The Backpack Manual

January 20, 2015

What is this? This is an in-depth technical specification of all of the new components associated with Backpack, a new module system for Haskell. This is *not* a tutorial, and it assumes you are familiar with the basic motivation and structure of Backpack.

How to read this manual This manual is split into three sections, in dependency order. The first section describes the new features added to GHC, e.g., new compilation flags and input formats. In principle, a user could take advantage of Backpack using just these features, without using Cabal or cabal-install; thus, we describe it first. The next section describes the new features added to the library Cabal, and the last section describes how cabal-install drives the entire process. A downside of this approach is that we start off by describing low-level GHC features which are quite dissimilar from the high-level Backpack interface, but we're not really trying to explain Backpack to complete new users. Red indicates features which are not implemented yet.

1 GHC

1.1 Signatures

An `hsig` file represents a (type) signature for a Haskell module, containing type signatures, data declarations, type classes, type class instances, but not value definitions.[1] The syntax of an `hsig` file is similar to an `hs-boot` file. 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
```

For entities that can be explicitly exported and imported, the export list of a module signature behaves in the same way as the export list for a normal module (e.g., if no list is provided, only entities defined in the signature are made available.)

However, type class instances and type family instances operate differently: an instance is *only* exported if it is directly defined in the signature. This is in contrast to the module behavior, where an instance is *implicitly* brought into scope if it is imported in any way (even with an empty import list.)

Even if an instance is "hidden" (i.e., not exported by a signature but in the implementation), we still take it into account when calculating conflicting instances (e.g., the soundness checks for type families). Thus, some compilation errors may only occur when linking an implementation and user, even if they compiled individually fine against the signature in question.

An `hsig` file can either be type-checked or compiled against some *backing implementation*, an `hs` module which provides all of the declarations that a signature advertises.

[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.)

Typechecking A signature file can be type-checked in the same way as an ordinary Haskell file:

```
$ ghc -c Map.hsig -fno-code -fwrite-interface
```

This procedure generates an interface file, which can be used to type check other modules which depend on the signature, even if no backing implementation is available. By default, this generated interface file is given *fresh* original names for everything in the signature. For example, if `data T` is defined in two signature files `A.hsig` and `B.hsig`, they would not be considered type-equal, and could not be used interconvertibly, even if they had the same structure.

To explicitly specify what original name should be assigned (e.g., to make the previous example type-equal) the `-shape-of` flag can be used:

```
$ ghc -c Map.hsig -shape-of "Map is containers_KEY:Data.Map.Map" \
    -fno-code -fwrite-interface
```

`-shape-of` is comma separated list of `name is orignname` entries, where `name` is an unqualified name and `orignname` is an original name, of the form `package_KEY:Module.name`, where `package_KEY` is a package key identifying the origin of the identifier (or a fake identifier for a symbol whose provenance is not known). Each instance of `orignname` in the signature is instead assigned the original name `orignname`, instead of the default original name.

(ToDo: This interface will work pretty poorly with `--make`)

Compiling We can specify a backing implementation for a signature and compile the signature against it using the `-sig-of` flag:

```
$ ghc -c Map.hsig -sig-of "package_KEY:Module"
```

The `-sig-of` flag takes as an argument a module, specified as a package key, a colon, and then the module name. This module must be a proper, `exposed-module`, and not a reexport or signature.

Compilation of a signature entails two things. First, a consistency check is performed between the signature and the backing implementation, ensuring that the implementation accurately implements all of the types in the signature. For every declaration in the signature, there must be an equivalent one in the backing implementation with an identical type (this check is quite similar to the one used for `hs-boot`). Second, an interface file is generated which reexports the set of identifiers from the backing implementation that were specified in the signature. A file which imports the signature will use this interface file.[2]

ToDo: In what cases is a type class instance/type family instance reexported? Currently, type classes from the backing implementation leak through. We also need to fix #9422.

1.2 Extended format in the installed package database

After a set of Haskell modules has been compiled, they can be registered as a package in the *installed package database* using `ghc-pkg`. An entry in the installed package database specifies what modules and signatures from the package itself are available for import. It can also re-export modules and signatures from other packages.[3]

There are three fields of an entry in the installed package database of note.

[2]This interface file is similar to a module which reexports identifiers from another module, except that we also record the backing implementation for the purpose of handling imports, described in the next section.

[3]Signature reexports are essential for creating signature packages in a modular way; module reexports are very useful for backwards-compatibility packages and also taking an package that has been instantiated multiple ways and giving its modules unique names.

exposed-modules A comma-separated list of module names which this package makes available for import, possibly with two extra, optional pieces of information about the module in question: what the *original module/signature* is (`from MODULE`)[4], and what the *backing implementation* is (`is MODULE`)[5].

