same `OccName` (even if it may be qualified.) You might imagine relaxing this restriction so that declarations can be used under different `OccName`s; in such a world, we need a different definition of shape:

```
Shape ::=
    provided: ModName -> { OccName -> Name }
    required: ModName -> { OccName -> Name }
```

Presently, however, such an `OccName` annotation would be redundant: it can be inferred from the `Name`.

Holes of a package are a mapping, not a set. Why can't the `PkgKey` just record a set of `Module`s, e.g. `PkgKey ::= SrcPkgKey { Module }`? Consider:

```
package p (A) requires (H1, H2) where
    signature H1(T) where
        data T
    signature H2(T) where
        data T
    module A(A(..)) where
        import qualified H1
        import qualified H2
        data A = A H1.T H2.T

package q (A12, A21) where
    module I1(T) where
        data T = T Int
    module I2(T) where
        data T = T Bool
    include p (A as A12) requires (H1 as I1, H2 as I2)
    include p (A as A21) requires (H1 as I2, H2 as I1)
```

With a mapping, the first instance of `p` has key `p(H1 -> q():I1, H2 -> q():I2)` while the second instance has key `p(H1 -> q():I2, H2 -> q():I1)`; with a set, both would have the key `p(q():I1, q():I2)`.

Signatures can require a specific entity. With requirements like `A -> { HOLE:A.T, HOLE:A.foo }`, why not specify it as `A -> { T, foo }`, e.g., `required: { ModName -> { OccName } }`? Consider:

```
package p () requires (A, B) where
    signature A(T) where
        data T
    signature B(T) where
        import T
```

The requirements of this package specify that `A.T` = `B.T`; this can be expressed with `Name`s as

```
A -> { HOLE:A.T }
B -> { HOLE:A.T }
```

But, without `Names`, the sharing constraint is impossible: `A -> { T }; B -> { T }`. (NB: `A` and `B` don't have to be implemented with the same module.)

The `Name` of a value is used to avoid ambiguous identifier errors. We state that two types are equal when their `Name`s are the same; however, for values, it is less clear why we care. But consider this example:

```
package p (A) requires (H1, H2) where
    signature H1(x) where
        x :: Int
    signature H2(x) where
        import H1(x)
    module A(y) where
        import H1
        import H2
        y = x
```

The reference to `x` in `A` is unambiguous, because it is known that `x` from `H1` and `x` from `H2` are the same (have the same `Name`.) If they were not the same, it would be ambiguous and should cause an error. Knowing the `Name` of a value distinguishes between these two cases.

Absence of `Module` in `requires` implies holes are linear Because the requirements do not record a `Module` representing the identity of a requirement, it means that it's not possible to assert that hole `A` and hole `B` should be implemented with the same module, as might occur with aliasing:

```
signature A where
signature B where
alias A = B
```

The benefit of this restriction is that when a requirement is filled, it is obvious that this is the only requirement that is filled: you won't magically cause some other requirements to be filled. The downside is it's not possible to write a package which looks for an interface it is looking for in one of n names, accepting any name as an acceptable linkage. If aliasing was allowed, we'd need a separate physical shaping context, to make sure multiple mentions of the same hole were consistent.

3.2 `signature M`

A signature declaration creates a requirement at module name `M`. It has the shape:

```
provides: (nothing)
requires: M -> { exports of renamed M }
```

Example:

```
signature H(T) where
    data T

-- provides: H -> (nothing)
-- requires: H -> { HOLE:H.T }
```

In-scope signatures are not provisions. We enforce the invariant that a provision is always (syntactically) a `module` and a requirement is always a `signature`. This means that if you have a requirement and a provision of the same name, the requirement can *always* be filled with the provision. Without this invariant, it's not clear if a provision will actually fill a signature. Consider this example, where a signature is required and exposed:

```
package a-sigs (A) requires (A) where -- ***
    signature A where
        data T

package a-user (B) requires (A) where
    signature A where
        data T
        x :: T
    module B where
        ...

package p where
    include a-sigs
    include a-user
```

When we consider merging in the shape of `a-user`, does the `A` provided by `a-sigs` fill in the `A` requirement in `a-user`? It *should not*, since `a-sigs` does not actually provide enough declarations to satisfy `a-user`'s requirement: the intended semantics *merges* the requirements of `a-sigs` and `a-user`.

```
package a-sigs (M as A) requires (H as A) where
    signature H(T) where
        data T
    module M(T) where
        import H(T)
```

We rightly should error, since the provision is a module. And in this situation:

```
package a-sigs (H as A) requires (H) where
    signature H(T) where
        data T
```

The requirements should be merged, but should the merged requirement be under the name `H` or `A`?
It may still be possible to use the `(A) requires (A)` syntax to indicate exposed signatures, but this would be a mere syntactic alternative to `() requires (exposed A)`.

