The Backpack algorithm

July 30, 2015

In this document, we look at the compilation of a Backpack unit. Here are the steps a unit goes through:

1. The **unit renamer** takes the Backpack file consisting of units transforms each unit name to an indefinite unit ID *IndefUnitId*. In particular, it associates each unit name with its local binding site (unit declaration), or an external unit declaration from the indefinite unit database.
2. The **dependency solver** takes a *unit* and converts it into a directed acyclic graph representing the dependency structure of the declarations in a unit. It also computes the *Module* of each module/signature declaration, the *UnitKey* of each include, and the overall requirements of the unit.
3. The **shaping pass** takes the DAG and computes its *Shape*. A *Shape* describes precisely what a unit provides and requires at the Haskell declaration level (*AvailInfo*).
4. The **indefinite pipeline** takes the DAG and the shape and typechecks each module and signature against the indefinite unit database. The type checking results are saved in the indefinite unit database under the *IndefiniteUnitId*.
5. The **definite pipeline** takes the DAG as well as a *hole mapping* specifying how each requirement in the unit is to be filled, and type-checks and compiles the unit against the installed unit database. The type checking results and object files are saved to the installed unit database under the *InstalledUnitId*.

1 Front-end syntax

p, q, r		Unit names	
m, n		Module names	
Units			
unit	::=	`unit` p [*provreq*] `where {`d_1`;` ... `;`d_n`}`	
Declarations			
d	::=	`module` m [*exports*] `where` *body*	
	|	`signature` m [*exports*] `where` *body*	
	|	`include` p [*provreq*]	
Provides/requires specification			
provreq	::=	`(` *rns* `)` [`requires(` *rns* `)`]	
rns	::=	rn_0`,` ... `,` rn_n[`,`]	Renamings
rn	::=	m `as` n	Renaming
Haskell code			
exports		A Haskell module export list	
body		A Haskell module body	

Figure 1: Syntax of Backpack

A (slightly simplified) syntax of Backpack is given in Figure 1.

2 Unit renamer

InstalledPackageId		Installed package IDs
IndefiniteUnitId	::=	*InstalledPackageId* `-` *p*
InstalledUnitId	::=	*IndefiniteUnitId* `(` *m* `->` *Module* `, ...` `)` Also known as *UnitKey*
Module	::=	*InstalledUnitId* `:` *m*

Figure 2: Unit identification

The unit renamer is responsible for transforming unit names p into *IndefiniteUnitId*s, given some current *InstalledPackageId* (*ThisInstalledPackageId*) and a mapping from p to *IndefiniteUnitId* (*UnitNameMap*). Its operation on a Backpack file (collection of units) is very simple:

- Every unit declaration `unit` p is renamed to *ThisInstalledPackageId* `-` p
- Every unit include `include` p is renamed to *ThisInstalledPackageId* `-` p if p was declared in the Backpack file; otherwise it is renamed according to *UnitNameMap*.

The purpose of an *IndefiniteUnitId* is to uniquely identify the results of typechecking an indefinite unit; similarly, an *InstalledUnitId* uniquely identifies the results of compiling a unit with all its holes filled. It thus records a *hole mapping* which specifies how each hole was filled.

An *installed package ID* (IPID) is an opaque string provided by Cabal which uniquely identifies an installed package. A recipe for computing an IPID would incorporate both the source info, such as the hash of the source code distribution tarball, as well as build info, such as the selected Cabal and GHC flags, what the provided mapping from p to $IndefiniteUnitId$ was, etc.

The difference between units and packages Cabal packages are:

- The unit of distribution
- The unit that Hackage handles
- The unit of versioning
- The unit of ownership (who maintains it etc)

Backpack units are the building blocks of modular development; there may be multiple units per a Cabal package. While in theory Cabal could do sophisticated things with multiple units in a package, we expect Cabal to pick a distinguished unit (with the same unit name p as the package) which serves as the publically visible unit.