```
exposed-modules:
    A,                              # original module
    B from ipid:B,                  # reexported module
    C is ipid:CImpl,                # exposed signature
    D from ipid:D is ipid:DImpl,    # reexported signature
    D from ipid:D2 is ipid:DImpl    # duplicates can be OK
```

If no reexports or signatures are used, the commas can be omitted (making this syntax backwards compatible with the original syntax.)[6]

instantiated-with A map from hole name to the *original module* which instantiated the hole (i.e., what `-sig-of` parameters were used during compilation.)

key The *package key* of a package, an opaque identifier identifying a package which serves as the basis for type identity and linker symbols.[7] When files are compiled as part of a package, the package key must be specified using the `-this-package-key` flag.[8]

The package key is programatically generated by Cabal[9]. While GHC doesn't specify what the format of the package key is, Cabal's must choose distinct package keys if any of the following fields in the installed package database are distinct:

- `name` (e.g., `containers`)
- `version` (e.g., `0.8`)
- `depends` (with respect to package keys)
- `instantiated-with` (with respect to package keys and module names)

1.3 Module thinning and renaming

The command line flag `-package pkgname` causes all exposed modules of `pkgname` (from the installed package database) to become visible under their original names for imports. The `-package` flag and its variants (`-package-id` and `-package-key`) support "thinning and renaming" annotations, which allows a user to selectively expose only certain modules from a package, possibly under different names.[10]

[4]Knowing the original module/signature makes it possible for GHC to directly load the interface file, without having to follow multiple hops in the package database.

[5]Knowing the backing implementation makes it possible to tell if an import is unambiguous without having to load the interface file first.

[6]Actually, the parser is a bit more lenient than that and can figure out commas when it's omitted. But it's better to just put commas in.

[7]Informally, you might think of a package as a package name and its version, e.g., `containers-0.9`; however, sometimes, it is necessary to distinguish between installed instances of a package with the same name and version which were compiled with different dependencies.

[8]The package key is different from an *installed package ID*, which is a more fine-grained identifier for a package. Identical installed package IDs imply identical package keys, but not vice versa. However, within a single run of GHC, we enforce that package keys and (non-shadowed) installed package IDs are in one-to-one correspondence.

[9]In practice, a package key looks something like `conta_GtvvBIboSRuDmyUQfSZoAx`. In this document, we'll use `containers_KEY` as a convenient shorthand to refer to some package key for the `containers` package.

[10]This feature has utility both with and without Backpack. The ability to rename modules makes it easier to deal with third-party packages which export conflicting module names; under Backpack, this situation becomes especially common when an indefinite package is instantiated multiple time with different dependencies.

Thinning and renaming can be applied using the extended syntax `-package "pkgname (rns)"`, where `rns` is a comma separated list of module renamings `OldName as NewName`. Bare module names are also accepted, where `Name` is shorthand for `Name as Name`. A package exposed this way only causes modules (specified before the `as`) explicitly listed in the renamings to become visible under their new names (specified after the `as`). For example, `-package "containers (Data.Set, Data.Map as Map)"` makes `Data.Set` and `Map` (pointing to `Data.Map`) available for import.[11]

When the `-hide-all-packages` flag is applied, uses of the `-package` flag are *cumulative*; each argument is processed and its bindings added to the global module map. For example, `-hide-all-packages -package containers -pac` brings both the default exposed modules of containers and a binding for `Map` into scope.[12][13]

1.4 Disambiguating imports

With module thinning and renaming, as well as the installed package database, it is possible for GHC to have multiple bindings for a single module name. If the bindings are ambiguous, GHC will report an error when the user attempts to use the identifier.

Define the *true module* associated with a binding to be the backing implementation, if the binding is for a signature,[14] and the original module otherwise. A binding is unambiguous if the true modules of all the bindings are equal. Here is an example of an unambiguous set of exposed modules:

```
exposed-modules:
    A from pkg:AImpl,
    A is pkg:AImpl,
    A from other-pkg:Sig is pkg:AImpl
```

This mapping says that this package reexports `pkg:AImpl` as `A`, has an `A.hsig` which was compiled against `pkg:AImpl`, and reexports a signature from `other-pkg` which itself was compiled against `pkg:AImpl`.

When Haskell code makes an import, we either load the backing implementation, if it is available as a direct reexport or original definition, or else load *all* of the interface files available as signatures. Loading all of the interfaces is guaranteed to not cause conflicts, as the backing implementation of all the signatures is guaranteed to be identical (assuming that it is unambiguous.)

Home package signatures In some circumstances, we may both define a signature in the home package, as well as import a signature with the same name from an external package. While multiple signatures from external packages are always merged together, in some cases, we will ignore the external package signature and *only* use the home package signature: in particular, if an external signature is not exposed from an explicit `-package` flag, it is not merged in.

Package imports A package import, e.g.,

```
import "foobar" Baz
```

operates as follows: ignore all exposed modules under the name which were not directly exposed by the package in question. If the same package name was included multiple times, all instances of it are considered (thus, package imports cannot be used to disambiguate between multiple versions or instantiations of the same package. For complex disambiguation, use thinning and renaming.)

In particular, package imports consider the *immediate* package which exposed a module, not the original package which defined the module.

[11] See also Cabal files for a twist on this syntax.

[12] The previous behavior, and the current behavior when `-hide-all-packages` is not set, is for a second package flag for the same package name to override the first one.

[13] We defer discussion of what happens when a module name is bound multiple times until we have discussed signatures, which have interesting behavior on this front.

[14] This implements signature merging, as otherwise, we would not necessarily expect original signatures to be equal

Typechecking When typechecking only, there is not necessarily a backing implementation associated with a signature. In this case, even if the original names match up, we must perform an *additional* check to ensure declarations have compatible types. This check is not necessary during compilation, because `-sig-of` will ensure that the signatures are compatible with a common, unique backing implementation.

User-interface A number of operations in the compiler accept a module name, and perform some operation assuming that, if the name successfully resolves, it will identify a unique module. In the presence of signatures, this assumption no longer holds. In this section, we describe how to adjust the behavior of these various operations:

- `ghc --abi-hash M` fails if `M` resolves to multiple signatures. Same rules for home/external package resolution apply, so in the absence of any other flags we will hash the signature interface in the home package.
-

1.5 Indefinite external packages

Not implemented yet.

2 Backpack

This entire section is a proposed and has not been implemented.

In this section, we describe an expanded version of the package language described in the Backpack paper which GHC accepts as input. Given a *Backpack file*, GHC performs shaping analysis, typechecking, compilation and registration of multiple packages (whose source code is specified by the Backpack file). A Backpack file replaces use of `-shape-of`, `-sig-of` and `-package` flags.[15][16]

Here is a very simple Backpack file which defines two packages:[17]

```
package a
    exposed-modules:     A

package b
    includes:            a
    exposed-modules:     B
```

Here is a more complicated Backpack file taking advantage of the availability of signatures in Backpack:

```
installed package base
    installed-id:        base-4.0.6-0123456789abcdef

package p
    includes:            base
```

[15] Backpack files are *generated* by Cabal. Cabal is responsible for downloading source files, resolving what versions of packages are to be used, executing conditional statements. Once the Cabal files are compiled into a Backpack file, it is passed to GHC, which is responsible for instantiating holes and compiling the packages. The package descriptions in a Backpack file are not full Cabal packages, but contain the minimum information necessary for GHC to work: they are more akin to entries in the installed package database (with some differences).

[16] One design goal of this separate package language from Cabal is that it can more easily be incorporated into a language specification, without needing the specification to pull in a full description of Cabal.

[17] It could have been written as two separate files: the effect of processing this Backpack file is to compile both packages simultaneously.

```
    exposed-modules:     P
    other-modules:       InternalsP
    required-signatures: Map
    source-dir:          /srv/code/p
    installed-id:        p-2.0.1-abcdef0123456789
                         p-2.0.1-def0123456789abc

package q
    includes:            base, p (Map as QMap)
    exposed-modules:     Q
    other-modules:       QMap
    source-dir:          /srv/code/q
```

A Backpack file consists of a list of named packages, each of which is composed of fields (similar to fields in Cabal package description) which specify various aspects of the package. A package may optionally be an *installed* package (specified by the `installed` keyword), in which case the package refers to an existing package (with no holes) in the installed package database; in this case, all fields are omitted except for `installed-id`, which identifies the specific package in use.

All packages in a Backpack file live in the global namespace. A possible future addition would be the ability to specify private packages which are not exposed.

