The Backpack Manual

April 24, 2015

Backpack is a new module system for Haskell, intended to enable "separate modular development": a style of development where application-writers can develop against abstract interfaces or *signatures*, while separately library-writers write software which implement these interfaces. Backpack was originally described in a POPL'14 paper `http://plv.mpi-sws.org/backpack/`, but the point of this document is to describe the syntax of a language you might actually be able to *write*, as well as describe some of the changes to the design we've made since this paper.

Examples Before we dive in, here are some examples of Backpack files to whet your appetite:

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

package q(B, C) where
    include p
    module B(b) where
        import A
        b = a
    module C(c) where
        import B
        c = b
```

In this example package `p` exports a single module `A`; package `q` exports two modules, and includes package `p` so that its modules are in scope. The form of a `package` and a `module` are quite similar: a package can express an explicit export list of modules in much the same way a module exports identifiers.

Here is a more complicated Backpack file taking advantage of the availability of signatures in Backpack. This example omits the actual module bodies: GHC will automatically look for them in appropriate files (e.g. `p/Map.hsig`, `p/P/Types.hs`, `p/P.hs`, `q/QMap.hs` and `q/Q.hs`).

```
package p (P) requires (Map) where
    include base
    signature Map
    module P.Types
    module P

package q (Q) where
    include base
    module QMap
    include p (P) requires (Map as QMap)
    module Q
```

Package `p` provides a module `P`, but it also *requires* a module `Map` (which must fulfill the specification described in `signature Map`). This makes it an *indefinite package*: a package which isn't fully implemented, and cannot be compiled into executable code until its "hole" (`Map`) is filled. It also sports an internal module named `P.Types` which is missing from the export list, and thus not exported.

Package `q`, on the other hand, is a *definite package*; as it has no requirements, it can be compiled. When it includes `p` into scope, it also fills the `Map` requirement with an implementation `QMap`.

1 The Backpack file

```
packages ::= "{" package_0 ";" ... ";" package_n "}"
```

A Backpack file consists of a list of named packages. All packages in a Backpack file live in the global namespace. A package defines a collection of modules, exporting some of these modules so that other modules can make use of them via *include*. You can compile a definite package `p` in a Backpack file `foo.bkp` with `ghc --backpack foo.bkp p`; you can type-check an indefinite package by adding `-fno-code`.

ToDo: How do you import an external Backpack file?

ToDo: Facility for private packages.

```
package ::= "package" pkgname [pkgexports] "where" pkgbody
pkgname ::= /* package name, e.g. containers (no version!) */
pkgbody ::= "{" pkgdecl_0 ";" ... ";" pkgdecl_n "}"
```

A package begins with the keyword `package`, its name, and an optional export specification (e.g., a list of modules to be exposed). The header is followed a list of declarations which define modules, signatures and include other packages.

2 Declarations

```
pkgdecl ::= "module"    modid [exports] "where" body
          | "signature" modid [exports] "where" body
          | "include"   pkgname ["as" pkgname] [inclspec]
```

A package is made up of package declarations, which either introduce a new module implementation, introduces a new module signature, or includes a package from the package environment. The declaration of modules and signatures is exactly as it is in Haskell98, so we don't reprise the grammar here.

ToDo: Clarify whether order matters. Both are valid designs, but I think order-less is more user-friendly.

We generally don't expect users to place their module source code in package files; thus we provide the following forms to refer to `hs` and `hsig` files living on the file-system:

```
pkgdecl ::= "module"    modid_0 "=" path_0 "," ... "," modid_n "=" path_n
          | "signature" modid_0 "=" path_0 "," ... "," modid_n "=" path_n
          | "module"    modid_0 "," ... "," modid_n
          | "signature" modid_0 "," ... "," modid_n
```

Thus, `module A = "A.hs"` defines the body of `A` based on the contents of `A.hs` in the package's source directory. When the assignment is omitted, we implicitly refer to the file path created by replacing periods with directory separators and adding an appropriate file extension (thus, we can also write `module A`).