Notational convention In the rest of this document, when it is unambiguous, we will use p, q and r interchangeably with *IndefiniteUnitId*, as after unit renaming, there are no occurrences of unit names.

3 Dependency solver

$$
\begin{array}{rcl}
\tilde{d} & ::= & \texttt{module}\ \mathit{Module}\ [\mathit{exports}]\ \texttt{where}\ \mathit{body} \\
& | & \texttt{signature}\ \mathit{Module}\ [\mathit{exports}]\ \texttt{where}\ \mathit{body} \\
& | & \texttt{include}\ p\ \texttt{(}\ m\ \texttt{->}\ \mathit{Module}\ \texttt{, ...)} \\
& | & \texttt{merge}\ \mathit{Module}\ \texttt{of}\ (\mathit{Module}, \mathit{IsSource?})\ldots
\end{array}
$$

Figure 3: Resolved declarations

The first phase of compilation is defines a directed acyclic graph on the source syntax representing the dependency structure of the modules/signatures/includes in the unit. This DAG has a node per:

- Each source-level module, signature and include, and
- Each unfilled requirement (called a "signature merge" node).

Each module, signature and signature merge node can be identified with the tuple (*Module*, *IsSource*?), where *IsSource*? is true for signatures and false for modules and signature merges. The four nodes are described in Figure 3.

The edges of the directed graph signify a "depends on" relation, and are defined as follows:

- A module/signature m depends on `include` p if m imports a module provided by p.
- A module/signature m depends on a module/signature merge n if m imports n.
- A module/signature m depends on a signature n if m `{-# SOURCE #-}` imports n.
- A module/signature merge m depends on a local signature m (if it exists).
- A module/signature merge m depends on a `include` p, if the include requires m.

For compilation, these extra edges can also be defined if they do not introduce a cycle:

- An `include` p depends on `include` q if, for some module name m, p requires m and q provides m.
- An `include` p depends on a module m if p requires a module named m.

If the resulting graph has a cycle, this is an error.

Computing unfilled requirements To compute unfilled requirements, maintain two sets of module names: the provisions P and the possible requirements R'. For each declaration:

- `include` p: union provisions with P and requirements with R'.
- `module` m: add m to P
- `signature` m: add m to R'

The unfilled requirements $R = R' - P$.

Computing the *Module* of declarations The *Module* of any declaration m in a unit p is simply `p(A -> HOLE:A, ...):m`, where the hole map is a map from each unfilled requirement n to `HOLE:n`.

Computing the hole mapping of includes In absence of mutual recursion of includes, the DAG is acyclic with include-include edges. Process includes in this topological order, maintaining a mapping of provided modules Γ, accumulating provisions of includes as we go along. For each `include`;p, the hole map is simply the requirements of p, mapping m to $\Gamma(m)$ if it is defined, and `HOLE:m` otherwise.

With mutual recursion, we have to use the regular tree unification algorithm described in the Backpack paper. We omit it from here for now.

4 Shaping

Shape	::=	`provides:` m `->` *Module* `{` *AvailInfo* `, ... };` ...	
		`requires:` m `->` `{` *AvailInfo* `, ... };` ...	
AvailInfo	::=	*Name*	Plain identifiers
	|	*Name* `{` $Name_0$ `, ... ,` $Name_n$ `}`	Type constructors
Name	::=	*Module* `.` *OccName*	
OccName		Unqualified name in a namespace	

Figure 4: Shaping

Shaping computes a *Shape*, whose form is described in Figure 4. A shape describes what modules a unit implements and exports (the *provides*) and what signatures a unit needs to have filled in (the *requires*). Both provisions and requires are available for import by units which include this unit.

We incrementally build a shape by starting with an empty shape context and adding to it as follows:

1. Calculate the shape of a declaration, with respect to the current shape context. (e.g., by renaming a module/signature, or using the shape from an included unit.)
2. Merge this shape into the shape context.

