Implementing Backpack

Monday 23rd March, 2015

The purpose of this document is to describe an implementation path for Backpack in GHC.

Contents

1 What we are trying to solve

While the current ecosystem has proved itself serviceable for many years, there are a number of major problems which causes significant headaches for many users. Here are some of them:

1.1 Package reinstalls are destructive

When attempting to install a new package, you might get an error like this:

```
$ cabal install hakyll
cabal: The following packages are likely to be broken by the reinstalls:
pandoc-1.9.4.5
Graphalyze-0.14.0.0
Use --force-reinstalls if you want to install anyway.
```

While this error message is understandable if you're really trying to reinstall a package, it is quite surprising that it can occur even if you didn't ask for any reinstalls!

The underlying cause of this problem is related to an invariant Cabal currently enforces on a package database: there can only be one instance of a package for any given package name and version. This means that it is not possible to install a package multiple times, compiled against different dependencies. However, sometimes, reinstalling a package with different dependencies is the only way to fulfill version bounds of a package! For example: say we have three packages `a`, `b` and `c`. `b-1.0` is the only version of `b` available, and it has been installed and compiled against `c-1.0`. Later, the user installs an updated version `c-1.1` and then attempts to install `a`, which depends on the specific versions `c-1.1` and `b-1.0`. We *cannot* use the already installed version of `b-1.0`, which depends on the wrong version of `c`, so our only choice is to reinstall `b-1.0` compiled against `c-1.1`. This will break any packages, e.g. `d`, which were built against the old version of `b-1.0`.

Our solution to this problem is to *abolish* destructive package installs, and allow a package to be installed multiple times with the same package name and version. However, allowing this poses some interesting user interface problems, since package IDs are now no longer unambiguous identifiers.

1.2 Version bounds are often over/under-constrained

When attempting to install a new package, Cabal might fail in this way:

```
$ cabal install hledger-0.18
Resolving dependencies...
cabal: Could not resolve dependencies:
# pile of output
```

There are a number of possible reasons why this could occur, but usually it's because some of the packages involved have over-constrained version bounds, which are resulting in an unsatisfiable set of constraints (or, at least, Cabal gave up backtracking before it found a solution.) To add insult to injury, most of the time the bound is nonsense and removing it would result in a working compilation. In fact, this situation is so common that Cabal has a flag `--allow-newer` which lets you override the package upper bounds.

However, the flip-side is when Cabal finds a satisfying set, but your compilation fails with a type error. Here, you had an under-constrained set of version bounds which didn't actually reflect the compatible versions of a package, and Cabal picked a version of the package which was incompatible.

Our solution to this problem is to use signatures instead of version numbers as the primary mechanism by which compatibility is determined: e.g., if it typechecks, it's a valid choice. Version numbers can still be used to reflect semantic changes not seen in the types (in particular, ruling out buggy versions of a package is a useful operation), but these bounds are empirical observations and can be collected after-the-fact.

1.3 It is difficult to support multiple implementations of a type

This problem is perhaps best described by referring to a particular instance of it Haskell's ecosystem: the `String` data type. Haskell, by default, implements strings as linked lists of integers (representing characters). Many libraries use `String`, because it's very convenient to program against. However, this representation is also very *slow*, so there are alternative implementations such as `Text` which implement efficient, UTF-8 encoded packed byte arrays.

Now, suppose you are writing a library and you don't care if the user of your library is using `String` or `Text`. However, you don't want to rewrite your library twice to support both data types: rather, you'd like to rely on some *common interface* between the two types, and let the user instantiate the implementation. The only way to do this in today's Haskell is using type classes; however, this necessitates rewriting all type signatures from a nice `String -> String` to `StringLike s => s -> s`. The result is less readable, required a large number of trivial edits to type signatures, and might even be less efficient, if GHC does not appropriately specialize your code written in this style.

Our solution to this problem is to introduce a new mechanism of pluggability: module holes, which let us use types and functions from a module `Data.String` as before, but defer choosing *what* module should be used in the implementation to some later point (or instantiate the code multiple times with different choices.)

1.4 Fast moving APIs are difficult to develop/develop against

Most packages that are uploaded to Hackage have package authors which pay some amount of attention to backwards compatibility and avoid making egregious breaking changes. However, a package like the `ghc-api` follows a very different model: the library is a treated by its developers as an internal component of an application (GHC), and is frequently refactored in a way that changes its outwards facing interface.

Arguably, an application like GHC should design a stable API and maintain backwards compatibility against it. However, this is a lot of work (including refactoring) which is only being done slowly, and in the meantime, the dump of all the modules gives users the functionality they want (even if it keeps breaking every version.)

One could say that the core problem is there is no way for users to easily communicate to GHC authors what parts of the API they rely on. A developer of GHC who is refactoring an interface will often rely on the typechecker to let them know which parts of the codebase they need to follow and update, and often could say precisely how to update code to use the new interface. User applications, which live out of tree, don't receive this level of attention.

Our solution is to make it possible to typecheck the GHC API against a signature. Important consumers can publish what subsets of the GHC API they rely against, and developers of GHC, as part of their normal build process, type-check against these signatures. If the signature breaks, a developer can either do the refactoring differently to avoid the compatibility-break, or document how to update code to use the new API.

2 Backpack in a nutshell

For a more in-depth tutorial about Backpack's features, check out Section 2 of the original Backpack paper. In this section, we briefly review the most important points of Backpack's design.

Thinning and renaming at the module level A user can specify a build dependency which only exposes a subset of modules (possibly under different names.) By itself, it's a way for the user to resolve ambiguous module imports at the package level, without having to use the `PackageImports` syntax extension.

Holes (abstract module definitions) The core component of Backpack's support for *separate modular development* is the ability to specify abstract module bindings, or holes, which give users of the module an obligation to provide an implementation which fulfills the signature of the hole. In this example:

```
package p where
    A :: [ ... ]
    B = [ import A; ... ]
```

`p` is an *indefinite package*, which cannot be compiled until an implementation of A is provided. However, we can still type check B without any implementation of A, by type checking it against the signature. Holes can be put into signature packages and included (depended upon) by other packages to reuse definitions of signatures.

Filling in holes with an implementation A hole in an indefinite package can be instantiated in a *mix-in* style: namely, if a signature and an implementation have the same name, they are linked together:

```
package q where
    A = [ ... ]
    include p -- has signature A
```

Renaming is often useful to rename a module (or a hole) so that a signature and implementation have the same name and are linked together. An indefinite package can be instantiated multiple times with different implementations: the *applicativity* of Backpack means that if a package is instantiated separately with the same module, the results are type equal:

```
package q' where
    A = [ ... ]
    include p (A, B as B1)
    include p (A, B as B2)
    -- B1 and B2 are equivalent
```

Combining signatures together Unlike implementations, it's valid for a multiple signatures with the same name to be in scope.

```
package a-sig where
    A :: [ ... ]
package a-sig2 where
    A :: [ ... ]
package q where
    include a-sig
    include a-sig2
    B = [ import A; ... ]
```

These signatures *merge* together, providing the union of the functionality (assuming the types of individual entities are compatible.) Backpack has a very simple merging algorithm: types must match exactly to be compatible (*width* subtyping).

3 Module and package identity

```
package p where
    A :: [ data X ]
    P = [ import A; data Y = Y X ]
package q where
    A1 = [ data X = X1 ]
    A2 = [ data X = X2 ]
    include p (A as A1, P as P1)
    include p (A as A2, P as P2)
```

(a) Type equality must consider holes. . .

```
package p where
    A :: [ data X ]
    P = [ data T = T ] -- no A import!
package q where
    A1 = [ data X = X1 ]
    A2 = [ data X = X2 ]
    include p (A as A1, P as P1)
    include p (A as A2, P as P2)
```

(b) . . . but how do we track dependencies?

Figure 1: Two similar examples

One of the central questions one encounters when type checking Haskell code is: *when are two types equal*? In ordinary Haskell, the answer is simple: "They are equal if their *original names* (i.e., where they were originally defined) are the same." However, in Backpack, the situation is murkier due to the presence of *holes*. Consider the pair of examples in Figure 1. In Figure 1a, the types `B1.Y` and `B2.Y` should not be considered equal, even though naïvely their original names are `p`:`B.Y`, since their arguments are different `X`'s! On the other hand, if we instantiated `p` twice with the same `A` (e.g., change the second include to `include p (A as A1, P as P2)`), we might consider the two resulting `Y`'s equal, an *applicative* semantics of identity instantiation. In Figure 1b, we see that even though `A` was instantiated differently, we might reasonably wonder if `T` should still be considered the same, since it has no dependence on the actual choice of `A`.

In fact, there are quite a few different choices that can be made here. Figures 2 and 3 summarize the various choices on two axes: the granularity of applicativity (under what circumstances do we consider two types equal) and the granularity of dependency (what circumstances do we consider two types not equal)? A ✔ means the design we have chosen answers the question affirmatively—✖, negatively—but all of these choices are valid points on the design space.

3.1 The granularity of applicativity

An applicative semantics of package instantiation states that if a package is instantiated with the "same arguments", then the resulting entities it defines should also be considered equal. Because Backpack uses *mix-in modules*, it is very natural to consider the arguments of a package instantiation as the modules, as shown in Figure 2b: the same module `A` is linked for both instantiations, so `P1` and `P2` are considered equal.

However, we consider the situation at a finer granularity, we might say, "Well, for a declaration `data Y = Y X`, only the definition of type `X` matters. If they are the same, then `Y` is the same." In that case, we might accept that in Figure 2a, even though `p` is instantiated with different modules, at the end of the day, the important component `X` is the same in both cases, so `Y` should also be the same. This is a sort of "extreme" view of modular development, where every declaration is desugared into a separate module. In our design, we will be a bit more conservative, and continue with module level applicativity, in the same manner as Paper Backpack.

