Backpack shaping by example

April 23, 2015

Note: this document assumes familiarity with the syntax of Backpack. Go read the Backpack Manual (`http://web.mit.edu/~ezyang/Public/backpack-manual.pdf`) if you haven't already.

Guru meditation. These asides contain more complex examples which justify certain design choices. They can be skipped without missing out on important information. You might have to have read further to understand them.

---

1 What is shaping?

When you write an ordinary Haskell package, if you define the data type `ByteString` in `Data.ByteString.Lazy`, and define it again in `Data.ByteString.Strict`, you do not expect these types to be the same.

However, if you are doing modular programming with module interfaces, you might want to define a type in a module interface, but not say where it comes from: that's the job of whoever implements the interface. `ByteString` defined in two interfaces could be the same... or they might be different. Shaping tells you whether or not they are the same.

Imagine you have two signature packages: `haskell98-base-sigs`, which just exports `Prelude` defining `Int`; and `ghc-base-sigs`, which provides a more internal version of `Int` in `GHC.Base`, with `Prelude` simply reexporting the definition there.

```
package haskell98-base-sigs (Prelude) where
    signature Prelude(Int) where
        data Int

package ghc-base-sigs (Prelude, GHC.Base) where
    include ghc-prim
    signature GHC.Base(Int(..)) where
        import GHC.Prim(Int#)
        data Int = I# Int#
    signature Prelude(Int) where
        import GHC.Base
```

Now, suppose you want to type-check a package which is using both signatures at the same time:

```
package p (A) where
    include haskell98-base-sigs
    include ghc-base-sigs
    module A where
        import Prelude
        import GHC.Base
        ... Int ...
```

There are two places where `Int` is defined, and we ought not to accept `A` unless `haskell98-base-sigs:Prelude.Int` and `ghc-base-sigs:GHC.Base.Int` are the same. In fact, they are the same;[1] however, if you remove module

[1] The reason is `p` *linked* the two `Prelude`s together: they must be implemented with the same module. Since any implemen-

ghc-base-sigs:Prelude, the two Ints are no longer equal!

Shaping is the process responsible for concluding that these two types are equal. By the end of this document, you should understand how and why Int is type equal in the previous example, as well as understand other examples.

2 Defining pre-shape

Informally, a package consists of a collection of modules and signatures, which, given some *required* holes at some module names, can *provide* some modules (at some other module names) for import by anyone who includes the package. The requires and provides of a package can be written explicitly in the header of a package:

```
pkgexports ::= { ModName } "requires" { ModName }
```

Here are the explicit headers of the packages in the introductory example:

```
package haskell98-base-sigs (Prelude)           requires (Prelude)
package ghc-base-sigs       (Prelude, GHC.Base) requires (Prelude, GHC.Base)
package p                   (A)                 requires (Prelude, GHC.Base)
```

When you instantiate a package, an instance is identified by a *package key*, which what module each hole was instantiated with (or the module is put in a special HOLE package if it was not instantiated at all, as you might when type-checking a package which still has holes in it):

```
PkgKey  ::= SrcPkgId "(" { ModName "->" Module } ")"
          | HOLE
```

A module might get instantiated multiple times (when its package is instantiated multiple times): a particular instance of a module is identified by the its enclosing package key plus its module name:

```
Module  ::= PkgKey ":" ModName
```

To infer the provides and requires of a package, however,

The full pre-shape of a package, however, also specifies the module identities of everything it exports:

```
preshape ::= { ModName "->" Module } "requires" { ModName "->" Module }

haskell98-base-sigs
    provides: Prelude  -> HOLE:Prelude

ghc-base-sigs
    provides: Prelude  -> HOLE:Prelude
              GHC.Base -> HOLE:GHC.Base
p
    provides: A        -> p(Prelude  -> HOLE:Prelude,
                            GHC.Base -> HOLE:GHC.Base):A
```

Depending on how we vary how the requirements of a package are filled, the types and values defined in the package may be different. So a mapping of required hole names to proper modules uniquely defines an instance of the package: we identify these instances with *package keys* (PkgKey).

An example package key would be prelude-sig(Prelude -> base():Prelude), where prelude-sig has a single requirement that was filled by the Module base():Prelude.

tation of Prelude can only define one entity named Int, we can infer that the separate Ints in the signatures must be the same;

3 Defining shape

A *shape* adds more information about the declarations that the module exports. Nothing as fancy as a full type; just a `Name` which identifies the name in question. We'll say more about what `Name`s are shortly, but the important property is that if two `Name`s of two types are the same, they are type-equal (value-equal in the case of values). We can thus define a shape as a mapping of module name to the set of `Name`s it provides and the set of `Name`s it requires.