3.3 include pkg (X) requires (Y)

We merge with the transformed shape of package `pkg`, where this shape is transformed by:

- Renaming and thinning the provisions according to `(X)`
- Renaming requirements according to `(Y)` (requirements cannot be thinned, so non-mentioned requirements are implicitly passed through.) For each renamed requirement from `Y` to `Y'`, substitute `HOLE:Y` with `HOLE:Y'` in the `Module`s and `Name`s of the provides and requires.

If there are no thinnings/renamings, you just merge the shape unchanged! Here is an example:

```
package p (M) requires (H) where
    signature H where
        data T
    module M where
        import H
        data S = S T

package q (A) where
    module X where
        data T = T
    include p (M as A) requires (H as X)
```

The shape of package `p` is:

```
requires: M -> { p(H -> HOLE:H):M.S }
provides: H -> { HOLE:H.T }
```

Thus, when we process the `include` in package `q`, we make the following two changes: we rename the provisions, and we rename the requirements, substituting `HOLE`s. The resulting shape to be merged in is:

```
provides: A -> { p(H -> HOLE:X):M.S }
requires: X -> { HOLE:X.T }
```

After merging this in, the final shape of `q` is:

```
provides: X -> { q():X.T }              -- from shaping 'module X'
          A -> { p(H -> q():X):M.S }
requires: (nothing)                     -- discharged by provided X
```

3.4 Merging

The shapes we've given for individual declarations have been quite simple. Merging combines two shapes, filling requirements with implementations and substituting information we learn about the identities of `Names`; it is the most complicated part of the shaping process.

The best way to think about merging is that we take two packages with inputs (requirements) and outputs (provisions) and "wiring" them up so that outputs feed into inputs. In the absence of mutual recursion, this wiring process is *directed*: the provisions of the first package feed into the requirements of the second package, but never vice versa. (With mutual recursion, things can go in the opposite direction as well.)

Suppose we are merging shape p with shape q (e.g., $p; q$). Merging proceeds as follows:

1. *Fill every requirement of q with provided modules from p.* For each requirement M of q that is provided by p (in particular, all of its required `Names` are provided), substitute each `Module` occurrence of `HOLE:M` with the provided $p(M)$, merge the names, and remove the requirement from q. Error if a provision is insufficient for the requirement.

2. If mutual recursion is supported, *fill every requirement of p with provided modules from q.*

3. *Merge leftover requirements.* For each requirement M of q that is not provided by p but required by p, merge the names. (It's not necessary to substitute `Modules`, since they are guaranteed to be the same.)

4. *Add provisions of q.* Union the provisions of p and q, erroring if there is a duplicate that doesn't have the same identity.

To merge two sets of names, union the two sets, handling each pair of names with matching `OccNames` n and m as follows:

1. If both are from holes, pick a canonical representative m and substitute n with m.

2. If one n is from a hole, substitute n with m.

3. Otherwise, error if the names are not the same.

It is important to note that substitutions on `Modules` and substitutions on `Names` are disjoint: a substitution from `HOLE:A` to `HOLE:B` does *not* substitute inside the name `HOLE:A.T`.

Since merging is the most complicated step of shaping, here are a big pile of examples of it in action.

3.4.1 A simple example

In the following set of packages:

```
package p(M) requires (A) where
    signature A(T) where
        data T
    module M(T, S) where
        import A(T)
        data S = S T

package q where
    module A where
        data T = T
    include p
```

When we `include p`, we need to merge the partial shape of `q` (with just provides `A`) with the shape of `p`. Here is each step of the merging process:

```
         shape 1                      shape 2
         ---------------------------------------------------------------------
(initial shapes)
provides: A -> THIS:A { q():A.T }      M -> p(A -> HOLE:A) { HOLE:A.T, p(A -> HOLE:A).S }
requires: (nothing)                    A ->                { HOLE:A.T }

(after filling requirements)
provides: A -> THIS:A { q():A.T }      M -> p(A -> THIS:A) { q():A.T, p(A -> THIS:A).S }
requires: (nothing)                    (nothing)

(after adding provides)
provides: A -> THIS:A         { q():A.T }
          M -> p(A -> THIS:A) { q():A.T, p(A -> THIS:A).S }
requires: (nothing)
```

Notice that we substituted `HOLE:A` with `THIS:A`, but `HOLE:A.T` with `q():A.T`.