Implementation considerations Compiling Figure 2b to dynamic libraries poses an interesting challenge, if every package compiles to a dynamic library. When we compile package `q`, the libraries we end up producing are `q` and an instance of `p` (instantiated with `q`:`A`). Furthermore, `q` refers to code in `p` (the import in `Q`), and vice versa (the usage of the instantiated hole `A`). When building static libraries, this circular dependency doesn't matter: when we link the executable, we can resolve all of the symbols in one

```
package q where
  A = [ data X = X ]
  A1 = [ import A; x = 0 ]
  A2 = [ import A; x = 1 ]
  include p (A as A1, P as P1)
  include p (A as A2, P as P2)
  Q = [ import P1; ... ]
```

(a) Declaration applicativity ✖

```
package q where
  A = [ data X = X ]


  include p (A, P as P1)
  include p (A, P as P2)
  Q = [ import P1; ... ]
```

(b) Module applicativity ✔

```
package a where
  A = [ data X = X ]
package q where
  include a
  include p (A, P as P1)
  include p (A, P as P2)
  Q = [ import P1; ... ]
```

(c) Package applicativity ✔

Figure 2: Choices of granularity of applicativity on p: given data Y = Y X, is P1.Y equal to P2.Y?

```
package p(A,P) where
  A :: [ data X ]
  P = [
    import A
    data T = T
    data Y = Y X
  ]
```

(a) Declaration granularity ✖

```
package p(A,P) where
  A :: [ data X ]
  B = [ data T = T ]
  C = [
    import A
    data Y = Y X
  ]
  P = [
    import B
    import C
  ]
```

(b) Module granularity ✖

```
package b where
  B = [ data T = T ]
package c where
  A :: [ data X ]
  C = [
    import A
    data Y = Y X
  ]
package p(A,P) where
  include b; include c
  P = [ import B; import C ]
```

(c) Package granularity ✔

Figure 3: Choices of granularity for dependency: is the identity of T independent of how A is instantiated?

go. However, when the libraries in question are dynamic libraries `libHSq.so` and `libHSp(q:A).so`, we now have a *circular dependency* between the two dynamic libraries, and most dynamic linkers will not be able to load either of these libraries.

To break the circularity in Figure 2b, we have to *inline* the entire module A into the instance of p. Since the code is exactly the same, we can still consider the instance of A in q and in p type equal. However, in Figure 2c, applicativity has been done at a coarser level: although we are using Backpack's module mixin syntax, morally, this example is filling in the holes with the *package* a (rather than a module). In this case, we can achieve code sharing, since p can refer directly to a, breaking the circularity.

3.2 The granularity of dependency

In the previous section, we considered *what* entities may be considered for computing dependency; in this section we consider *which* entities are actually considered as part of the dependencies for the declaration/module/package we're writing. Figure 3 contains a series of examples which exemplify the choice of whether or not to collect dependencies on a per-declaration, per-module or per-package basis:

- Package-level granularity states that the modules in a package are considered to depend on *all* of the holes in the package, even if the hole is never imported. Figure 3c is factored so that T is defined in a distinct package b with no holes, so no matter the choice of A, B.T will be the same. On the other hand, in Figure 3b, there is a hole in the package defining B, so the identity of T will depend on the choice of A.

- Module-level granularity states that each module has its own dependency, computed by looking at its import statements. In this setting, `T` in Figure 3b is independent of `A`, since the hole is never imported in `B`. But once again, in Figure 3a, there is an import in the module defining `T`, so the identity of `T` once again depends on the choice of `A`.

- Finally, at the finest level of granularity, one could chop up `p` in Figure 3a, looking at the type declaration-level dependency to suss out whether or not `T` depends on `A`. It doesn't refer to anything in `A`, so it is always considered the same.

It is well worth noting that the system described by Paper Backpack tracks dependencies per module; however, we have decided that we will implement tracking per package instead: a coarser grained granularity which accepts less programs.

Is a finer form of granularity *better?* Not necessarily! For one, we can always split packages into further subpackages (as was done in Figure 3c) which better reflect the internal hole dependencies, so it is always possible to rewrite a program to make it typecheck—just with more packages. Additionally, the finer the granularity of dependency, the more work I have to do to understand what the identities of entities in a module are. In Paper Backpack, I have to understand the imports of all modules in a package; with declaration-granularity, I have to understand the entire code. This is a lot of work for the developer to think about; a more granular model is easier to remember and reason about. Finally, package-level granularity is much easier to implement, as it preserves the previous compilation model, *one library per package.* At a fine level of granularity, we may end up repeatedly compiling a module which actually should be considered "the same" as any other instance of it.

Nevertheless, finer granularity can be desirable from an end-user perspective. Usually, these circumstances arise when library-writers are forced to split their components into many separate packages, when they would much rather have written a single package. For example, if I define a data type in my library, and would like to define a `Lens` instance for it, I would create a new package just for the instance, in order to avoid saddling users who aren't interested in lenses with an extra dependency. Another example is test suites, which have dependencies on various test frameworks that a user won't care about if they are not planning on testing the code. (Cabal has a special case for this, allowing the user to write effectively multiple packages in a single Cabal file.)

3.3 Summary

We can summarize all of the various schemes by describing the internal data types that would be defined by GHC under each regime. First, we have the shared data structures, which correspond closely to what users are used to seeing:

```
<pkg-name>    ::= containers, ...
<pkg-version ::= 1.0, ...
<pkg-id>      ::= <pkg-name>-<pkg-version>
<mod-name>    ::= Data.Set, ...
<occ>         ::= empty, ...
```

Changing the **granularity of applicativity** modifies how we represent the list of dependencies associated with an entity. With module applicativity, we list module identities (not yet defined); with declaration applicativity we actually list the original names (i.e., ids).

```
<deps>        ::= <id>, ...       # Declaration applicativity
<deps>        ::= <module>, ...  # Module applicativity
```

Changing the **granularity of dependency** affects how we compute the lists of dependencies, and what entities are well defined:

```
# Package-level granularity
<pkg-key>    ::= hash(<pkg-id> + <deps for pkg>)
<module>     ::= <pkg-key> : <mod-name>
<id>         ::= <module> . <occ>

# Module-level granularity
<pkg-key>    not defined
<module>     ::= hash(<pkg-id> : <mod-name> + <deps for mod>)
<id>         ::= <module-key> . <occ>

# Declaration-level granularity
<pkg-key>    not defined
<module>     not defined
<id>         ::= hash(<pkg-id> : <mod-name> . <occ> + <deps for decl>)
```

Notice that as we increase the granularity, the notion of a "package" and a "module" become undefined. This is because, for example, with module-level granularity, a single "package" may result in several modules, each of which have different sets of dependencies. It doesn't make much sense to refer to the package as a monolithic entity, because the point of splitting up the dependencies was so that if a user relies only on a single module, it has a correspondingly restricted set of dependencies.

3.4 The new scheme, formally

In this section, we give a formal treatment of our choice in the design space, in the same style as the Backpack paper, but omitting mutual recursion, as it follows straightforwardly. Physical module identities ν, the `Module` component of *original names* in GHC, are either (1) *variables* α, which are used to represent holes[1] or (2) a concrete module p defined in package P, with holes instantiated with other module identities (might be empty)[2].

Package Names (`PkgName`)	$P \in \mathit{PkgNames}$
Module Path Names (`ModName`)	$p \in \mathit{ModPaths}$
Module Identity Vars	$\alpha, \beta \in \mathit{IdentVars}$
Package Key (`PackageKey`)	$\mathcal{K} ::= P(\overline{p \mapsto \nu})$
Module Identities (`Module`)	$\nu ::= \alpha \mid \mathcal{K}{:}p$
Module Identity Substs	$\phi, \theta ::= \{\overline{\alpha := \nu}\}$

Figure 4: Module Identities

As in traditional Haskell, every package contains a number of module files at some module path p; within a package these paths are guaranteed to be unique.[3] When we write inline module definitions, we assume that they are immediately assigned to a module path p which is incorporated into their identity. A module identity ν simply augments this with subterms $\overline{p \mapsto \nu}$ representing how *all* holes in the package P were instantiated.[4] This naming is stable because the current Backpack surface syntax does not allow a logical path in a package to be undefined. A package key is $P(\overline{p \mapsto \nu})$.

Here is the very first example from Section 2 of the original Backpack paper, ab-1:

```
package ab-1 where
  A = [x = True]
  B = [import A; y = not x]
```

[1]In practice, these will just be fresh paths in a special package key for variables.

[2]In Paper Backpack, we would refer to just $P{:}p$ as the identity constructor. However, we've written the subterms specifically next to P to highlight the semantic difference of these terms.

[3]In Paper Backpack, the module expressions themselves are used to refer to globally unique identifiers for each literal. This makes the metatheory simpler, but for implementation purposes it is convenient to conflate the *original* module path that a module is defined at with its physical identity.

[4]In Paper Backpack, we do not distinguish between holes/non-holes, and we consider all imports of the *module*, not the package.

The identities of A and B are ab-1:A and ab-1:B, respectively.[5] In a package with holes, each hole (within the package definition) gets a fresh variable as its identity, and all of the holes associated with package P are recorded. Consider abcd-holes-1:

```
package abcd-holes-1 where
  A :: [x :: Bool]
  B :: [y :: Bool]
  C  = [x = False]
       ⎡import qualified A⎤
  D  = ⎢import qualified C⎥
       ⎣z = A.x && C.x    ⎦
```

The identities of the four modules are, in order, α_a, α_b, abcd-holes-1(α_a, α_b):C, and abcd-holes-1(α_a, α_b):D.[6] We include both α_a and α_b in both C and D, regardless of the imports. When we link the package against an implementation of the hole, these variables are replaced with the identities of the modules we linked against.

Shaping proceeds in the same way as in Paper Backpack, except that the shaping judgment must also accept the package key $P(\overline{p \mapsto \alpha})$ so we can create identifiers with mkident. This implies we must know ahead of time what the holes of a package are.

A full Backpack comparison If you're curious about how the rest of the Backpack examples translate, look no further than this section.