```
backpack ::= package_0
             ...
             package_n

package ::= "package" pkgname
               field_0
               ...
               field_n
          | "installed package" pkgname
               "installed-id:" ipid

pkgname ::= /* package name, e.g. containers (no version!) */

field ::= "includes:"            includes
        | "exposed-modules:"     modnames
        | "other-modules:"       modnames
        | "exposed-signatures:"  modnames
        | "required-signatures:" modnames
        | "reexported-modules:"  reexports
        | "source-dir:"          path
        | pkgdb_field
```

We now describe the package fields in more detail.

2.1 includes

```
includes ::= include_0 "," ... "," include_n
include ::= pkgname ["(" renames ")"]

renames ::= rename_0 "," ... "," rename_n
rename ::= modname
         | modname "as" modname
```

The `includes` field consists of a comma-separated list of packages to include. This field is similar to the Cabal `build-depends` field, except that no version numbers are allowed. Each package has all exposed modules and signatures are brought into scope under their original names, unless there is a parenthesized, comma-separated thinning and renaming specification which causes only the specified modules are brought into scope (under new names, if the `as` keyword is used). Here is an example `includes` field, which brings into scope all exposed modules from `base`, `P1` and `P2` from `p`, and `Q` from `q` under the name `MyQ`.

```
includes: base, p (P1, P2), q (Q as MyQ)
```

Package inclusion is the mechanism by which holes are instantiated: a hole and an implementation which are brought in the same scope with the same name are linked together. If a package is included multiple times, it is treated as a separate instantiation for the purpose of filling holes.

2.2 exposed-modules, other-modules, exposed-signatures, required-signatures

```
modnames ::= modname_0 ... modname_n
```

The `exposed-modules`, `other-modules`, `exposed-signatures` and `required-signatures` are exactly analogous to their Cabal counterparts, and consist of lists of module names which are to be compiled from the package's source directory. For example, to expose modules `P` and `Q`, you would write:

```
exposed-modules: P Q
```

2.3 reexported-modules

```
reexports ::= reexport_0 "," ... "," reexport_n
reexport  ::= modname "as" modname
            | modname
```

The `reexported-modules` field is exactly analogous to its Cabal counterpart, and allows reexporting an in-scope module under a different name.[18] For example, to reexport a locally available module `P` under the name `Q`, write:

```
reexported-modules: P as Q
```

2.4 source-dir

```
path ::=  /* file path, e.g. /home/alice/src/containers */
```

The `source-dir` field specifies where the source files of the package in question live, e.g. if `source-dir: /foo` then we expect the `hs` file for module `A` to live in `/foo/A.hs`.

2.5 installed-id

```
ipid ::= /* installed package ID, e.g. containers-0.8-HASH */
```

The `installed-id` field specifies an existing, *compiled* package in the installed package database, which should be used. This information is only used in the case of an `installed package` entry, because we would otherwise not have enough information to calculate a package key for the package. It is analogous to the `-package-id` flag.

This is enough if, in a package database, a given package key is unique. If package keys are not unique, it might also be necessary to explicitly provide multiple multiple `installed-id`s for an indefinite package, corresponding to valid compilations of the package with different hole instantiations. This never happens with current Cabal, since version numbers are built into package keys.

[18] This is different from *aliasing* in the original Backpack language, since reexported modules are not visible in the current package.

2.6 Installed package database fields