3.4.2 Requirements merging can affect provisions

When a merge results in a substitution, we substitute over both requirements and provisions:

```
signature H(T) where
    data T
module A(T) where
    import H(T)
module B(T) where
    data T = T

-- provides: A -> THIS:A { HOLE:H.T }
--           B -> THIS:B { THIS:B.T }
-- requires: H ->        { HOLE:H.T }

signature H(T, f) where
    import B(T)
    f :: a -> a

-- provides: A -> THIS:A { THIS:B.T }           -- UPDATED
--           B -> THIS:B { THIS:B.T }
-- requires: H ->        { THIS:B.T, HOLE:H.f } -- UPDATED
```

3.4.3 Sharing constraints

Suppose you have two signature which both independently define a type, and you would like to assert that these two types are the same. In the ML world, such a constraint is known as a sharing constraint. Sharing constraints can be encoded in Backpacks via clever use of reexports; they are also an instructive example for signature merging.

```
signature A(T) where
    data T
signature B(T) where
    data T

-- requires: A -> { HOLE:A.T }
```

```
             B -> { HOLE:B.T }

-- the sharing constraint!
signature A(T) where
    import B(T)
-- (shape to merge)
-- requires: A -> { HOLE:B.T }

-- (after merge)
-- requires: A -> { HOLE:A.T }
--           B -> { HOLE:A.T }
```

I'm pretty sure any choice of `Name` is OK, since the subsequent substitution will make it alpha-equivalent.

3.5 Export declarations

If an explicit export declaration is given, the final shape is the computed shape, minus any provisions not mentioned in the list, with the appropriate renaming applied to provisions and requirements. (Requirements are implicitly passed through if they are not named.) If no explicit export declaration is given, the final shape is the computed shape, including only provisions which were defined in the declarations of the package.

3.6 Package key

What is `THIS`? It is the package name, plus for every requirement `M`, a mapping `M -> HOLE:M`. Annoyingly, you don't know the full set of requirements until the end of shaping, so you don't know the package key ahead of time; however, it can be substituted at the end easily.

Signature visibility, and defaulting The simplest formulation of requirements is to have them always be visible. Signature visibility could be controlled by associating every requirement with a flag indicating if it is importable or not: a signature declaration sets a requirement to be visible, and an explicit export list can specify if a requirement is to be visible or not.

When an export list is absent, we have to pick a default visibility for a signature. If we use the same behavior as with modules, a strange situation can occur:

```
package p where -- S is visible
    signature S where
        x :: True

package q where -- use defaulting
    include p
    signature S where
        y :: True
    module M where
        import S
        z = x && y      -- OK

package r where
    include q
    module N where
        import S
        z = y           -- OK
        z = x           -- ???
```

Absent the second signature declaration in `q`, `S.x` clearly should not be visible in `N`. However, what ought to occur when this signature declaration is added? One interpretation is to say that only some (but not all) declarations are provided (`S.x` remains invisible); another interpretation is that adding `S` is enough to treat the signature as "in-line", and all declarations are now provided (`S.x` is visible).

The latter interpretation avoids having to keep track of providedness per declarations, and means that you can always express defaulting behavior by writing an explicit provides declaration on the package. However, it has the odd behavior of making empty signatures semantically meaningful:

```
package q where
    include p
    signature S where
```

4 Type constructor exports

In the previous section, we described the `Names` of a module as a flat namespace; but actually, there is one level of hierarchy associated with type-constructors. The type:

```
data A = B { foo :: Int }
```

brings three `OccNames` into scope, `A`, `B` and `foo`, but the constructors and record selectors are considered *children* of `A`: in an import list, they can be implicitly brought into scope with `A(..)`, or individually brought into scope with `foo` or `pattern B` (using the new `PatternSynonyms` extension). Symmetrically, a module may export only *some* of the constructors/selectors of a type; it may not even export the type itself!