First, consider the module identities in the Graph instantiations in multinst, shown in Figure 2 of the original Backpack paper. In the definition of structures, assume that the variables for Prelude and Array are α_P and α_A respectively. The identity of Graph is structures(α_P, α_A):Graph. Similarly, the identities of the two array implementations are $\nu_{AA} =$ arrays-a(α_P):Array and $\nu_{AB} =$ arrays-b(α_P):Array.[7]

The package graph-a is more interesting because it *links* the packages arrays-a and structures together, with the implementation of Array from arrays-a *instantiating* the hole Array from structures. This linking is reflected in the identity of the Graph module in graph-a: whereas in structures it was $\nu_G =$ structures(α_P, α_A):Graph, in graph-a it is $\nu_{GA} = \nu_G[\nu_{AA}/\alpha_A] =$ structures(α_P, ν_{AA}):Graph. Similarly, the identity of Graph in graph-b is $\nu_{GB} = \nu_G[\nu_{AB}/\alpha_A] =$ structures(α_P, ν_{AB}):Graph. Thus, linking consists of substituting the variable identity of a hole by the concrete identity of the module filling that hole.

Lastly, multinst makes use of both of these Graph modules, under the aliases GA and GB, respectively. Consequently, in the Client module, `GA.G` and `GB.G` will be correctly viewed as distinct types since they originate in modules with distinct identities.

As multinst illustrates, module identities effectively encode dependency graphs at the package level.[8] Like in Paper Backpack, we have an *applicative* semantics of instantiation, and the applicativity example in Figure 3 of the Backpack paper still type checks. However, because we are operating at a coarser granularity, modules may have spurious dependencies on holes that they don't actually depend on, which means less type equalities may hold.

3.5 Cabal dependency resolution

Currently, when we compile a Cabal package, Cabal goes ahead and resolves `build-depends` entries with actual implementations, which we compile against. The package key, independently of Backpack, records the transitive dependency tree selected during this dependency resolution process, so that we can install libfoo-1.0 twice compiled against different versions of its dependencies. What is the relationship to this

[5]In Paper Backpack, the identity for B records its import of A, but since it is definite, this is strictly redundant.

[6]In Paper Backpack, the granularity is at the module level, so the subterms of C and D can differ.

[7]Notice that the subterms coincide with Paper Backpack! A sign that module level granularity is not necessary for many use-cases.

[8]In Paper Backpack, module identities encode dependency graphs at the module level. In both cases, however, what is being depended on is always a module.

transitive dependency tree of *packages*, with the subterms of our package identities which are *modules*? Does one subsume the other? In fact, these are separate mechanisms—two levels of indirections, so to speak.

To illustrate, suppose I write a Cabal file with `build-depends: foobar`. A reasonable assumption is that this translates into a Backpack package which has `include foobar`. However, this is not actually a Paper Backpack package: Cabal's dependency solver has to rewrite all of these package references into versioned references `include foobar-0.1`. For example, this is a pre-package:

```
package foo where
    include bar
```

and this is a Paper Backpack package:

```
package foo-0.3[bar-0.1[baz-0.2]] where
    include bar-0.1[baz-0.2]
```

This tree is very similar to the one tracking dependencies for holes, but we must record this tree *even* when our package has no holes.

Linker symbols As we increase the amount of information in PackageId, it's important to be careful about the length of these IDs, as they are used for exported linker symbols (e.g. `base_TextziReadziLex_zdwvalDig_info`). Very long symbol names hurt compile and link time, object file sizes, GHCi startup time, dynamic linking, and make gdb hard to use. As such, we've done away with full package names and versions; instead, there is simply a base-62 encoded hash, with the first five characters of the package name for user-friendliness.

3.6 Package selection

When I fire up `ghci` with no arguments, GHC somehow creates out of thin air some consistent set of packages, whose modules I can load using `:m`. This functionality is extremely handy for exploratory work, but actually GHC has to work quite hard in order to generate this set of packages, the contents of which are all dumped into a global namespace. For example, GHC doesn't have access to Cabal's dependency solver, nor does it know *which* packages the user is going to ask for, so it can't just run a constraint solver, get a set of consistent packages to offer and provide them to the user.[9]

To make matters worse, while in the current design of the package database, a package is uniquely identified by its package name and version, in the Backpack design, it is *mandatory* that we support multiple packages installed in the database with the same package name and version, and this can result in complications in the user model. This further complicates GHC's default package selection algorithm.

In this section, we describe how the current algorithm operates (including what invariants it tries to uphold and where it goes wrong), and how to replace the algorithm to handle generalization to multiple instances in the package database. We'll also try to tease apart the relationship between package keys and installed package IDs in the database.

The current algorithm Abstractly, GHC's current package selection algorithm operates as follows. For every package name, select the package with the latest version (recall that this is unique) which is also *valid*. A package is valid if:

- It exists in the package database,
- All of its dependencies are valid,

[9]Some might argue that depending on a global environment in this fashion is wrong, because when you perform a build in this way, you have absolutely no ideas what dependencies you actually ended up using. But the fact remains that for end users, this functionality is very useful.

- It is not shadowed by a package with the same package ID[10] in another package database (unless it is in the transitive closure of a package named by `-package-id`), and
- It is not ignored with `-ignore-package`.

Package validity is probably the minimal criterion for to GHC to ensure that it can actually *use* a package. If the package is missing, GHC can't find the interface files or object code associated with the package. Ignoring packages is a way of pretending that a package is missing from the database.

Package validity is also a very weak criterion. Another criterion we might hope holds is *consistency*: when we consider the transitive closure of all selected packages, for any given package name, there should only be one instance providing that package. It is trivially easy to break this property: suppose that I have packages `a-1.0`, `b-1.0` compiled against `a-1.0`, and `a-1.1`. GHC will happily load `b-1.0` and `a-1.1` together in the same interactive session (they are both valid and the latest versions), even though `b-1.0`'s dependency is inconsistent with another package that was loaded. The user will notice if they attempt to treat entities from `a` reexported by `b-1.0` and entities from `a-1.1` as type equal. Here is one user who had this problem: `http://stackoverflow.com/questions/12576817/`. In some cases, the problem is easy to work around (there is only one offending package which just needs to be hidden), but if the divergence is deep in two separate dependency hierarchies, it is often easier to just blow away the package database and try again.

Perversely, destructive reinstallation helps prevent these sorts of inconsistent databases. While inconsistencies can arise when multiple versions of a package are installed, multiple versions will frequently lead to the necessity of reinstalls. In the previous example, if a user attempts to Cabal install a package which depends on `a-1.1` and `b-1.0`, Cabal's dependency solver will propose reinstalling `b-1.0` compiled against `a-1.1`, in order to get a consistent set of dependencies. If this reinstall is accepted, we invalidate all packages in the database which were previously installed against `b-1.0` and `a-1.0`, excluding them from GHC's selection process and making it more likely that the user will see a consistent view of the database.

Enforcing consistent dependencies From the user's perspective, it would be desirable if GHC never loaded a set of packages whose dependencies were inconsistent. There are two ways we can go about doing this. First, we can improve GHC's logic so that it doesn't pick an inconsistent set. However, as a point of design, we'd like to keep whatever resolution GHC does as simple as possible (in an ideal world, we'd skip the validity checks entirely, but they ended up being necessary to prevent broken database from stopping GHC from starting up at all). In particular, GHC should *not* learn how to do backtracking constraint solving: that's in the domain of Cabal. Second, we can modify the logic of Cabal to enforce that the package database is always kept in a consistent state, similar to the consistency check Cabal applies to sandboxes, where it refuses to install a package to a sandbox if the resulting dependencies would not be consistent.

The second alternative is a appealing, but Cabal sandboxes are currently designed for small, self-contained single projects, as opposed to the global "universe" that a default environment is intended to provide. For example, with a Cabal sandbox environment, it's impossible to upgrade a dependency to a new version without blowing away the sandbox and starting again. To support upgrades, Cabal needs to do some work: when a new version is put in the default set, all of the reverse-dependencies of the old version are now inconsistent. Cabal should offer to hide these packages or reinstall them compiled against the latest version. Furthermore, because we in general may not have write access to all visible package databases, this visibility information must be independent of the package databases themselves.

As a nice bonus, Cabal should also be able to snapshot the older environment which captures the state of the universe prior to the installation, in case the user wants to revert back.

Modifying the default environment Currently, after GHC calculates the default package environment, a user may further modify the environment by passing package flags to GHC, which can be used to explicitly hide or expose packages. How do these flags interact with our Cabal-managed environments? Hiding packages is simple enough, but exposing packages is a bit dicier. If a user asks for a different version of a package

[10]Recall that currently, a package ID uniquely identifies a package in the package database

than in the default set, it will probably be inconsistent with the rest of the dependencies. Cabal would have to be consulted to figure out a maximal set of consistent packages with the constraints given. Alternatively, we could just supply the package with no claims of consistency.

However, this use-case is rare. Usually, it's not because they want a specific version: the package is hidden simply because we're not interested in loading it by default (ghc-api is the canonical example, since it dumps a lot of modules in the top level namespace). If we distinguish packages which are consistent but hidden, their loads can be handled appropriately.

Consistency in Backpack We have stated as an implicit assumption that if we have both foo-1.0 and foo-1.1 available, only one should be loaded at a time. What are the consequences if both of these packages are loaded at the same time? An import of Data.Foo provided by both packages would be ambiguous and the user might find some type equalities they expect to hold would not. However, the result is not *unsound*: indeed, we might imagine a user purposely wanting two different versions of a library in the same program, renaming the modules they provided so that they could be referred to unambiguously. As another example, suppose that we have an indefinite package with a hole that is instantiated multiple times. In this case, a user absolutely may want to refer to both instantiations, once again renaming modules so that they have unique names.

There are two consequences of this. First, while the default package set may enforce consistency, a user should still be able to explicitly ask for a package instance, renamed so that its modules don't conflict, and then use it in their program. Second, instantiated indefinite packages should *never* be placed in the default set, since it's impossible to know which instantiation is the one the user prefers. A definite package can reexport an instantiated module under an unambiguous name if the user so pleases.

