

The Backpack algorithm

April 27, 2015

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

1 Front-end syntax

For completeness, here is the package language we will be shaping and typechecking:

```
package      ::= "package" pkgname [pkgexports] "where" pkgbody
pkgbody      ::= "{" pkgdecl_0 ";" ... ";" pkgdecl_n "}"
pkgdecl      ::= "module"      modid [exports] where body
               | "signature"  modid [exports] where body
               | "include"    pkgname [inclspec]
inclspec      ::= "(" renaming_0 "," ... "," renaming_n [","] ")"
               [ "requires" "(" renaming_0 "," ... "," renaming_n [","] "]" ]
pkgexports    ::= inclspec
renaming      ::= modid "as" modid
```

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 `modid` renaming is simply `modid as modid`.)

2 Shaping

Shaping computes a `Shape` which has this form:

```
Shape ::= provides: { ModName -> Module { Name } }
        requires: { ModName ->      { Name } }

PkgKey      ::= SrcPkgId "(" { ModName -> Module } ")"
               | HOLE
Module      ::= PkgKey ":" ModName
Name        ::= Module "." OccName
OccName     ::= -- a plain old name, e.g. undefined, Bool, Int
```

Starting with the empty shape, we incrementally construct a shape by shaping package declarations (the partially constructed shape serves as a context for renaming modules and signatures and instantiating includes) and merging them until we have processed all declarations. There are two things to specify: what shape each declaration has, and how the merge operation proceeds.

One variation of shaping also computes the renamed version of a package, i.e., where each identifier in the module and signature is replaced with the original name (equivalently, we record the output of GHC’s renaming pass). This simplifies type checking because you no longer have to recalculate the set of available names, which otherwise would be lost. See more about this in the type checking section.

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

2.1 module M

Merge with this shape:

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

Example:

```
-- provides: (nothing)  
-- requires: (nothing)  
  
module A(T) where  
  data T = T  
  
-- provides: A -> THIS:A { THIS:A.T }          -- NEW  
-- requires: (nothing)  
  
module M(T, f) where  
  import A(T)  
  f x = x  
  
-- provides: A -> THIS:A { THIS:A.T }  
              M -> THIS:M { THIS:A.T, THIS:M.f } -- NEW  
-- requires: (nothing)
```

2.2 signature M

Merge with this shape:

```
provides: { M -> HOLE:M { exports of renamed M } }
requires: { M ->          { exports of renamed M } }
```

Example:

```
-- provides: (nothing)
-- requires: (nothing)

signature H(T) where
  data T

-- provides: H -> HOLE:H { HOLE:H.T }
-- requires: H ->          { HOLE:H.T }

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

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

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

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

Notice that in the last example, when a signature with reexports is merged, it can result in updates to the shapes of other module names.

2.3 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 passed through.) For each renamed requirement from `Y` to `Y'`, substitute `HOLE:Y` with `HOLE:Y'` in the `Modules` and `Names` of the `provides` and `requires`. (Freshen holes.)
- If there are no thinnings/renamings, you just merge the shape unchanged!

Example:

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

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

package q (A) where
  module X where
    data T = T

-- provides: X -> { q():X.T }
-- requires: (nothing)

include p (M as A) requires (H as X)
-- 1. Rename/thin provisions:
--   provides: A -> { p(H -> HOLE:H):M.S }
--   requires: H -> { HOLE:H.T }
-- 2. Rename requirements, substituting HOLES:
--   provides: A -> { p(H -> HOLE:X):M.S }
--   requires: X -> { HOLE:X.T }

-- (after merge)
-- provides: X -> { q():X.T }
--           A -> { p(H -> q():X):M.S }
-- requires: (nothing)
```

2.4 Merging

Merging combines two shapes, filling requirements with implementations and substituting information we learn about the identities of `Names`. Importantly, merging is a *directed* process, akin to taking two boxes with input and output ports and wiring them up so that the first box feeds into the second box. This algorithm does not support mutual recursion.

Suppose we are merging shape p with shape 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 , substitute each `Module` occurrence of `HOLE:M` with the provided $p(M)$, merge the names, and remove the requirement from q .
2. *Merge in provided signatures of q , add the provided modules of q .* For each provision M of q : if $q(M)$ is a hole, substitute every `Module` occurrence of `HOLE:q(M)` with $p(M)$ if it exists and merge the names; otherwise, add it to p , erroring if $p(M)$ exists.

Substitutions apply to both shapes. To merge two sets of names, take each pair of names with matching `OccNames` n and m .