We *absolutely* need this information to rename a module or signature, which means that there is a little bit of extra information we have to collect when shaping. What is this information? If we take GHC's internal representation at face value, we have the more complex semantic representation seen in Figure 3:

$$
\begin{array}{rcll}
\mathit{Shape} & ::= & \texttt{provides:}\ m\ \texttt{->}\ \mathit{Module}\,\texttt{\{}\,\mathit{AvailInfo}\texttt{,}\ \ldots\,\texttt{\};}\ldots & \\
 & & \texttt{requires:}\ m\ \texttt{->}\ \qquad\qquad \texttt{\{}\,\mathit{AvailInfo}\texttt{,}\ \ldots\,\texttt{\};}\ldots & \\
\mathit{AvailInfo} & ::= & \mathit{Name} & \text{Plain identifiers} \\
 & | & \mathit{Name}\,\texttt{\{}\,\mathit{Name}_0\texttt{,}\ \ldots\texttt{,}\ \mathit{Name}_n\,\texttt{\}} & \text{Type constructors}
\end{array}
$$

Figure 3: Enriched semantic entities in Backpack

For type constructors, the outer $Name$ identifies the *parent* identifier, which may not necessarily be in scope (define this to be the `availName`); the inner list consists of the children identifiers that are actually in scope. If a wildcard is written, all of the child identifiers are brought into scope. In the following examples, we've ensured that types and constructors are unambiguous, although in Haskell proper they live in separate namespaces; we've also elided the `THIS` package key from the identifiers.

```
module M(A(..)) where
    data A = B { foo :: Int }
-- M.A{ M.A, M.B, M.foo }

module N(A) where
    data A = B { foo :: Int }
-- N.A{ N.A }

module O(foo) where
    data A = B { foo :: Int }
-- O.A{ O.foo }

module A where
    data T = S { bar :: Int }
module B where
    data T = S { baz :: Bool }
module C(bar, baz) where
    import A
    import B
-- A.T{ A.bar }, B.T{ B.baz }
-- NB: it would be illegal for the type constructors
-- A.T and B.T to be both exported from C!
```

Previously, we stated that we simply merged $Names$ based on their $OccNames$. We now must consider what it means to merge $AvailInfos$.

4.1 Algorithm

Our merging algorithm takes two sets of $AvailInfo$s and merges them into one set. In the degenerate case where every $AvailInfo$ is a $Name$, this algorithm operates the same as the original algorithm. Merging proceeds in two steps: unification and then simple union.

Unification proceeds as follows: for each pair of $Name$s with matching $OccName$s, unify the names. For each pair of $Name$ { $Name_0$, ..., $Name_n$ }, where there exists some pair of child names with matching $OccName$s, unify the parent $Name$s. (A single $AvailInfo$ may participate in multiple such pairs.) A simple identifier and a type constructor $AvailInfo$ with overlapping in-scope names fails to unify. After unification, the simple union combines entries with matching `availName`s (parent name in the case of a type constructor), recursively unioning the child names of type constructor $AvailInfo$s.

Unification of $Name$s results in a substitution, and a $Name$ substitution on $AvailInfo$ is a little unconventional. Specifically, substitution on $Name$ { $Name_0$, ..., $Name_n$ } proceeds specially: a substitution from $Name$ to $Name'$ induces a substitution from $Module$ to $Module'$ (as the $OccName$s of the $Name$s are guaranteed to be equal), so for each child $Name_i$, perform the $Module$ substitution. So for example, the substitution `HOLE:A.T` to `THIS:A.T` takes the $AvailInfo$ `HOLE:A.T { HOLE:A.B, HOLE:A.foo }` to `THIS:A.T { THIS:A.B, THIS:A.foo }`. In particular, substitution on children $Name$s is *only* carried out by substituting on the outer name; we will never directly substitute children.

4.2 Examples

Unfortunately, there are a number of tricky scenarios:

Merging when type constructors are not in scope

```
signature A1(foo) where
    data A = A { foo :: Int, bar :: Bool }

signature A2(bar) where
    data A = A { foo :: Int, bar :: Bool }
```

If we merge `A1` and `A2`, are we supposed to conclude that the types `A1.A` and `A2.A` (not in scope!) are the same? The answer is no! Consider these implementations:

```
module A1(A(..)) where
    data A = A { foo :: Int, bar :: Bool }

module A2(A(..)) where
    data A = A { foo :: Int, bar :: Bool }

module A(foo, bar) where
    import A1
    import A2
```

Here, `module A1` implements `signature A1`, `module A2` implements `signature A2`, and `module A` implements `signature A1` and `signature A2` individually and should certainly implement their merge. This is why we cannot simply merge type constructors based on the $OccName$ of their top-level type; merging only occurs between in-scope identifiers.