Shadowing, installed package IDs, ABI hashes and package keys Shadowing plays an important role for maintaining the soundness of compilation; call this the *compatibility* of the package set. The problem it addresses is when there are two distinct implementations of a module, but because their package ID (or package key, in the new world order) are the same, they are considered type equal. It is absolutely wrong for a single program to include both implementations simultaneously (the symbols would conflict and GHC would incorrectly conclude things were type equal when they're not), so *shadowing*'s job is to ensure that only one instance is picked, and all the other instances considered invalid (and their reverse-dependencies, etc.) Recall that in current GHC, within a package database, a package instance is uniquely identified by its package ID; thus, shadowing only needs to take place between package databases. An interesting corner case is when the same package ID occurs in both databases, but the installed package IDs are the *same*. Because the installed package ID is currently simply an ABI hash, we skip shadowing, because the packages are—in principle—interchangeable.

There are currently a number of proposed changes to this state of affairs:

- Change installed package IDs to not be based on ABI hashes. ABI hashes have a number of disadvantages as identifiers for packages in the database. First, they cannot be computed until after compilation, which gave the multi-instance GSoC project a few years some headaches. Second, it's not really true that programs with identical ABI hashes are interchangeable: a new package may be ABI compatible but have different semantics. Thus, installed package IDs are a poor unique identifier for packages in the package database. However, because GHC does not give ABI stability guarantees, it would not be possible to assume from here that packages with the same installed package ID are ABI compatible.

- Relaxing the uniqueness constraint on package IDs. There are actually two things that could be done here. First, since we have augmented package IDs with dependency resolution information to form package keys, we could simply state that package keys uniquely identify a package in a database. Shadowing rules can be implemented in the same way as before, by preferring the instance topmost on the stack. Second, we could also allow *same-database* shadowing: that is, not even package keys are guaranteed to be unique in a database: instead, installed package IDs are the sole unique identifier of a package. This architecture is Nix inspired, as the intent is to keep all package information in a centralized database.

Without mandatory package environments, same-database shadowing is a bad idea, because GHC now has no idea how to resolve shadowing. Conflicting installed package IDs can be simulated by placing them in multiple package databases (in principle, the databases can be concatenated together and treated as a single monolitic database.)

4 Shapeless Backpack

Backpack as currently defined always requires a *shaping* pass, which calculates the shapes of all modules defined in a package. The shaping pass is critical to the solution of the double-vision problem in recursive module linking, but it also presents a number of unpalatable implementation problems:

- *Shaping is a lot of work.* A module shape specifies the providence of all data types and identifiers defined by a module. To calculate this, we must preprocess and parse all modules, even before we do the type-checking pass. (Fortunately, shaping doesn't require a full parse of a module, only enough to get identifiers. However, it does have to understand import statements at the same level of detail as GHC's renamer.)

- *Shaping must be done upfront.* In the current Backpack design, all shapes must be computed before any typechecking can occur. While performing the shaping pass upfront is necessary in order to solve the double vision problem (where a module identity may be influenced by later definitions), it means that GHC must first do a shaping pass, and then revisit every module and compile them proper. Nor is it (easily) possible to skip the shaping pass when it is unnecessary, as one might expect to be the case in the absence of mutual recursion. Shaping is not a "pay as you go" language feature.

- *GHC can't compile all programs shaping accepts.* Shaping accepts programs that GHC, with its current hs-boot mechanism, cannot compile. In particular, GHC requires that any data type or function in a signature actually be *defined* in the module corresponding to that file (i.e., an original name can be assigned to these entities immediately.) Shaping permits unrestricted exports to implement modules; this shows up in the formalism as β module variables.

- *Shaping encourages inefficient program organization.* Shaping is designed to enable mutually recursive modules, but as currently implemented, mutual recursion is less efficient than code without recursive dependencies. Programmers should avoid this code organization, except when it is absolutely necessary.

- *GHC is architecturally ill-suited for directly implementing shaping.* Shaping implies that GHC's internal concept of an "original name" be extended to accommodate module variables. This is an extremely invasive change to all aspects of GHC, since the original names assumption is baked quite deeply into the compiler. Plausible implementations of shaping requires all these variables to be skolemized outside of GHC.

To be clear, the shaping pass is fundamentally necessary for some Backpack packages. Here is the example which convinced Simon:

```
package p where
    A :: [data T; f :: T -> T]
    B = [export T(MkT), h; import A(f); data T = MkT; h x = f MkT]
    A = [export T(MkT), f, h; import B; f MkT = MkT]
```

The key to this example is that B *may or may not typecheck* depending on the definition of A. Because A reexports B's definition T, B will typecheck; but if A defined T on its own, B would not typecheck. Thus, we *cannot* typecheck B until we have done some analysis of A (the shaping analysis!)

Thus, it is beneficial (from an optimization point of view) to consider a subset of Backpack for which shaping is not necessary. Here is a programming discipline which does just that, which we will call the **linking restriction**: *Module implementations must be declared before signatures.* Formally, this restriction modifies the rule for merging polarized module shapes ($\widetilde{\tau}_1^{m_1} \oplus \widetilde{\tau}_2^{m_2}$) so that $\widetilde{\tau}_1^{-} \oplus \widetilde{\tau}_2^{+}$ is always undefined.[11]

Here is an example of the linking restriction. Consider these two packages:

```
package random where
    System.Random = [ ... ].hs

package monte-carlo where
    System.Random :: ...
    System.MonteCarlo = [ ... ].hs
```

Here, random is a definite package which may have been compiled ahead of time; monte-carlo is an indefinite package with a dependency on any package which provides `System.Random`.

Now, to link these two applications together, only one ordering is permissible:

```
package myapp where
    include random
    include monte-carlo
```

If myapp wants to provide its own random implementation, it can do so:

```
package myapp2 where
    System.Random = [ ... ].hs
    include monte-carlo
```

In both cases, all of `monte-carlo`'s holes have been filled in by the time it is included. The alternate ordering is not allowed.

Why does this discipline prevent mutually recursive modules? Intuitively, a hole is the mechanism by which we can refer to an implementation before it is defined; otherwise, we can only refer to definitions which preceed our definition. If there are never any holes *which get filled*, implementation links can only go backwards, ruling out circularity.

It's easy to see how mutual recursion can occur if we break this discipline:

```
package myapp2 where
    include monte-carlo
    System.Random = [ import System.MonteCarlo ].hs
```

4.1 Typechecking of definite modules without shaping

If we are not carrying out a shaping pass, we need to be able to calculate $\widetilde{\Xi}_{\mathsf{pkg}}$ on the fly. In the case that we are compiling a package—there will be no holes in the final package—we can show that shaping is unnecessary quite easily, since with the linking restriction, everything is definite from the get-go.

Observe the following invariant: at any given step of the module bindings, the physical context $\widetilde{\Phi}$ contains no holes. We can thus conclude that there are no module variables in any type shapes. As the only time a previously calculated package shape can change is due to unification, the incrementally computed shape is in fact the true one.

As far as the implementation is concerned, we never have to worry about handling module variables; we only need to do extra typechecks against (renamed) interface files.

Algorithm 1 Compilation of definite packages (assume `-hide-all-packages` on all `ghc` invocations)

```
procedure COMPILE(package $P$ where $\overline{B}$, $H$, $db$)          ▷ $H$ maps hole module names to identities
    $flags := \emptyset$
    $\mathcal{K} := \textsc{Hash}(P + H)$
    In-place register the package $\mathcal{K}$ in $db$
    for $B$ in $\overline{B}$ do
        case “$p = p$.hs”
            EXEC(ghc -c $p$.hs -package-db $db$ -this-package-key $\mathcal{K}$ $flags$)
        case “$p$ :: $p$.hsig”
            EXEC(ghc -c $p$.hsig -package-db $db$ -sig-of $H(p)$ $flags$)
        case “$p = p'$”
            $flags := flags$ -alias $p$ $p'$
        case “include $P'$ $\langle \overline{p_H \mapsto p'_H}, \overline{p \mapsto p'} \rangle$”
            let $H'(p_H) = \textsc{ResolveModule}(p'_H)$
            $\mathcal{K}' := \textsc{Compile}(P', H', db)$          ▷ Nota bene: not $flags$
            $flags := flags$ -package $\mathcal{K}'$ $\langle \overline{p \mapsto p'} \rangle$
    end for
    Remove $\mathcal{K}$ from $db$
    Install the complete package $\mathcal{K}$ to the global database
    return $\mathcal{K}$
end procedure
```

4.2 Compiling definite packages

The full recursive procedure for compiling a Backpack package using one-shot compilation is given in Figure 1. We recursively walk through Backpack descriptions, processing each line by invoking GHC and/or modifying our package state. Here is a more in-depth description of the algorithm, line-by-line:

The parameters To compile a package description for package P, we need to know H, the mapping of holes p_H in package P to physical module identities ν which are implementing them; this mapping is used to calculate the package key $\mathcal{K}$ for the package in question. Furthermore, we have an inplace package database *db* in which we will register intermediate build results, including partially compiled parent packages which may provide implementations of holes for packages they include.

4.3 Compiling implementations

We compile modules in the same way we do today, but with some extra package visibility $flags$, which let GHC know how to resolve imports and look up original names. We'll describe what the new flags are and also discuss some subtleties with module lookup.

In-place registration Perhaps surprisingly, we start compilation by registering the (uncompiled) package in the in-place package database. This registration does not expose packages, and is purely intended to inform the compilation of subpackages where to find modules that are provided by the parent (in-progress) package, as well as provide auxiliary information, e.g., such as the package name and version for error reporting. The pre-registration trick is an old one used by the GHC build system; the key invariant to look out for is that we shouldn't reference original names in modules that haven't been built yet. This is enforced by our manual tracking of holes in H: a module can't occur in H unless it's already been compiled!

[11]This seemed to be the crispest way of defining the restriction, although this means an error happens a bit later than I'd like it to: I'd prefer if we errored while merging logical contexts, but we don't know what is a hole at that point.