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

We should say a little bit about `Name`s. This terminology comes from the internals of GHC, where it is very useful to have a representation of identity that distinguishes a type from anything else. In old versions of GHC, a `Name` was the source package ID (`bytestring-0.1`) plus the module name it was defined in (`Data.ByteString.Lazy`) plus the actual name of the type (`ByteString`). As a simplifying assumption in this document, we'll assume version numbers don't exist, but technically everywhere there is a package name in this document, there should also be a version number.

In Backpack, this is *still* not enough: we must also record the mapping from each required module name to the actual `Module` which is fulfilling that requirement. Thus, the full specification of `Name` (omitting version numbers) is:

```
Name    ::= Module "." OccName
OccName ::= -- a plain old name, e.g. undefined, Bool, Int
```

and by inference, that `ghc-base-sigs:GHC.Base.Int` is equivalent as well.

Mini-guru meditation. Why do we need the mapping of holes to modules? Consider:

```
package p (A) requires (H) where
    signature H(T) where
        data T
    module A(A) where
        import H
        data A = A T
package q (A1, A2) where
    module H1(T) where
        data T = T Int
    module H2(T) where
        data T = T Bool
    include p (A as A1) requires (H as H1)
    include p (A as A2) requires (H as H2)
```

If we conclude that `A1.T` = `A2.T`, that would be disaster!

Guru meditation. 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)
```

The sets of modules provided for both includions of `p` are the same, but `A12.A :: I1.T -> I2.T -> A12.A` while `A21.A :: I2.T -> I1.T -> A12.A`.

Guru meditation. Why can't the required portion of a shape refer to `OccName`s instead of `Name`s, e.g. `"required:" { ModName "->" { OccName } }`? Consider:

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

This has the shape:

```
provided: (empty)
required:
    A -> { HOLE:A.T }
    B -> { HOLE:A.T }
```

In particular, we conclude `A.T` = `B.T`.

Guru meditation. Why do `Name`s matter for values? Consider:

```
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 this was not known, it would be ambiguous and cause an error.

---

4 How to shape

You might consider skipping this section and reading some of the examples, before coming back.

Here is the core Backpack language (minus some syntactic sugar and ascription.)

```
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 ["," ] ")" -- (provides list)
             [ "requires" "(" renaming_0 "," ... "," renaming_n ["," ] ")" ]
pkgexports ::= inclspec
renaming ::= modid [ "as" modid ]
```

Shaping proceeds in a few steps:

Pre-shaping Pre-shaping recursively calculates the provided and required module names of packages. Equivalently, it elaborates package declarations and includes so that `pkgexports` and `inclspec` are specified explicitly.

The pre-shape of a package is calculated by processing declarations in order, calculating a set of provided module names P (modules we are planning to expose outside the package), available module names A (modules which can be imported and fill requirements) and required module names R (requirements that must be filled by a user of the package). Then, absent a `pkgexports`, the shape of the package is (provides: P, requires: R).

Module Given "`module M`": let $P' = P \cup \{M\}$, $A' = A \cup \{M\}$ and $R' = R - \{M\}$. A module definition is both provided and available, and fills any requirement with the same name.

Signature Given "`signature S`": let $R' = R \cup \{S\}$ if $S \notin A$, and no change otherwise. A signature definition creates a requirement if there is not already another definition available. This definition could be another signature, in which case $S \in R$ already!

Include Let the pre-shape of the included package be (provides: P_I, requires: R_I).
Given "`include pkgname (X0 as X'0, ..., Xn as X'n) requires (Y0 as Y'0, ..., Yn as Yn')`":

- Fail if `X0, ..., Xn` $\not\subseteq P_I$
- Fail if `Y0, ..., Yn` $\not\subseteq R_I$
- Let $A' = A \cup \{$ `X'0, ..., X'n` $\}$
- Let $R_0 =$
- Let $R' = R - \{$ `X'0, ..., X'n` $\} + \{$ `Y'0, ..., Y'n` $\}$
- Add `InclRequires` minus `Y0, ..., Yn` to R, for all not in A

If you have a sole `Xi` in any renaming list, it is sugar for `Xi as Xi`. When an `inclspec` is absent, let the `inclspec` be P_I `requires` R_I.

5 Definite packages are simple

When there aren't any signatures, package shapes are simple: given an identifier named `T` declared in a module `A` in a package `p`, the module `A` provides the name `p():A.T`. Thus

```
package p (A) where
    module A(T,x) where
        data T = T
        x = False
```

has the shape

```
provides:
    A -> { p():A.T, p():A.x }
requires:
    (nothing)
```

Reexports The Haskell source-language supports reexports. In such a case, the shape of the module reports the *original* name.

```
package p(A,B) where
    module A(T) where
```

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

has shape

```
provides:
    A -> { p():A.T }
    B -> { p():A.T } -- not p():B.T!
requires:
    (nothing)
```

Haskell does not support changing the `OccName` upon reexport; the usual way of renaming types and values results in a new `Name`.

```
package p (A,B) where
    module A(T, x) where
        data T = T
        x = True
    module B(S, y) where
        import A
        type S = T
        y = x
```

has shape

```
provides:
    A -> { p():A.T, p():A.x }
    B -> { p():B.S, p():B.y } -- not p():A.T, p():A.x!
requires:
    (nothing)
```

Guru meditation. If we can change `OccName`s on reexport, we need a different definition of shape:

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

Without `OccName` renaming, the `OccName` always equals the `OccName` of the `Name`.