```
pkgdecl ::= "source" path
```

The `source` keyword is another construct which allows us to define modules by simply scanning the path in question. For example, if `src` contains two files, `A.hs` and `B.hsig`, then "`source "src"`" is equivalent to "`module A = "src/A.hs"; signature B = "src/B.hsig"`".

ToDo: Allow defining package-wide module imports, which propagate to all inline modules and signatures.

ToDo: Allow defining anonymous modules with bare type/expression declarations.

3 Signatures

A signature, denoted with `signature` in a Backpack file and a file with the `hsig` extension on the file system, represents a (type) signature for a Haskell module. It can contain type signatures, data declarations, type classes, type class instances and reexports(!), but it cannot contain any value definitions.[1] Signatures are essentially `hs-boot` modules which do not support mutual recursion but have no runtime efficiency cost. Here is an example of a module signature representing an abstract map type:

```
module Map where
    type role Map nominal representational
    data Map k v
    instance Functor (Map k)
    empty :: Map k a
```

4 Includes and exports

```
pkgdecl  ::= "include" pkgname ["as" pkgname] [inclspec]

inclspec ::= "(" renaming_0 "," ... "," renaming_n ["," ] ")"
             [ "requires" "(" renaming_0 "," ... "," renaming_n ["," ] ")" ]

renaming ::= modid [ "as" modid ]
           | "package" pkgname
```

An include brings the modules and signatures of a package into scope. If these modules/signatures have the same names as other modules/signatures in scope, *mix-in linking* occurs. In particular:

- Module + module = error (unless they really are the same!)
- Module + signature = the signature is filled in, and is no longer part of the requirements of the package.
- Signature + signature = the signatures are merged together.

An include is associated with an optional `inclspec`, which can be to thin the provided modules and rename the provided and required modules of an include. In its simplest mode of use, an `inclspec` is a list of modules to be brought into scope, e.g. `include p (A, B)`. Requirements cannot be hidden, but they can be renamed to provide an implementation (or even to just reexport the requirement under another name.) If a requirement is not mentioned in an explicit requirements list, it is implicitly included (thus, `requires (Hole)` has only a purely documentary effect). It is not valid to rename a provision to a requirement, or a requirement to a provision.

```
pkgexports ::= inclspec
```

An export, symmetrically, specifies what modules a package will bring into scope if it is included without any `inclspec`. Any module which is omitted from an explicit export list is not exposed (however, like before, requirements cannot be hidden.)

When an explicit export list is omitted, you can calculate the provides and requires of a package as follows:

- A package provides any non-included modules and signatures. (It only provides an included module/signature if it is explicitly reexported.)

[1] Signatures are the backbone of the Backpack module system. A signature can be used to type-check client code which uses a module (without the module implementation), or to verify that an implementation upholds some signature (without a client implementation.)

- A package requires any transitively reachable signatures or hole signatures which are not filled in with an implementation.

ToDo: Properly describe "hole signatures" in the declarations section

4.1 Requirements

The fact that requirements are *implicitly* propagated from package to package can result in some spooky "action at a distance". However, this implicit behavior is one of the key ingredients to making mix-in modular development scale: you don't want to have to explicitly link everything up, as you might have to do in a traditional ML module system.

You cannot, however, import a requirement, unless it is also provided, which helps increase encapsulation. If a package provides a module, it can be imported:

```
package p (A) requires (A) where
    signature A where
        x :: Bool
package q (B) requires (A) where
    include p
    module B where
        import A    -- OK
```

If it does not, it cannot be imported: Alternately, the import is OK but doesn't result in any identifiers being brought into scope.

```
package p () requires (A) where -- yes, this is kind of pointless
    signature A where
        x :: Bool
package q (B) requires (A) where
    include p
    module B where
        import A    -- ERROR!
```

This means that it is always safe for a package to remove requirements or weaken holes; clients will always continue to compile.

Of course, if there is a different signature for the hole in scope, the import is not an error; however, no declarations from `p` are in scope:

```
package p () requires (A) where
    signature A where
        x :: Bool
package q (B) requires (A) where
    include p
    signature A where
        y :: Bool
    module B where
        import A
        x' = x      -- ERROR!
        y' = y      -- OK
```

To summarize, requirements are part of the interface of a package; however, they provide no identifiers as far as imports are concerned. There is some subtle interaction with requirements and shaping; see Shaping by example for more details.

4.2 Package includes/exports

A package export is easy enough to explain by analogy of module exports in Haskell: a `package p` in an export list explicitly reexports the identifiers from that package; whereas even a default, wildcard export list would only export locally defined identifiers/modules.

For example, this module exports the modules of both `base` and `array`.