New package resolution algorithm Currently, invocations of `-package` and similar flags have the result of *hiding* other exposed packages with the same name. However, this is not going to work for Backpack: an indefinite package may get loaded multiple times with different instantiations, and it might even make sense to load multiple versions of the same package simultaneously, as long as their modules are renamed to not conflict.

Thus, we impose the following behavior change: when `-hide-all-packages` is specified, we do *not* automatically hide packages with the same name as a package specified by `-package` (or a similar flag): they are all included, even if there are conflicts. To deal with conflicts, we augment the syntax of `-package` to also accept a list of thinnings and renamings, e.g. `-package` containers $\langle$Data.Set, Data.Map $\mapsto$ Map$\rangle$ says to make visible for import Data.Set and Map (which is Data.Map renamed.) This means that `-package` containers-0.9 $\langle$Data.Set $\mapsto$ Set09$\rangle$ `-package` containers-0.8 $\langle$Data.Set $\mapsto$ Set08$\rangle$ now uses both packages concurrently (previously, GHC would hide one of them.)

Additionally, it's important to note that two packages exporting the same module do not *necessarily* cause a conflict; the modules may be linkable. For example, `-package` containers $\langle$Data.Set$\rangle$ `-package` containers $\langle$Data.Set$\rangle$ is fine, because precisely the same implementation of Data.Set is loaded in both cases. A similar situation can occur with signatures:

```
package p where
    A :: [ x :: Int ]
package q
    include p
    A :: [ y :: Int ]
    B = [ import A; z = x + y ] -- *
package r where
    A = [ x = 0; y = 0 ]
    include q
```

Here, both `p` and `q` are visible when compiling the starred module, which compiles with the flags `-package p`, but there are two interface files available: one available locally, and one from `p`. Both of these interface files are *forwarding* to the original implementation `r` (more on this in the "Compiling signatures" file), so rather than reporting an ambiguous import, we instead have to merge the two interface files together. This is done by simulating multiple imports: one to each interface file. This works because GHC does not consider symbols with equal original names as conflicting.

Note that we do not need to merge signatures with an implementation, in such cases, we should just use the implementation interface. E.g.

```
package p where
    A :: ...
package q where
    A = ...
    include p
    B = [ import A ]    -- *
```

Here, A is available both from `p` and `q`, but the use in the starred module should be done with respect to the full implementation.

The `-alias` flag We introduce a new flag `-alias` for aliasing modules. Aliasing is analogous to the merging that can occur when we include packages, but it also applies to modules which are locally defined. When we alias a module p with p', we require that p' exists in the current module mapping, and then we attempt to add an entry for it at entry p. If there is no mapping for p, this succeeds; otherwise, we apply the same conflict resolution algorithm.

4.4 Compiling signatures

Signature compilation is triggered when we compile a signature file. This mode similar to how we process `hs-boot` files, except we pass an extra flag `-sig-of` which specifies what the identity of the actual implementation of the signature is (according to our H mapping). This is guaranteed to exist, due to the linking restriction, although it may be in a partially registered package in *db*. If the module is *not* currently available under the same name of the `hsig` file, we output an `hi` file which, for all declarations the signature exposes, forwards their definitions to the original implementation file. The intent is that any code in the current package which compiles against this signature will use this signature `hi` file, not the original one `hi` file. For example, the `hi` file produced when compiling the starred interface points to the implementation in package `q`.

```
package p where
    A :: ...    -- *
    B = [ import A; ... ]
package q where
    A = [ ... ]
    include p
```

Sometimes `hi` is unnecessary In the following package:

```
package p where
    P = ...
    P :: ...
```

Paper Backpack specifies that we check the signature `P` against implementation `P`, but otherwise no changes are made (i.e., the signature does not narrow the implementation.) In this case, it is not necessary to generate an `hi` file; the original interface file suffices.

Multiple signatures As a simplification, we assume that there is only one signature per logical name in a package. (This prevents us from expressing mutual recursion in signatures, but let's not worry about it for now.)

Restricted recursive modules ala hs-boot When we compile an `hsig` file without any `-sig-of` flag (because no implementation is known), we fall back to old-style GHC mutual recursion. Naïvely, a shaping pass would be necessary; so we adopt an existing constraint that already applies to hs-boot files: *at the time we define a signature, we must know what the original name for all data types is.* In practice, GHC enforces this by stating that: (1) an hs-boot file must be accompanied with an implementation, and (2) the implementation must in fact define (and not reexport) all of the declarations in the signature. We can discover if a signature is intended to break a recursive module loop when we discover that $p \notin flags_H$; in this case, we fallback to the old hs-boot behavior. (Alternatively, the user can explicitly ask for it.)

Why does this not require a shaping pass? The reason is that the signature is not really polymorphic: we require that the α module variable be resolved to a concrete module later in the same package, and that all the β module variables be unified with α. Thus, we know ahead of time the original names and don't need to deal with any renaming.[12]

Compiling packages in this way gives the tantalizing possibility of true separate compilation: the only thing we don't know is what the actual package name of an indefinite package will be, and what the correct references to have are. This is a very minor change to the assembly, so one could conceive of dynamically rewriting these references at the linking stage. But separate compilation achieved in this fashion would not be able to take advantage of cross-module optimizations.

[12]This strategy doesn't completely resolve the problem of cross-package mutual recursion, because we need to first compile a bit of the first package (signatures), then the second package, and then the rest of the first package.

4.5 Compiling includes

Includes are the most interesting part of the compilation process, as we have calculate how the holes of the subpackage we are filling in are compiled H' and modify our flags to make the exports of the include visible to subsequently compiled modules. We consider the case with renaming, since includes with no renaming are straightforward.

First, we assume that we know *a priori* what the holes of a package p_H are (either by some sort of pre-pass, or explicit declaration.) For each of their *renamed targets* p'_H, we determine what the original module associated with the p'_H is, based off of the package database that we have been manipulating. For example:

```
package p where
    A :: ...
    ...
package q where
    A = [ ... ]
    B = [ ... ]
    include p (A as B)
```

When computing the entry $H(\mathsf{A})$, we determine what the original module for B is.

Next, we recursively call COMPILE with the computed H'. Note that the entries in H may refer to modules which would not be picked up by $flags$, but they will be registered in the inplace package database *db*. For example, in this situation:

```
package p where
    B :: ...
    C = [ import B; ... ]
package q where
    A = [ ... ]
    B = [ import A; ... ]
    include p
    D = [ import C; ... ]
```

When we recursively process package p, H will refer to q:B, and we need to know where to find it (q is only partially processed and so is in the inplace package database.) Furthermore, the interface file for B may refer to q:A, and thus we likewise need to know how to find its interface file.

Note that the inplace package database is not expected to expose intermediate packages. Otherwise, this example would improperly compile:

```
package p where
    B = [ import A; ... ]
package q where
    A = ...
    include p
```

p does not compile on its own, so it should not compile if it is recursively invoked from q. However, if we exposed the modules of the partially registered package q, A is now suddenly resolvable.

Finally, once the subpackage is compiled, we can add it to our $flags$ so later modules we compile see its (appropriately thinned and renamed) modules, and like aliasing.

Absence of an `hi` file It is important that when we resolve a module, we look up the *implementor* of a module, and not just a signature which is providing it at some name. Sometimes, it can be a bit indirect, for example:

```
package p where
    A :: [ y :: Int ]
package q where
    A :: [ x :: Int ]
    include p -- *
package r where
    A = [ x = 0; y = 1 ]
    include q
```

When computing H' for the starred include, our $flags$ only include `-package-dir` r cwd_r $\langle\rangle$: with a thinning that excludes all modules! The only interface file we can pick up with these $flags$ is the local definition of A. However, we *absolutely* should set $H'(\mathsf{A})$ = `q : A`; if we do so, then we will incorrectly conclude when compiling the signature in `p` that the implementation doesn't export enough identifiers to fulfill the signature (`y` is not available from just the signature in `q`). Instead, we have to look up the original implementor of A in r, and use that in H'. If you maintain the invariant that you always know what the original implementor is of all modules in scope, it's easy enough to figure this out.

4.6 Commentary

Just because it compiled, doesn't mean the individual packages type check The compilation mechanism described is slightly more permissive than vanilla Backpack. Here is a simple example:

```
package p where
    A :: [ data T = T ]
    B :: [ data T = T ]
    C = [
        import A
        import B
        x = A.T :: B.T
    ]
package q where
    A = [ data T = T ]
    B = A
    include p
```

Here, we incorrectly decide A.T and B.T are type equal when typechecking C, because the `hisig` files we generate for them all point to the same original implementation. However, `p` should not typecheck.

The problem here is that type checking asks "does it compile with respect to all possible instantiations of the holes", whereas compilation asks "does it compile with respect to this particular instantiation of holes." In the absence of a shaping pass, this problem is unavoidable.

5 Shaped Backpack

Despite the simplicity of shapeless Backpack with the linking restriction in the absence of holes, we will find that when we have holes, it will be very difficult to do type-checking without some form of shaping. This section is very much a work in progress, but the ability to typecheck against holes, even with the linking restriction, is a very important part of modular separate development, so we will need to support it at some point.

5.1 Efficient shaping

(These are Edward's opinion, he hasn't convinced other folks that this is the right way to do it.)

In this section, I want to argue that, although shaping constitutes a pre-pass which must be run before compilation in earnest, it is only about as bad as the dependency resolution analysis that GHC already does in `ghc -M` or `ghc --make`.

In Paper Backpack, what information does shaping compute? It looks at exports, imports, data declarations and value declarations (but not the actual expressions associated with these values.) As a matter of fact, GHC already must look at the imports associated with a package in order to determine the dependency graph, so that it can have some order to compile modules in. There is a specialized parser which just parses these statements, and then ignores the rest of the file.

A bit of background: the *renamer* is responsible for resolving imports and figuring out where all of these entities actually come from. SPJ would really like to avoid having to run the renamer in order to perform a shaping pass.