Does merging a selector merge the type constructor?

```
signature A1(A(..)) where
    data A = A { foo :: Int, bar :: Bool }
```

```
signature A2(A(..)) where
    data A = A { foo :: Int, bar :: Bool }

signature A2(foo) where
    import A1(foo)
```

Does the last signature, which is written in the style of a sharing constraint on `foo`, also cause `bar` and the type and constructor `A` to be unified? Because a merge of a child name results in a substitution on the parent name, the answer is yes.

Incomplete data declarations

```
signature A1(A(foo)) where
    data A = A { foo :: Int }

signature A2(A(bar)) where
    data A = A { bar :: Bool }
```

Should `A1` and `A2` merge? If yes, this would imply that data definitions in signatures could only be *partial* specifications of their true data types. This seems complicated, which suggests this should not be supported; however, in fact, this sort of definition, while disallowed during type checking, should be *allowed* during shaping. The reason that the shape we abscribe to the signatures `A1` and `A2` are equivalent to the shapes for these which should merge:

```
signature A1(A(foo)) where
    data A = A { foo :: Int, bar :: Bool }

signature A2(A(bar)) where
    data A = A { foo :: Int, bar :: Bool }
```

4.3 Subtyping record selectors as functions

```
signature H(foo) where
    data A
    foo :: A -> Int

module M(foo) where
    data A = A { foo :: Int, bar :: Bool }
```

Does `M` successfully fill `H`? If so, it means that anywhere a signature requests a function `foo`, we can instead validly provide a record selector. This capability seems quite attractive but actually it is quite complicated, because we can no longer assume that every child name is associated with a parent name.

As a workaround, `H` can equivalently be written as:

```
signature H(foo) where
    data A = A { foo :: Int, bar :: Bool }
```

This is suboptimal, however, as the otherwise irrelevant `bar` must be mentioned in the definition.

So what if we actually want to write the original signature `H`? The technical difficulty is that we now need to unify a plain identifier $AvailInfo$ (from the signature) with a type constructor $AvailInfo$ (from a module.) It is not clear what this should mean. Consider this situation:

```
package p where
```

```
    signature H(A, foo, bar) where
        data A
        foo :: A -> Int
        bar :: A -> Bool
    module X(A, foo) where
        import H
package q where
    include p
    signature H(bar) where
        data A = A { foo :: Int, bar :: Bool }
    module Y where
        import X(A(..)) -- ???
```

Should the wildcard import on `X` be allowed? Probably not? How about this situation:

```
package p where
    -- define without record selectors
    signature X1(A, foo) where
        data A
        foo :: A -> Int
    module M1(A, foo) where
        import X1

package q where
    -- define with record selectors (X1s unify)
    signature X1(A(..)) where
        data A = A { foo :: Int, bar :: Bool }
    signature X2(A(..)) where
        data A = A { foo :: Int, bar :: Bool }

    -- export some record selectors
    signature Y1(bar) where
        import X1
    signature Y2(bar) where
        import X2

package r where
    include p
    include q

    -- sharing constraint
    signature Y2(bar) where
        import Y1(bar)

    -- the payload
    module Test where
        import M1(foo)
        import X2(foo)
        ... foo ... -- conflict?
```

Without the sharing constraint, the `foo`s from `M1` and `X2` should conflict. With it, however, we should conclude that the `foo`s are the same, even though the `foo` from `M1` is *not* considered a child of `A`, and even though in the sharing constraint we *only* unified `bar` (and its parent `A`). To know that `foo` from `M1` should also be unified, we have to know a bit more about `A` when the sharing constraint performs unification; however, the $AvailInfo$ will only tell us about what is in-scope.

5 Type checking

Type checking computes, for every `Module`, a `ModIface` representing the type of the module in question:

```
Type ::= { Module "->" ModIface }
```

5.1 The basic plan

Given a module or signature, we can type check given these two assumptions:

- We have a renamed syntax tree, whose identifiers have been resolved as according to the result of the shaping pass.
- For any `Name` in the renamed tree, the corresponding `ModDetails` for the `Module` has been loaded (or can be lazily loaded).

The result of type checking is a `ModDetails` which can then be converted into a `ModIface`. Arranging for these two assumptions to be true is the bulk of the complexity of type checking.

5.2 A little bit of night music

A little bit of background about the relationship of GHC `ModIface` and `ModDetails`.