---

6 Signatures in indefinite packages

If there is a signatures, we say its identifiers are from the special `HOLE` package. (These are a bit like skolem variables.) Signatures add to the requirement of a module shape in addition to the provides.

```
package p-sig (A) requires (A) where
    signature A(T,x) where
        data T
        x :: Bool
```

has shape

```
provides:
    A -> { HOLE:A.T, HOLE:A.x }
requires:
    A -> { HOLE:A.T, HOLE:A.x }
```

No export You don't have to export a signature, but you must require it. In that case, it is required but not provided.

```
package p-sig (B) requires (A,B) where
    signature A(T) where
        data T
    signature B(S) where
        import A
        data S = S T
```

has shape

```
provides:
    B -> { HOLE:B.S }
requires:
    A -> { HOLE:A.T }
    B -> { HOLE:B.S }
```

Reexports Signatures also support reexports. They work in the same way as in modules.

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

has shape

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

Signatures can import modules too!

7 Modules in indefinite packages

When you define a module in a package with holes, when constructing the package key for names defined in this module, you must also specify how the holes in the package are filled in. For example:

```
package p (A) requires (H) where
    signature H where
        x :: Bool
    module A where
        import H
        y = x
```

has shape

```
provides:
    A -> { p(H -> HOLE:H):A.y } -- not p():A.y!
requires:
    H -> { HOLE:H.x }
```

The mapping `H -> HOLE:H` says that `p` was instantiated with

8 Includes

An include brings the shape of the package included into the context of our package:

```
package p (A) where
    module A(x) where
        x = True

package q (A, B) where
    include A
    module B(y) where
        y = True
```

p provides `A -> { p:A.x }`, while q has shape:

```
provides:
    A -> { p:A.x }
    B -> { q:B.y }
requires:
    (nothing)
```

If none of the module names from the included package and the current package overlap, things are simple. Things are more complex when there are overlapping names: in this case, *linking* should occur.

Renaming holes If you rename a hole, the occurrences of `HOLE:A` in modules and names are renamed:

```
package p (M) requires (A) where
    signature A(x) where
        x :: Bool
    module M(y) where
        import A
        y = x

package q (M) requires (B) where
    include A (M) requires (A as B)
```

has shapes:

```
p provides:
    M -> { p(A -> HOLE:A):M.y }
p requires:
    A -> { HOLE:A.x }

q provides:
    M -> { p(A -> HOLE:B):M.y }
q requires:
    B -> { HOLE:B.x }
```

Linking a signature with an implementation If you fill a hole with an implementation, the occurrences of the hole's `Module`, e.g. `HOLE:A`, are replaced with the implementation module identity, and the occurences of the hole's `Name`s, e.g. `HOLE:A.x`, are replaced with the implementation's matching `Name` (e.g., having the same `OccName`). These are two separate substitutions!

```
package p (B) requires (A) where
    signature A(T) where
        data T
    module B(T, x) where
        import A(T)
        x :: Bool

package q (A, B) where
    module A(T) where
        data T = T
    include p
```

has shapes:

```
p provides:
    B -> { HOLE:A.T, p(A -> HOLE:A):B.x }
p requires:
    A -> { HOLE:A.T }

q provides:
    B -> { q():A.T, p(A -> q():A):B.x }
    A -> { q():A.T }
q requires:
    (nothing)
```

Note that I can also include first and then define the module; the result is the same.

Guru meditation. Why can't we just replace all occurrences of `HOLE:A` with `q():A`? A modified `package q`:

```
package q (TyA, A, B) where
    module TyA(T) where
        data T = T
    module A(T) where
        import TyA(T)
    include p
```

should have shape:

```
q provides:
    TyA -> { q():TyA.T }
    A   -> { q():TyA.T }
    B   -> { q():TyA.T, p(A -> q():A):B.x }     -- NB: not p(A -> q():TyA)
q requires:
    (nothing)
```

`HOLE:A.T` is substituted with `q():TyA.T`, but `HOLE:A` is substituted with `q():A`!

Linking a signature with a signature

```
package p requires (H) where
    include base
    signature H(Int) where
        import Prelude

package q requires (H) where
```

```
include base
```