Is it necessary to run the Renamer to do shaping? Edward and Scott believe the answer is no, well, partially. Shaping needs to know the original names of all entities exposed by a module/signature. Then it needs to know (a) which entities a module/signature defines/declares locally and (b) which entities that module/signature exports. The former, (a), can be determined by a straightforward inspection of a parse tree of the source file.[13] The latter, (b), is a bit trickier. Right now it's the Renamer that interprets imports and exports into original names, so we would still rely on that implementation. However, the Renamer does other, harder things that we don't need, so ideally we could factor out the import/export resolution from the Renamer for use in shaping.

Unfortunately the Renamer's import resolution analyzes `.hi` files, but for local modules, which haven't yet been typechecked, we don't have those. Instead, we could use a new file format, `.hsi` files, to store the shape of a locally defined module. (Defined packages are bundled with their shapes, so included modules have `.hsi` files as well.) (What about the logical vs. physical distinction in file names?) If we refactor the import/export resolution code, could we rewrite it to generically operate on both `.hi` files and `.hsi` files?

Alternatively, rather than storing shapes on a per-source basis, we could store (in memory) the entire package shape. Similarly, included packages could have a single shape file for the entire package. Although this approach would make shaping non-incremental, since an entire package's shape would be recomputed any time a constituent module's shape changes, we do not expect shaping to be all that expensive.

5.2 Typechecking of indefinite modules

Recall in our argument in the definite case, where we showed there are no holes in the physical context. With indefinite modules, this is no longer true. While (with the linking restriction) these holes will never be linked against a physical implementation, they may be linked against other signatures. (Note: while disallowing signature linking would solve our problem, it would disallow a wide array of useful instances of signature reuse, for example, a package mylib that implements both mylib-1x-sig and mylib-2x-sig.)

With holes, we must handle module variables, and we sometimes must unify them:

```
package p where
    A :: [ data A ]
package q where
    A :: [ data A ]
package r where
    include p
    include q
```

In this package, it is not possible to a priori assign original names to module A in p and q, because in package r, they should have the same original name. When signature linking occurs, unification may occur, which means we have to rename all relevant original names. (A similar situation occurs when a module is typechecked against a signature.)

[13]Note that no expression or type parsing is necessary. We only need names of local values, data types, and data constructors.

An invariant which would be nice to have is this: when typechecking a signature or including a package, we may apply renaming to the entities being brought into context. But once we've picked an original name for our entities, no further renaming should be necessary. (Formally, in the unification for semantic object shapes, apply the unifier to the second shape, but not the first one.)

However, there are plenty of counterexamples here:

```
package p where
    A :: [ data A ]
    B :: [ data A ]
    M = ...
    A = B
```

In this package, does module M know that A.A and B.A are type equal? In fact, the shaping pass will have assigned equal module identities to A and B, so M *equates these types*, despite the aliasing occurring after the fact.

We can make this example more sophisticated, by having a later subpackage which causes the aliasing; now, the decision is not even a local one (on the other hand, the equality should be evident by inspection of the package interface associated with q):

```
package p where
    A :: [ data A ]
    B :: [ data A ]
package q where
    A :: [ data A ]
    B = A
package r where
    include p
    include q
```

Another possibility is that it might be acceptable to do a mini-shaping pass, without parsing modules or signatures, *simply* looking at names and aliases. But logical names are not the only mechanism by which unification may occur:

```
package p where
    C :: [ data A ]
    A = [ data A = A ]
    B :: [ import A(A) ]
    C = B
```

It is easy to conclude that the original names of C and B are the same. But more importantly, C.A must be given the original name of p:A.A. This can only be discovered by looking at the signature definition for B. In any case, it is worth noting that this situation parallels the situation with hs-boot files (although there is no mutual recursion here).

The conclusion is that you will probably, in fact, have to do real shaping in order to typecheck all of these examples.

Hey, these signature imports are kind of tricky. . . When signatures and modules are interleaved, the interaction can be complex. Here is an example:

```
package p where
    C :: [ data A ]
    M = [ import C; ... ]
    A = [ import M; data A = A ]
    C :: [ import A(A) ]
```

Here, the second signature for C refers to a module implementation A (this is permissible: it simply means that the original name for p:C.A is p:A.A). But wait! A relies on M, and M relies on C. Circularity? Fortunately not: a client of package p will find it impossible to have the hole C implemented in advance, since they will need to get their hands on module A. . . but it will not be defined prior to package p.

In any case, however, it would be good to emit a warning if a package cannot be compiled without mutual recursion.

5.3 Rename on entry

Consider the following example:

```
package p where
    A :: [ data T = T ]
    B = [ import A; x = T ]
package q where
    C :: ...
    A = [ data T = T ]
    include p
    D = [
        import qualified A
        import qualified B
        import C
        x = B.T :: A.T
    ]
```

We are interested in type-checking q, which is an indefinite package on account of the uninstantiated hole C. Furthermore, let's suppose that p has already been independently typechecked, and its interface files installed in some global location with α_A used as the module identity of A. (To simplify this example, we'll assume $\beta_{AT} = \alpha_A$.)

The first three lines of q type check in the normal way, but D now poses a problem: if we load the interface file for B the normal way, we will get a reference to type T with the original name α_A.T, whereas from A we have an original name q:A.T.

Let's suppose that we already have the result of a shaping pass, which maps our identity variables to their true identities. Let's consider the possible options here:

- We could re-typecheck p, feeding it the correct instantiations for its variables. However, this seems wasteful: we typechecked the package already, and up-to-renaming, the interface files are exactly what we need to type check our application.
- We could make copies of all the interface files, renamed to have the right original names. This also seems wasteful: why should we have to create a new copy of every interface file in a library we depend on?
- When *reading in* the interface file to GHC, we could apply the renaming according to the shaping pass and store that in memory.

That last solution is pretty appealing, however, there are still circumstances we need to create new interface files; these exactly mirror the cases described in Section 4.2.

5.4 Incremental typechecking

We want to typecheck modules incrementally, i.e., when something changes in a package, we only want to re-typecheck the modules that care about that change. GHC already does this today.[14] Is the same

[14] https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/RecompilationAvoidance

mechanism sufficient for Backpack? Edward and Scott think that it is, mostly. Our conjecture is that a module should be re-typechecked if the existing mechanism says it should *or* if the logical shape context (which maps logical names to physical names) has changed. The latter condition is due to aliases that affect typechecking of modules.

Let's look again at an example from before:

```
package p where
    A :: [ data A ]
    B :: [ data A ]
    M = [ import A; import B; ... ]
```

Let's say that `M` is typechecked successfully. Now we add an alias binding at the end of the package, `A = B`. Does `M` need to be re-typechecked? Yes! (Well, it seems so, but let's just assert "yes" for now. Certainly in the reverse case—if we remove the alias and then ask—this is true, since `M` might have depended on the two `A` types being the same.) The logical shape context changed to say that `A` and `B` now map to the same physical module identity. But does the existing recompilation avoidance mechanism say that `M` should be re-typechecked? It's unclear. The `.hi` file for `M` records that it imported `A` and `B` with particular ABIs, but does it also know about the physical module identities (or rather, original module names) of these modules?

Scott thinks this highlights the need for us to get our story straight about the connection between logical names, physical module identities, and file names!

5.5 Installing indefinite packages

If an indefinite package contains no code at all, we only need to install the interface file for the signatures. However, if they include code, we must provide all of the ingredients necessary to compile them when the holes are linked against actual implementations. (Figure **??**)

Source tarball or preprocessed source? What is the representation of the source that is saved is. There are a number of possible choices:

- The original tarballs downloaded from Hackage,
- Preprocessed source files,
- Some sort of internal, type-checked representation of Haskell code (maybe the output of the desugarer).

Storing the tarballs is the simplest and most straightforward mechanism, but we will have to be very certain that we can recompile the module later in precisely the same we compiled it originally, to ensure the hi files match up (fortunately, it should be simple to perform an optional sanity check before proceeding.) The appeal of saving preprocessed source, or even the IRs, is that this is conceptually this is exactly what an indefinite package is: we have paused the compilation process partway, intending to finish it later. However, our compilation strategy for definite packages requires us to run this step using a *different* choice of original names, so it's unclear how much work could actually be reused.

Sources in sandboxes Another nice way to implement indefinite packages is to register them as source packages in a Cabal sandbox, and then teach Cabal how to build them multiple times in the compile process. Perhaps the global package database should be extended with a directory of source packages in order to support indefinite packages.

6 Surface syntax

In the Backpack paper, a brand new module language is presented, with syntax for inline modules and signatures. This syntax is probably worth implementing, because it makes it easy to specify compatibility packages, whose module definitions in general may be very short:

```
package ishake-0.12-shake-0.13 where
    include shake-0.13
    Development.Shake.Sys = Development.Shake.Cmd
    Development.Shake = [ (**>) = (&>) ; (*>>) = (|*>)]
    Development.Shake.Rule = [ defaultPriority = rule . priority 0.5 ]
    include ishake-0.12
```

However, there are a few things that are less than ideal about the surface syntax proposed by Paper Backpack:

- It's completely different from the current method users specify packages. There's nothing wrong with this per se (one simply needs to support both formats) but the smaller the delta, the easier the new packaging format is to explain and implement.
- Sometimes order matters (relative ordering of signatures and module implementations), and other times it does not (aliases). This can be confusing for users.
- Users have to order module definitions topologically, whereas in current Cabal modules can be listed in any order, and GHC figures out an appropriate order to compile them.

Here is an alternative proposal, closely based on Cabal syntax. Given the following Backpack definition:

```
package libfoo(A, B, C, Foo) where
    include base
    -- renaming and thinning
    include libfoo (Foo, Bar as Baz)
    -- holes
    A :: [ a :: Bool ].hsig
    A2 :: [ b :: Bool ].hsig
    -- normal module
    B = [
        import {-# SOURCE #-} A
        import Foo
        import Baz
        ...
    ].hs
    -- recursively linked pair of modules, one is private
    C :: [ data C ].hsig
    D = [ import {-# SOURCE #-} C; data D = D C ].hs
    C = [ import D; data C = C D ].hs
    -- alias
    A = A2
```

We can write the following Cabal-like syntax instead (where all of the signatures and modules are placed in appropriately named files):

```
package: libfoo
...
build-depends: base, libfoo (Foo, Bar as Baz)
required-signatures: A A2 -- deferred for now
exposed-modules: Foo B C
aliases: A = A2
other-modules: D
```

Notably, all of these lists are *insensitive* to ordering! The key idea is use of the `{-# SOURCE #-}` pragma, which is enough to solve the important ordering constraint between signatures and modules.