1. If both are from holes, pick a canonical representative m and substitute n with m . (E.g., pick the one with the lexicographically first `ModName`).
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`. Here is a simple example:

<pre> shape 1 provides: A -> THIS:A { q():A.T } requires: (nothing) </pre>	<pre> shape 2 M -> p(A -> HOLE:A) { HOLE:A.T, p(A -> HOLE:A).S } A -> { HOLE:A.T } </pre>
<pre> (after filling requirements) provides: A -> THIS:A { q():A.T } requires: (nothing) </pre>	
<pre> (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) </pre>	

Here are some more involved examples, which illustrate some important cases:

2.4.1 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. For brevity, we've omitted `provided` from the shapes in this example.

```

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

```

```

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

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

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

```

Notably, we could equivalently have chosen `HOLE:B.T` as the post-merge name. *Actually, I don't think any choice can be wrong. The point is to ensure that the substitution applies to everything we know about, and since requirements monotonically increase in size (or are filled), this will hold.*

2.4.2 Provision linking does not discharge requirements

It is not an error to define a module, and then define a signature afterwards: this can be useful for checking if a module implements a signature, and also for sharing constraints:

```

module M(T) where
  data T = T
signature S(T) where
  data T

signature M(T)
  import S(T)
-- (partial)
-- provides: S -> HOLE:S { THIS:M.T } -- resolved!

-- alternately:
signature S(T) where
  import M(T)

```

However, in some circumstances, linking a signature to a module can cause an unrelated requirement to be “filled”:

```

package p (S) requires (S) where
  signature S where
    data T

package q (A) requires (B) where
  include S (S as A) requires (S as B)

package r where
  module A where
    data T = T
  include q (A) requires (B)
  -- provides: A -> THIS:A { THIS:A.T }
  -- requires: (nothing)

```

Notice that the requirement was discharged because we unified `HOLE:B` with `THIS:A`. While this is certainly the most accurate picture of the package we can get from this situation, it is a bit unsatisfactory in that looking at the text of module `r`, it is not obvious that all the requirements were filled; only that there is some funny business going on with multiple provisions with `A`.

Note that we *cannot* disallow multiple bindings to the same provision: this is a very important use-case when you want to include one signature, include another signature, and see the merge of the two signatures in your context. *So maybe this is what we should do.* However, there is a saving grace, which is signature-signature linking can be done when linking requirements; linking provisions is unnecessary in this case. So perhaps the merge rule should be something like:

1. *Fill every requirement of q with provided modules from p .* For each requirement M of q that is provided by p , substitute each `Module` occurrence of `HOLE:M` with the provided $p(M)$, merge the names, and remove the requirement from q .
2. *Merge requirements.* For each requirement M of q that is not provided by p but required by p , merge the names.
3. *Add provisions of q .* For each provision M of q : add it to p , erroring if the addition is incompatible with an existing entry in p .

Now, because there is no provision linking, and the requirement `B` is not linked against anything, `A` ends up being incompatible with the `A` in context and is rejected. We also reject situations where a provision unification would require us to pick a signature:

```
package p (S) requires (S) where
  signature S
```

```
package q where
  include p (S) requires (S as A)
  include p (S) requires (S as B)
  -- rejected; provided S doesn't unify
  -- (if we accepted, what's the requirement? A? B?)
```

How to relax this so `hs-boot` works

Example of how loopy modules which rename requirements make it un-obvious whether or not a requirement is still required. Same example works declaration level.

package `p` (`A`) requires (`A`); the input output ports should be the same

2.5 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. (Provisions are implicitly passed through if they are not named.)

If no explicit export declaration is given, the final shape is the computed shape, minus any provisions which did not have an in-line module or signature declaration.

Guru meditation. The defaulting behavior for signatures is slightly delicate, as can be seen in this example:

```
package p (S) requires (S) where
  signature S where
    x :: True

package q where
  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. 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
```

Note that if `p` didn’t provide `S`, `x` would *never* be visible unless it was redeclared in an interface.

2.6 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.

3 Type checking

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

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

3.1 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.

3.2 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 `Names` which have not been resolved to their locations yet. Once a `ModIface` is loaded, we type check it (`tcIface`), which turns them into `TyThings` and `Types` (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`.

3.3 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 `ModIfaces`:

```
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 `Names` and `Modules` 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.

3.4 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 `Names` from the shape as the destinations of these `OccNames`! 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.)

3.5 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.