```
package combined(package base, package array) where
    include base
    include array
```

However, in Backpack, a package may be included multiple times, making such declarations ambiguous. Thus, a package can be included as a local package name to disambiguate:

```
package p(package q1) where         -- equivalent to B1
    include impls (A1, A2)
    include q as q1 (hole A as A1, B as B1)
    include q as q2 (hole A as A2, B as B2)
```

A package include, e.g. `include a (package p)` is only valid if `a` exports the package `p` explicitly.[2]

5 (Transparent) signature ascription

```
inclspec ::= ...
           | "::" pkgexp

pkgexp ::= pkgname
         | "package" [exports] "where" pkgbody
```

Signature ascription subsumes thinning: it narrows the exports of modules in a package to those specified by a signature package. This package `pkgexp` is specified with either a reference to a named package or an *anonymous package* (in prior work, these have been referred to as *units*, although here the distinction is not necessary as our system is *purely applicative*).

Ascription also imposes a *requirement* on the package being abscribed. Suppose you have `p :: psig`, then:

- Everything provided `psig` must also be provided by `p`.
- Everything required by `p` must also be required by `psig`.

Alternately, the second requirement is not necessary, and you calculate the new requirements by taking the requirements of `psig`, removing the provides of `p`, and then adding the requirements of `p`. This makes it possible to ascribe includes for *adapter* packages, which provide some modules given a different set of requirements.

Semantically, ascription replaces the module with a signature, type-checks the package against the signature, and then *post facto* links the signature against the implementation. An ascribed include can be replaced with the signature it is ascribed with, resulting in a package which still typechecks but has more holes. You have to link at the VERY END, because if you link immediately after processing the module with the ascribed include, the module identities will leak. Of course, if we're compiling we just link eagerly. But now this means that if you have a definite package which uses ascription, even assuming all packages in the environment type-check, you must still type-check this package twice, once indefinitely and then with the actual linking relationship.

For example, ascription in the export specification thins out all private identifiers from the package:

[2] It's probably possible to use anonymous packages to allow easily dividing a package into subpackages, but this is silly and you can always just put it in an actual package.

```
package psig where
    signature A where
        public :: Bool
package p :: psig where
    module A.Internal where
        not_exported = 0
    module A where
        public = True
        private = False
```

and, symmetrically, ascription in an include hides identifiers:

```
package psig where
    signature A where
        public :: Bool
package p where
    module A where
        public = True
        private = False
package q where
    include p :: psig
    module B where
        import A
        ... public ...  -- OK
        ... private ... -- ERROR
```

OBSERVATION: thinning is subsumed by transparent signature ascription, but NOT renaming. Thus RENAMING does not commute across signature ascription; you must do it either before or after. Syntax for this is tricky.

Syntactic sugar for anonymous packages

```
pkgexp ::= pkgbody
         | path
```

It may be useful to provide two forms of sugar for specifying anonymous packages: `pkgbody` is equivalent to `package where pkgbody`; and `"path"` is equivalent to `package where source "path"`.

A Full grammar

```
packages ::= "{" package_0 ";" ... ";" package_n "}"

package ::= "package" pkgname [pkgexports] "where" pkgbody
pkgname ::= /* package name, e.g. containers (no version!) */
pkgbody ::= "{" pkgdecl_0 ";" ... ";" pkgdecl_n "}"

pkgdecl ::= "module"    modid [exports] "where" body
          | "signature" modid [exports] "where" body
          | "include"   pkgname ["as" pkgname] [inclspec]
          | "module"    modid_0 "=" path_0 "," ... "," modid_n "=" path_n
          | "signature" modid_0 "=" path_0 "," ... "," modid_n "=" path_n
          | "module"    modid_0 "," ... "," modid_n
```

```
          | "signature" modid_0 "," ... "," modid_n
          | "source" path

inclspec ::= "(" renaming_0 "," ... "," renaming_n ["," "] ")"
             [ "requires" "(" renaming_0 "," ... "," renaming_n ["," "] ")" ]
           | "::" pkgexp
pkgexports ::= inclspec

renaming ::= modid [ "as" modid ]
           | "package" pkgname

pkgexp ::= pkgname
         | "package" [exports] "where" pkgbody
         | pkgbody
         | path
```