Here is how the elaboration works. For simplicity, in this algorithm description, we assume all packages being compiled have no holes (including `build-depends` packages). Later, we'll discuss how to extend the algorithm to handle holes in both subpackages and the main package itself.

1. At the top-level with `package` *p* and `exposed-modules` *ms*, record `package p (ms) where`
2. For each package *p* with thinning/renaming *ms* in `build-depends`, record a `include p (ms)` in the Backpack package. The ordering of these includes does not matter, since none of these packages have holes.
3. Take all modules *m* in `other-modules` and `exposed-modules` which were not exported by build dependencies, and create a directed graph where hs and hs-boot files are nodes and imports are edges (the target of an edge is an hs file if it is a normal import, and an hs-boot file if it is a SOURCE import). Topologically sort this graph, erroring if this graph contains cycles (even with recursive modules, the cycle should have been broken by an hs-boot file). For each node, in this order, record `M = [ ... ]` or `M :: [ ... ]` depending on whether or not it is an hs or hs-boot. If possible, sort signatures before implementations when there is no constraint otherwise.

Here is a simple example which shows how SOURCE can be used to disambiguate between two important cases. Suppose we have these modules:

```
-- A1.hs
import {-# SOURCE #-} B

-- A2.hs
import B

-- B.hs
x = True

-- B.hs-boot
x :: Bool
```

Then we translate the following packages as follows:

```
exposed-modules: A1 B
-- translates to
B :: [ x :: Bool ]
A1 = [ import B ]
B = [ x = True ]
```

but

```
exposed-modules: A2 B
-- translates to
B = [ x = True ]
B :: [ x :: Bool ]
A2 = [ import B ]
```

The import controls placement between signature and module, and in A1 it forces B's signature to be sorted before B's implementation (whereas in the second section, there is no constraint so we preferentially place the B's implementation first)

Holes in the database In the presence of holes, `build-depends` resolution becomes more complicated. First, let's consider the case where the package we are building is definite, but the package database contains indefinite packages with holes. In order to maintain the linking restriction, we now have to order packages from step (2) of the previous elaboration. We can do this by creating a directed graph, where nodes are packages and edges are from holes to the package which implements them. If there is a cycle, this indicates a mutually recursive package. In the absence of cycles, a topological sorting of this graph preserves the linking invariant.

One subtlety to consider is the fact that an entry in `build-depends` can affect how a hole is instantiated by another entry. This might be a bit weird to users, who might like to explicitly say how holes are filled when instantiating a package. Food for thought, surface syntax wise.

Holes in the package Actually, this is quite simple: the ordering of includes goes as before, but some indefinite packages in the database are less constrained as they're "dependencies" are fulfilled by the holes at the top-level of this package. It's also worth noting that some dependencies will go unresolved, since the following package is valid:

```
package a where
    A :: ...
package b where
    include a
```

Multiple signatures In Backpack syntax, it's possible to define a signature multiple times, which is necessary for mutually recursive signatures:

```
package a where
    A :: [ data A ]
    B :: [ import A; data B = B A ]
    A :: [ import B; data A = A B ]
```

Critically, notice that we can see the constructors for both module B and A after the signatures are linked together. This is not possible in GHC today, but could be possible by permitting multiple hs-boot files. Now the SOURCE pragma indicating an import must *disambiguate* which hs-boot file it intends to include. This might be one way of doing it:

```
-- A.hs-boot2
data A

-- B.hs-boot
import {-# SOURCE hs-boot2 #-} A

-- A.hs-boot
import {-# SOURCE hs-boot #-} B
```

Explicit or implicit reexports One annoying property of this proposal is that, looking at the `exposed-modules` list, it is not immediately clear what source files one would expect to find in the current package. It's not obvious what the proper way to go about doing this is.

Better syntax for SOURCE If we enshrine the SOURCE import as a way of solving Backpack ordering problems, it would be nice to have some better syntax for it. One possibility is:

```
abstract import Data.Foo
```

which makes it clear that this module is pluggable, typechecking against a signature. Note that this only indicates how type checking should be done: when actually compiling the module we will compile against the interface file for the true implementation of the module.

It's worth noting that the SOURCE annotation was originally made a pragma because, in principle, it should have been possible to compile some recursive modules without needing the hs-boot file at all. But if we're moving towards boot files as signatures, this concern is less relevant.

7 Type classes and type families

7.1 Background

Before we talk about how to support type classes in Backpack, it's first worth talking about what we are trying to achieve in the design. Most would agree that *type safety* is the cardinal law that should be preserved (in the sense that segfaults should not be possible), but there are many instances of "bad behavior" (top level mutable state, weakening of abstraction guarantees, ambiguous instance resolution, etc) which various Haskellers may disagree on the necessity of ruling out.

With this in mind, it is worth summarizing what kind of guarantees are presently given by GHC with regards to type classes and type families, as well as characterizing the *cultural* expectations of the Haskell community.

Type classes When discussing type class systems, there are several properties that one may talk about. A set of instances is *confluent* if, no matter what order constraint solving is performed, GHC will terminate with a canonical set of constraints that must be satisfied for any given use of a type class. In other words, confluence says that we won't conclude that a program doesn't type check just because we swapped in a different constraint solving algorithm.

Confluence's closely related twin is *coherence* (defined in "Type classes: exploring the design space"). This property states that "every different valid typing derivation of a program leads to a resulting program that has the same dynamic semantics." Why could differing typing derivations result in different dynamic semantics? The answer is that context reduction, which picks out type class instances, elaborates into concrete choices of dictionaries in the generated code. Confluence is a prerequisite for coherence, since one can hardly talk about the dynamic semantics of a program that doesn't type check.

In the vernacular, confluence and coherence are often incorrectly used to refer to another related property: *global uniqueness of instances*, which states that in a fully compiled program, for any type, there is at most one instance resolution for a given type class. Languages with local type class instances such as Scala generally do not have this property, and this assumption is frequently used for abstraction.

So, what properties does GHC enforce, in practice? In the absence of any type system extensions, GHC's employs a set of rules (described in "Exploring the design space") to ensure that type class resolution is confluent and coherent. Intuitively, it achieves this by having a very simple constraint solving algorithm (generate wanted constraints and solve wanted constraints) and then requiring the set of instances to be *nonoverlapping*, ensuring there is only ever one way to solve a wanted constraint. Overlap is a more stringent restriction than either confluence or coherence, and via the `OverlappingInstances` and `IncoherentInstances`, GHC allows a user to relax this restriction "if they know what they're doing."

Surprisingly, however, GHC does *not* enforce global uniqueness of instances. Imported instances are not checked for overlap until we attempt to use them for instance resolution. Consider the following program:

```
-- T.hs
data T = T
-- A.hs
import T
instance Eq T where
-- B.hs
```

```
import T
instance Eq T where
-- C.hs
import A
import B
```

When compiled with one-shot compilation, `C` will not report overlapping instances unless we actually attempt to use the `Eq` instance in C.[15] This is by design[16]: ensuring that there are no overlapping instances eagerly requires eagerly reading all the interface files a module may depend on.

We might summarize these three properties in the following manner. Culturally, the Haskell community expects *global uniqueness of instances* to hold: the implicit global database of instances should be confluent and coherent. GHC, however, does not enforce uniqueness of instances: instead, it merely guarantees that the *subset* of the instance database it uses when it compiles any given module is confluent and coherent. GHC does do some tests when an instance is declared to see if it would result in overlap with visible instances, but the check is by no means perfect[17]; truly, *type-class constraint resolution* has the final word. One mitigating factor is that in the absence of *orphan instances*, GHC is guaranteed to eagerly notice when the instance database has overlap.[18]

Clearly, the fact that GHC's lazy behavior is surprising to most Haskellers means that the lazy check is mostly good enough: a user is likely to discover overlapping instances one way or another. However, it is relatively simple to construct example programs which violate global uniqueness of instances in an observable way:

```
-- A.hs
module A where
data U = X | Y deriving (Eq, Show)

-- B.hs
module B where
import Data.Set
import A

instance Ord U where
compare X X = EQ
compare X Y = LT
compare Y X = GT
compare Y Y = EQ

ins :: U -> Set U -> Set U
ins = insert

-- C.hs
module C where
import Data.Set
import A

instance Ord U where
compare X X = EQ
```

[15]When using batch compilation, GHC reuses the instance database and is actually able to detect the duplicated instance when compiling B. But if you run it again, recompilation avoidance skips A, and it finishes compiling! See this bug: `https://ghc.haskell.org/trac/ghc/ticket/5316`

[16]`https://ghc.haskell.org/trac/ghc/ticket/2356`

[17]`https://ghc.haskell.org/trac/ghc/ticket/9288`

[18]Assuming that the instance declaration checks actually worked. . .

```
compare X Y = GT
compare Y X = LT
compare Y Y = EQ

ins' :: U -> Set U -> Set U
ins' = insert

-- D.hs
module Main where
import Data.Set
import A
import B
import C

test :: Set U
test = ins' X $ ins X $ ins Y $ empty

main :: IO ()
main = print test

-- OUTPUT
$ ghc -Wall -XSafe -fforce-recomp --make D.hs
[1 of 4] Compiling A ( A.hs, A.o )
[2 of 4] Compiling B ( B.hs, B.o )

B.hs:5:10: Warning: Orphan instance: instance [safe] Ord U
[3 of 4] Compiling C ( C.hs, C.o )

C.hs:5:10: Warning: Orphan instance: instance [safe] Ord U
[4 of 4] Compiling Main ( D.hs, D.o )
Linking D ...
$ ./D
fromList [X,Y,X]
```

Locally, all type class resolution was coherent: in the subset of instances each module had visible, type class resolution could be done unambiguously. Furthermore, the types of `ins` and `ins'` discharge type class resolution, so that in `D` when the database is now overlapping, no resolution occurs, so the error is never found.