```
pkgdb_field ::= ...
```

GHC's installed package database supports a number of other fields which are necessary for GHC to build some packages, e.g., the `extraLibraries` field which specifies operating system libraries which also have to be linked in. Backpack packages accept any fields which are valid in the installed package database, except for: `name`, `id`, `key` and `instantiated-with` (which are computed by GHC itself). The full list of available fields can be found in the `bin-package-db` package.

2.7 Structure of a Backpack file

In general, a Backpack file must contain the package descriptions of *all* packages which are transitively depended on (in case one of those packages must be rebuilt.) However, if we know a specific version of a package is already in the installed package database, its description may be replaced with an `installed package` entry, in which case the description (and description of its dependencies) can be omitted. An alternative is to have an indefinite package database, in which case this database is simply always in scope. This might be better if we want to save interface files associated with indefinite packages.

It should be emphasized that while the Backpack file leaves the instantiation of holes implicit (to be resolved by looking at the included packages and linking modules together), *all package versions* must be resolved prior to writing a Backpack file. A Backpack file assumes that the versions of all packages are consistent (e.g., any reference to `foo` will always be a reference to `foo-1.2`).

3 Cabal

3.1 Fields in the Cabal file

The Cabal file is a user-facing description of a package, which is converted into an `InstalledPackageInfo` during a Cabal build. Backpack extends the Cabal files with four new fields, all of which are only valid in the `library` section of a package:

required-signatures A space-separated list of module names specifying internal signatures (in `hsig` files) of the package. Signatures specified in this field are not put in the `exposed-modules` field in the installed package database and are not available for external import; however, in order for a package to be compiled, implementations for all of its signatures must be provided (so they are not completely *hidden* in the same way `other-modules` are).

exposed-signatures A space-separated list of module names specifying externally visible signatures (in `hsig` files) of the package. It is represented in the installed package database as an `exposed-module` with a non-empty backing implementation (`Sig is Impl`). Signatures exposed in this way are available for external import. In order for a package to be compiled, implementations for all exposed signatures must be provided.

indefinite A package is *indefinite* if it has any uninstantiated `required-signatures` or `exposed-signatures`, or it depends on an indefinite package without instantiating all of the holes of that package. In principle, this parameter can be calculated by Cabal, but it serves a documentory purpose for packages which do not have any signatures themselves, but depend on packages which are indefinite. Actually, this field is in the top-level at the moment.

reexported-modules A comma-separated list of module or signature reexports. It is represented in the installed package database as a module with a non-empty original module/signature: the original module is resolved by Cabal. There are three valid syntactic forms:

- `Orig`, which reexports any module with the name `Orig` in the current scope (e.g., as specified by `build-depends`).
- `Orig as New`, which reexports a module with the name `Orig` in the current scope. `Orig` can be a home module and doesn't necessarily have to come from `build-depends`.
- `package:Orig as New`, which reexports a module with name `Orig` from the specific source package `package`.

If multiple modules with the same name are in scope, we check if it is unambiguous (the same check used by GHC); if they are we reexport all of the modules; otherwise, we give an error. In this way, packages which reexport multiple signatures to the same name can be valid; a package may also reexport a signature onto a home `hsig` signature.

3.2 build-depends

This field has been extended with new syntax to provide the access to GHC's new thinning and renaming functionality and to have the ability to include an indefinite package *multiple times* (with different instantiations for its holes).

Here is an example entry in `build-depends`: `foo >= 0.8 (ASig as A1, B as B1; ASig as A2, ...)`. This statement includes the package `foo` twice, once with `ASig` instantiated with `A1` and `B` renamed as `B1`, and once with `ASig` instantiated with `A2`, and all other modules imported with their original names. Assuming that the key of the first instance of `foo` is `foo_KEY1` and the key of the second instance is `foo_KEY2`, and that `ASig` is an `exposed-signature`, then this `build-depends` would turn into these flags for GHC: `-package-key "foo\_KEY1 (ASig as A1, B as B1)" -package-key "foo\_KEY2" -package-key "foo\_KEY2 (ASig as`

Syntactically, the thinnings and renamings are placed inside a parenthetical after the package name and version constraints. Semicolons distinguish separate inclusions of the package, and the inner comma-separated lists indicate the thinning/renamings of the module. You can also write `...`, which simply includes all of the default bindings from the package. This is not implemented. Should this only refer to modules which were not referred to already? Should it refer only to holes?

There are two remarks that should be made about separate instantiations of the package. First, Cabal will automatically "de-duplicate" instances of the package which are equivalent: thus, `foo (A; B)` is equivalent to `foo (A, B)` when `foo` is a definite package, or when the holes instantiation for each instance is equivalent. Second, when merging two `build-depends` statements together (for example, due to a conditional section in a Cabal file), they are considered *separate inclusions of a package.*

3.3 Setup flags

There is one new flag for the `Setup` script, which can be used to manually provide instantiations for holes in a package: `--instantiate-with NAME=PKG:MOD`, which binds a module `NAME` to the implementation `MOD` provided by installed package ID `PKG`. The flag can be specified multiply times to provide bindings for all signatures. The module in question must be the *original* module, not a re-export.

3.4 Metadata in the installed package database

Cabal records

`instantiated-with`

4 cabal-install

4.1 Indefinite package instantiation