The Backpack algorithm

April 28, 2015

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

1 Changelog

April 28, 2015 A signatures declaration no longer provides a signature in the technical shaping sense; the motivation for this change is explained in **Signatures cannot be provided**. This means that, by default, all requirements are importable (Derek has stated that he doesn't think this will be too much of a problem in practice); we can consider extensions which allow us to hide requirements from import.

2 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`.)

3 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     ::= 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.

`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 `Name`s, 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.

3.1 `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)
```

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 }
```

Signatures cannot be provided. A signature *never* counts as a provision, as far as shaping is concerned. While it seems like a signature package which provides signatures for import should actually, you know, *provide* its signatures, this observation at its logical conclusion is a mess. The problem can most clearly be seen in this example:

```
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`, but doesn't use the provision to fill the signature. However, in this case:

```
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 `Name`s; 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 `Name`s 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 `Module`s, 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, take each pair of names with matching `OccName`s n and m.

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 `Module`s and substitutions on `Name`s 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.

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
```

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.

4 Type checking

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

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

4.1 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.

4.2 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`.

4.3 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 `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.

4.4 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.)

4.5 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.