It is easy to dismiss this example as an implementation wart in GHC, and continue pretending that global uniqueness of instances holds. However, the problem with *global uniqueness of instances* is that they are inherently nonmodular: you might find yourself unable to compose two components because they accidentally defined the same type class instance, even though these instances are plumbed deep in the implementation details of the components.

As it turns out, there is already another feature in Haskell which must enforce global uniqueness, to prevent segfaults. We now turn to type classes' close cousin: type families.

Type families With type families, confluence is the primary property of interest. (Coherence is not of much interest because type families are elaborated into coercions, which don't have any computational content.) Rather than considering what the set of constraints we reduce to, confluence for type families considers the reduction of type families. The overlap checks for type families can be quite sophisticated, especially in the case of closed type families.

Unlike type classes, however, GHC *does* check the non-overlap of type families eagerly. The analogous program does *not* type check:

```
-- F.hs
type family F a :: *
-- A.hs
import F
type instance F Bool = Int
-- B.hs
import F
type instance F Bool = Bool
-- C.hs
import A
import B
```

The reason is that it is *unsound* to ever allow any overlap (unlike in the case of type classes where it just leads to incoherence.) Thus, whereas one might imagine dropping the global uniqueness of instances invariant for type classes, it is absolutely necessary to perform global enforcement here. There's no way around it!

7.2 Local type classes

Here, we say **NO** to global uniqueness.

This design is perhaps best discussed in relation to modular type classes, which shares many similar properties. Instances are now treated as first class objects (in MTCs, they are simply modules)—we may explicitly hide or include instances for type class resolution (in MTCs, this is done via the `using` top-level declaration). This is essentially what was sketched in Section 5 of the original Backpack paper. As a simple example:

```
package p where
    A :: [ data T = T ]
    B = [ import A; instance Eq T where ... ]

package q where
    A = [ data T = T; instance Eq T where ... ]
    include p
```

Here, `B` does not see the extra instance declared by `A`, because it was thinned from its signature of `A` (and thus never declared canonical.) To declare an instance without making it canonical, it must placed in a separate (unimported) module.

Like modular type classes, Backpack does not give rise to incoherence, because instance visibility can only be changed at the top level module language, where it is already considered best practice to provide explicit signatures. Here is the example used in the Modular Type Classes paper to demonstrate the problem:

```
structure A = using EqInt1 in
    struct ...fun f x = eq(x,x)... end
structure B = using EqInt2 in
    struct ...val y = A.f(3)... end
```

Is the type of f `int -> bool`, or does it have a type-class constraint? Because type checking proceeds over the entire program, ML could hypothetically pick either. However, ported to Haskell, the example looks like this:

```
EqInt1 :: [ instance Eq Int ]
EqInt2 :: [ instance Eq Int ]
A = [
    import EqInt1
    f x = x == x
]
B = [
    import EqInt2
    import A hiding (instance Eq Int)
    y = f 3
]
```

There may be ambiguity, yes, but it can be easily resolved by the addition of a top-level type signature to `f`, which is considered best-practice anyway. Additionally, Haskell users are trained to expect a particular inference for `f` in any case (the polymorphic one).

Here is another example which might be considered surprising:

```
package p where
    A :: [ data T = T ]
    B :: [ data T = T ]
    C = [
        import qualified A
        import qualified B
        instance Show A.T where show T = "A"
        instance Show B.T where show T = "B"
        x :: String
        x = show A.T ++ show B.T
    ]
```

In the original Backpack paper, it was implied that module `C` should not type check if `A.T = B.T` (failing at link time). However, if we set aside, for a moment, the issue that anyone who imports `C` in such a context will now have overlapping instances, there is no reason in principle why the module itself should be problematic. Here is the example in MTCs, which I have good word from Derek does type check.

```
signature SIG = sig
    type t
    val mk : t
end
signature SHOW = sig
    type t
    val show : t -> string
end
functor Example (A : SIG) (B : SIG) =
    let structure ShowA : SHOW = struct
        type t = A.t
        fun show _ = "A"
    end in
    let structure ShowB : SHOW = struct
        type t = B.t
        fun show _ = "B"
    end in
    using ShowA, ShowB in
    struct
```

```
        val x = show A.mk ++ show B.mk
    end : sig val x : string end
```

The moral of the story is, even though in a later context the instances are overlapping, inside the functor, the type-class resolution is unambiguous and should be done (so `x = "AB"`).

Up until this point, we've argued why MTCs and this Backpack design are similar. However, there is an important sociological difference between modular type-classes and this proposed scheme for Backpack. In the presentation "Why Applicative Functors Matter", Derek mentions the canonical example of defining a set:

```
signature ORD = sig type t; val cmp : t -> t -> bool end
signature SET = sig type t; type elem;
    val empty : t;
    val insert : elem -> t -> t ...
end
functor MkSet (X : ORD) :> SET where type elem = X.t
  = struct ... end
```

This is actually very different from how sets tend to be defined in Haskell today. If we directly encoded this in Backpack, it would look like this:

```
package mk-set where
    X :: [
        data T
        cmp :: T -> T-> Bool
    ]
    Set :: [
        data Set
        empty :: Set
        insert :: T -> Set -> Set
    ]
    Set = [
        import X
        ...
    ]
```

It's also informative to consider how MTCs would encode set as it is written today in Haskell:

```
signature ORD = sig type t; val cmp : t -> t -> bool end
signature SET = sig type 'a t;
    val empty : 'a t;
    val insert : (X : ORD) => X.t -> X.t t -> X.t t
end
struct MkSet :> SET = struct ... end
```

Here, it is clear to see that while functor instantiation occurs for implementation, it is not occuring for types. This is a big limitation with the Haskell approach, and it explains why Haskellers, in practice, find global uniqueness of instances so desirable.

Implementation-wise, this requires some subtle modifications to how we do type class resolution. Type checking of indefinite modules works as before, but when go to actually compile them against explicit implementations, we need to "forget" that two types are equal when doing instance resolution. This could probably be implemented by associating type class instances with the original name that was utilized when typechecking, so that we can resolve ambiguous matches against types which have the same original name now that we are compiling.

As we've mentioned previously, this strategy is unsound for type families.

7.3 Globally unique

Here, we say **YES** to global uniqueness.

When we require the global uniqueness of instances (either because that's the type class design we chose, or because we're considering the problem of type families), we will need to reject declarations like the one cited above when `A.T = B.T`:

```
A :: [ data T ]
B :: [ data T ]
C :: [
    import qualified A
    import qualified B
    instance Show A.T where show T = "A"
    instance Show B.T where show T = "B"
]
```

The paper mentions that a link-time check is sufficient to prevent this case from arising. While in the previous section, we've argued why this is actually unnecessary when local instances are allowed, the link-time check is a good match in the case of global instances, because any instance *must* be declared in the signature. The scheme proceeds as follows: when some instances are typechecked initially, we type check them as if all of variable module identities were distinct. Then, when we perform linking (we `include` or we unify some module identities), we check again if to see if we've discovered some instance overlap. This linking check is akin to the eager check that is performed today for type families; it would need to be implemented for type classes as well: however, there is a twist: we are *redoing* the overlap check now that some identities have been unified.

As an implementation trick, one could deferring the check until `C` is compiled, keeping in line with GHC's lazy "don't check for overlap until the use site." (Once again, unsound for type families.)

What about module inequalities? An older proposal was for signatures to contain "module inequalities", i.e., assertions that two modules are not equal. (Technically: we need to be able to apply this assertion to β module variables, since `A != B` while `A.T = B.T`). Currently, Edward thinks that module inequalities are only marginal utility with local instances (i.e., not enough to justify the implementation cost) and not useful at all in the world of global instances!

With local instances, module inequalities could be useful to statically rule out examples like `show A.T ++ show B.T`. Because such uses are not necessarily reflected in the signature, it would be a violation of separate module development to try to divine the constraint from the implementation itself. I claim this is of limited utility, however, because, as we mentioned earlier, we can compile these "incoherent" modules perfectly coherently. With global instances, all instances must be in the signature, so while it might be aesthetically displeasing to have the signature impose extra restrictions on linking identities, we can carry this out without violating the linking restriction.

8 Bits and bobs

8.1 Abstract type synonyms

In Paper Backpack, abstract type synonyms are not permitted, because GHC doesn't understand how to deal with them. The purpose of this section is to describe one particularly nastiness of abstract type synonyms, by way of the occurs check:

```
A :: [ type T ]
B :: [ import qualified A; type T = [A.T] ]
```

At this point, it is illegal for `A = B`, otherwise this type synonym would fail the occurs check. This seems like pretty bad news, since every instance of the occurs check in the type-checker could constitute a module inequality.

9 Open questions

Here are open problems about the implementation which still require hashing out.

- In Section 4, we argued that we could implement Backpack without needing a shaping pass. We're pretty certain that this will work for typechecking and compiling fully definite packages with no recursive linking, but in Section 5.2, we described some of the prevailing difficulties of supporting signature linking. Renaming is not an insurmountable problem, but backwards flow of shaping information can be, and it is unclear how best to accommodate this. This is probably the most important problem to overcome.

- In Section 5.5, a few choices for how to store source code were pitched, however, there is not consensus on which one is best.

- What is the impact of the multiplicity of PackageIds on dependency solving in Cabal? Old questions of what to prefer when multiple package-versions are available (Cabal originally only needed to solve this between different versions of the same package, preferring the oldest version), but with signatures, there are more choices. Should there be a complex solver that does all signature solving, or a preprocessing step that puts things back into the original Cabal version. Authors may want to suggest policy for what packages should actually link against signatures (so a crypto library doesn't accidentally link against a null cipher package).