A `ModIface` corresponds to an interface file, it is essentially a big pile of `Name`s which have not been resolved to their locations yet. Once a `ModIface` is loaded, we type check it (`tcIface`), which turns them into `TyThing`s and `Type`s (linked up against their true locations.) Conversely, once we finish type checking a module, we have a `ModDetails`, which we then serialize into a `ModIface`.

One very important (non-obvious) distinction is that a `ModDetails` does *not* contain the information for handling renaming. (Actually, it does carry along a `md_exports`, but this is only a hack to transmit this information when we're creating an interface; no code actually uses it.) So any information about reexports is recorded in the `ModIface` and used by the renamer, at which point we don't need it anymore and can drop it from `ModDetails`.

5.3 Loading modules from indefinite packages

Everything is done modulo a shape Consider this package:

```
package p where
    signature H(T) where
        data T = T
    module A(T) where
        data T = T
    signature H(T) where
        import A(T)

-- provides: A -> THIS:A { THIS:A.T }
--           H -> HOLE:H { THIS:A.T }
-- requires: H ->        { THIS:A.T }
```

With this shaping information, when we are type-checking the first signature for `H`, it is completely wrong to try to create a definition for `HOLE:H.T`, since we know that it refers to `THIS:A.T` via the requirements of the shape. This applies even if `H` is included from another package. Thus, when we are loading `ModDetails` into memory, it is always done *with respect to some shaping*. Whenever you reshape, you must clear the module environment.

Figuring out where to consult for shape information For this example, let's suppose we have already type-checked this package `p`:

```
package p (A) requires (S) where
    signature S where
        data S
        data T
    module A(A) where
        import S
        data A = A S T
```

giving us the following `ModIface`s:

```
module HOLE:S.S where
    data S
    data T
module THIS:A where
    data A = A HOLE:S.S HOLE:S.T
-- where THIS = p(S -> HOLE:S)
```

Next, we'd like to type check a package which includes `p`:

```
package q (T, A, B) requires (H) where
    include p (A) requires (S as H)
    module T(T) where
        data T = T
    signature H(T) where
        import T(T)
    module B(B) where
        import A
        data B = B A
```

Prior to typechecking, we compute its shape:

```
provides: (elided)
requires: H -> { HOLE:H.S, THIS:T.T }
-- where THIS = q(H -> HOLE:H)
```

Our goal is to get the following type:

```
module THIS:T where
    data T = T
module THIS:B where
    data B = B p(S -> HOLE:H):A.A
        -- where data A = A HOLE:H.S THIS:T.T
-- where THIS = q(H -> HOLE:H)
```

This type information does *not* match the pre-existing type information from `p`: when we translate the `ModIface` for `A` in the context into a `ModDetails` from this typechecking, we need to substitute `Name`s and `Module`s as specified by shaping. Specifically, when we load `p(S -> HOLE:H):A` to find out the type of `p(S -> HOLE:H):A.A`, we need to take `HOLE:S.S` to `HOLE:H.S` and `HOLE:S.T` to `THIS:T.T`. In both cases, we can determine the right translation by looking at how `S` is instantiated in the package key for `p` (it is instantiated with `HOLE:H`), and then consulting the shape in the requirements.

This process is done lazily, as we may not have typechecked the original `Name` in question when doing this. `hs-boot` considerations apply if things are loopy: we have to treat the type abstractly and re-typecheck it to the right type later.

5.4 Re-renaming

Theoretically, the cleanest way to do shaping and typechecking is to have shaping result in a fully renamed syntax tree, which we then typecheck: when done this way, we don't have to worry about logical contexts (i.e., what is in scope) because shaping will already have complained if things were not in scope.

However, for practical purposes, it's better if we don't try to keep around renamed syntax trees, because this could result in very large memory use; additionally, whenever a substitution occurs, we would have to substitute over all of the renamed syntax trees. Thus, while type-checking, we'll also re-compute what is in scope (i.e., just the `OccName` bits of `provided`). Nota bene: we still use the `Name`s from the shape as the destinations of these `OccName`s! Note that we can't just use the final shape, because this may report more things in scope than we actually want. (It's also worth noting that if we could reduce the set of provided things in scope in a single package, just the `Shape` would not be enough.)

5.5 Merging `ModDetails`

After type-checking a signature, we may turn to add it to our module environment and discover there is already an entry for it! In that case, we simply merge it with the existing entry, erroring if there are incompatible entries.