The final shape context is the shape of the unit as a whole. Optionally, we can also compute the renamed syntax trees of modules and signatures.

In the description below, we'll assume `THIS` is the unit key of the unit being processed.

4.1 `module M`

A module declaration provides a module `THIS:M` at module name `M`. It has the shape:

```
provides: M -> THIS:M { exports of renamed M under THIS:M }
requires: (nothing)
```

Example:

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

-- provides: A -> THIS:A { THIS:A.T }
-- requires: (nothing)
```

***OccName* is implied by *Name*.** In Haskell, the following is not valid syntax:

```
import A (foobar as baz)
```

In particular, a *Name* which is in scope will always have the same *OccName* (even if it may be qualified.) You might imagine relaxing this restriction so that declarations can be used under different *OccName*s; in such a world, we need a different definition of shape:

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

Presently, however, such an *OccName* annotation would be redundant: it can be inferred from the *Name*.

Holes of a unit are a mapping, not a set. Why can't the *UnitKey* just record a set of *Module*s, e.g. *UnitKey* ::= *SrcUnitKey* `{` *Module* `}`? Consider:

```
unit p (A) requires (H1, H2) where
    signature H1(T) where
        data T
    signature H2(T) where
        data T
    module A(A(..)) where
        import qualified H1
        import qualified H2
        data A = A H1.T H2.T

unit q (A12, A21) where
    module I1(T) where
        data T = T Int
    module I2(T) where
        data T = T Bool
    include p (A as A12) requires (H1 as I1, H2 as I2)
    include p (A as A21) requires (H1 as I2, H2 as I1)
```

With a mapping, the first instance of `p` has key `p(H1 -> q():I1, H2 -> q():I2)` while the second instance has key `p(H1 -> q():I2, H2 -> q():I1)`; with a set, both would have the key `p(q():I1, q():I2)`.

Signatures can require a specific entity. With requirements like `A -> { HOLE:A.T, HOLE:A.foo }`, why not specify it as `A -> { T, foo }`, e.g., `required: { ModName -> { OccName } }`? Consider:

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

The requirements of this unit specify that `A.T` = `B.T`; this can be expressed with *Name*s as

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

But, without *Name*s, the sharing constraint is impossible: `A -> { T }; B -> { T }`. (NB: `A` and `B` don't have to be implemented with the same module.)

The *Name* of a value is used to avoid ambiguous identifier errors. We state that two types are equal when their *Name*s are the same; however, for values, it is less clear why we care. But consider this example:

```
unit p (A) requires (H1, H2) where
    signature H1(x) where
        x :: Int
    signature H2(x) where
        import H1(x)
    module A(y) where
        import H1
        import H2
        y = x
```

The reference to `x` in `A` is unambiguous, because it is known that `x` from `H1` and `x` from `H2` are the same (have the same *Name*.) If they were not the same, it would be ambiguous and should cause an error. Knowing the *Name* of a value distinguishes between these two cases.

Holes are linear Requirements do not record what *Module* represents the identity of a requirement, which means that it's not possible to assert that hole `A` and hole `B` should be implemented with the same module, as might occur with aliasing:

```
signature A where
signature B where
alias A = B
```

The benefit of this restriction is that when a requirement is filled, it is obvious that this is the only requirement that is filled: you won't magically cause some other requirements to be filled. The downside is it's not possible to write a unit which looks for an interface it is looking for in one of n names, accepting any name as an acceptable linkage. If aliasing was allowed, we'd need a separate physical shaping context, to make sure multiple mentions of the same hole were consistent.

4.2 `signature M`

A signature declaration creates a requirement at module name `M`. It has the shape:

```
provides: (nothing)
requires: M -> { exports of renamed M under HOLE:M }
```

Example:

```
signature H(T) where
    data T

-- provides: H -> (nothing)
-- requires: H -> { HOLE:H.T }
```

In-scope signatures are not provisions. We enforce the invariant that a provision is always (syntactically) a `module` and a requirement is always a `signature`. This means that if you have a requirement and a provision of the same name, the requirement can *always* be filled with the provision. Without this invariant, it's not clear if a provision will actually fill a signature. Consider this example, where a signature is required and exposed:

```
unit a-sigs (A) requires (A) where -- ***
    signature A where
        data T

unit a-user (B) requires (A) where
    signature A where
        data T
        x :: T
    module B where
        ...

unit p where
    include a-sigs
    include a-user
```

When we consider merging in the shape of `a-user`, does the `A` provided by `a-sigs` fill in the `A` requirement in `a-user`? It *should not*, since `a-sigs` does not actually provide enough declarations to satisfy `a-user`'s requirement: the intended semantics *merges* the requirements of `a-sigs` and `a-user`.

```
unit a-sigs (M as A) requires (H as A) where
    signature H(T) where
        data T
    module M(T) where
        import H(T)
```

We rightly should error, since the provision is a module. And in this situation:

```
unit a-sigs (H as A) requires (H) where
    signature H(T) where
        data T
```

The requirements should be merged, but should the merged requirement be under the name `H` or `A`?
It may still be possible to use the `(A) requires (A)` syntax to indicate exposed signatures, but this would be a mere syntactic alternative to `() requires (exposed A)`.

4.3 `include pkg (X) requires (Y)`

We merge with the transformed shape of unit `pkg`, where this shape is transformed by:

- Renaming and thinning the provisions according to `(X)`
- Renaming requirements according to `(Y)` (requirements cannot be thinned, so non-mentioned requirements are implicitly passed through.) For each renamed requirement from `Y` to `Y'`, substitute `HOLE:Y` with `HOLE:Y'` in the *Module*s and *Name*s of the provides and requires.

If there are no thinnings/renamings, you just merge the shape unchanged! Here is an example:

```
unit p (M) requires (H) where
    signature H where
        data T
    module M where
        import H
        data S = S T

unit q (A) where
    module X where
        data T = T
    include p (M as A) requires (H as X)
```

The shape of unit `p` is:

```
requires: M -> { p(H -> HOLE:H):M.S }
provides: H -> { HOLE:H.T }
```

Thus, when we process the `include` in unit `q`, we make the following two changes: we rename the provisions, and we rename the requirements, substituting `HOLE`s. The resulting shape to be merged in is:

```
provides: A -> { p(H -> HOLE:X):M.S }
requires: X -> { HOLE:X.T }
```

After merging this in, the final shape of `q` is:

```
provides: X -> { q():X.T }              -- from shaping 'module X'
          A -> { p(H -> q():X):M.S }
requires: (nothing)                     -- discharged by provided X
```

4.4 Merging

The shapes we've given for individual declarations have been quite simple. Merging combines two shapes, filling requirements with implementations, unifying *Name*s, and unioning requirements; it is the most complicated part of the shaping process.

The best way to think about merging is that we take two units with inputs (requirements) and outputs (provisions) and "wiring" them up so that outputs feed into inputs. In the absence of mutual recursion, this wiring process is *directed*: the provisions of the first unit feed into the requirements of the second unit, but never vice versa. (With mutual recursion, things can go in the opposite direction as well.)

Suppose we are merging shape p with shape q (e.g., $p; q$). Merging proceeds as follows:

1. *Fill every requirement of q with provided modules from p.* For each requirement M of q that is provided by p (in particular, all of its required `Name`s are provided), substitute each *Module* occurrence of `HOLE:M` with the provided p`(`M`)`, unify the names, and remove the requirement from q. If the names of the provision are not a superset of the required names, error.

2. If mutual recursion is supported, *fill every requirement of p with provided modules from q.*

3. *Merge leftover requirements.* For each requirement M of q that is not provided by p but required by p, unify the names, and union them together to form the new requirement. (It's not necessary to substitute *Module*s, since they are guaranteed to be the same.)

4. *Add provisions of q.* Union the provisions of p and q, erroring if there is a duplicate that doesn't have the same identity.

To unify two sets of names, find each pair of names with matching *OccName*s n and m and do the following:

1. If both are from holes, pick a canonical representative m and substitute n with m.

2. If one n is from a hole, substitute n with m.

3. Otherwise, error if the names are not the same.

It is important to note that substitutions on *Module*s and substitutions on *Name*s are disjoint: a substitution from `HOLE:A` to `HOLE:B` does *not* substitute inside the name `HOLE:A.T`.

Since merging is the most complicated step of shaping, here are a big pile of examples of it in action.

4.4.1 A simple example

In the following set of units:

```
unit p(M) requires (A) where
    signature A(T) where
        data T
    module M(T, S) where
        import A(T)
        data S = S T

unit q where
    module A where
        data T = T
    include p
```

When we `include p`, we need to merge the partial shape of `q` (with just provides `A`) with the shape of `p`. Here is each step of the merging process:

```
          shape 1                        shape 2
          ------------------------------------------------------------------------------
(initial shapes)
provides: A -> THIS:A { q():A.T }       M -> p(A -> HOLE:A) { HOLE:A.T, p(A -> HOLE:A).S }
requires: (nothing)                     A ->                { HOLE:A.T }

(after filling requirements)
provides: A -> THIS:A { q():A.T }       M -> p(A -> THIS:A) { q():A.T, p(A -> THIS:A).S }
requires: (nothing)                     (nothing)

(after adding provides)
provides: A -> THIS:A         { q():A.T }
          M -> p(A -> THIS:A) { q():A.T, p(A -> THIS:A).S }
requires: (nothing)
```

Notice that we substituted `HOLE:A` with `THIS:A`, but `HOLE:A.T` with `q():A.T`.

4.4.2 Requirements merging can affect provisions

When a merge results in a substitution, we substitute over both requirements and provisions:

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

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

signature H(T, f) where
    import B(T)
    f :: a -> a

-- provides: A -> THIS:A { THIS:B.T }           -- UPDATED
--           B -> THIS:B { THIS:B.T }
-- requires: H ->        { THIS:B.T, HOLE:H.f } -- UPDATED
```

4.4.3 Sharing constraints

Suppose you have two signature which both independently define a type, and you would like to assert that these two types are the same. In the ML world, such a constraint is known as a sharing constraint. Sharing constraints can be encoded in Backpacks via clever use of reexports; they are also an instructive example for signature merging.

```
signature A(T) where
    data T
signature B(T) where
    data T

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

```
            B -> { HOLE:B.T }

-- the sharing constraint!
signature A(T) where
    import B(T)
-- (shape to merge)
-- requires: A -> { HOLE:B.T }

-- (after merge)
-- requires: A -> { HOLE:A.T }
--           B -> { HOLE:A.T }
```

I'm pretty sure any choice of *Name* is OK, since the subsequent substitution will make it alpha-equivalent.

4.5 Export declarations

If an explicit export declaration is given, the final shape is the computed shape, minus any provisions not mentioned in the list, with the appropriate renaming applied to provisions and requirements. (Requirements are implicitly passed through if they are not named.) If no explicit export declaration is given, the final shape is the computed shape, including only provisions which were defined in the declarations of the unit.

Signature visibility, and defaulting The simplest formulation of requirements is to have them always be visible. Signature visibility could be controlled by associating every requirement with a flag indicating if it is importable or not: a signature declaration sets a requirement to be visible, and an explicit export list can specify if a requirement is to be visible or not.
When an export list is absent, we have to pick a default visibility for a signature. If we use the same behavior as with modules, a strange situation can occur:

```
unit p where -- S is visible
    signature S where
        x :: True

unit q where -- use defaulting
    include p
    signature S where
        y :: True
    module M where
        import S
        z = x && y      -- OK

unit r where
    include q
    module N where
        import S
        z = y           -- OK
        z = x           -- ???
```

Absent the second signature declaration in `q`, `S.x` clearly should not be visible in `N`. However, what ought to occur when this signature declaration is added? One interpretation is to say that only some (but not all) declarations are provided (`S.x` remains invisible); another interpretation is that adding `S` is enough to treat the signature as "in-line", and all declarations are now provided (`S.x` is visible).
The latter interpretation avoids having to keep track of providedness per declarations, and means that you can always express defaulting behavior by writing an explicit provides declaration on the unit. However, it has the odd behavior of making empty signatures semantically meaningful:

```
unit q where
    include p
    signature S where
```

4.6 Merging AvailInfos

We describe how to take two sets of *AvailInfo*s and merges them into one set. In the degenerate case where every *AvailInfo* is a *Name*, this algorithm operates the same as the original algorithm. Merging proceeds in two steps: unification and then simple union.

Unification proceeds as follows: for each pair of *Name*s with matching *OccName*s, unify the names. For each pair of *Name* `{` $Name_0$ `,` ... `,` $Name_n$ `}`, where there exists some pair of child names with matching *OccName*s, unify the parent *Name*s. (A single *AvailInfo* may participate in multiple such pairs.) A simple identifier and a type constructor *AvailInfo* with overlapping in-scope names fails to unify. After unification, the simple union combines entries with matching `availName`s (parent name in the case of a type constructor), recursively unioning the child names of type constructor *AvailInfo*s.

Unification of *Name*s results in a substitution, and a *Name* substitution on *AvailInfo* is a little unconventional. Specifically, substitution on *Name* `{` $Name_0$ `,` ... `,` $Name_n$ `}` proceeds specially: a substitution from *Name* to $Name'$ induces a substitution from *Module* to $Module'$ (as the *OccName*s of the *Name*s are guaranteed to be equal), so for each child $Name_i$, perform the *Module* substitution. So for example, the substitution `HOLE:A.T` to `THIS:A.T` takes the *AvailInfo* `HOLE:A.T { HOLE:A.B, HOLE:A.foo }` to `THIS:A.T { THIS:A.B, THIS:A.foo }`. In particular, substitution on children *Name*s is *only* carried out by substituting on the outer name; we will never directly substitute children.

Unfortunately, there are a number of tricky scenarios:

Merging when type constructors are not in scope

```
signature A1(foo) where
    data A = A { foo :: Int, bar :: Bool }

signature A2(bar) where
    data A = A { foo :: Int, bar :: Bool }
```

If we merge `A1` and `A2`, are we supposed to conclude that the types `A1.A` and `A2.A` (not in scope!) are the same? The answer is no! Consider these implementations:

```
module A1(A(..)) where
    data A = A { foo :: Int, bar :: Bool }

module A2(A(..)) where
    data A = A { foo :: Int, bar :: Bool }

module A(foo, bar) where
    import A1(foo)
    import A2(bar)
```

Here, `module A1` implements `signature A1`, `module A2` implements `signature A2`, and `module A` implements `signature A1` and `signature A2` individually and should certainly implement their merge. This is why we cannot simply merge type constructors based on the *OccName* of their top-level type; merging only occurs between in-scope identifiers.

Does merging a selector merge the type constructor?

```
signature A1(A(..)) where
    data A = A { foo :: Int, bar :: Bool }

signature A2(A(..)) where
    data A = A { foo :: Int, bar :: Bool }
```

```
signature A2(foo) where
    import A1(foo)
```

Does the last signature, which is written in the style of a sharing constraint on `foo`, also cause `bar` and the type and constructor `A` to be unified? Because a merge of a child name results in a substitution on the parent name, the answer is yes.

Incomplete data declarations

```
signature A1(A(foo)) where
    data A = A { foo :: Int }

signature A2(A(bar)) where
    data A = A { bar :: Bool }
```

Should `A1` and `A2` merge? If yes, this would imply that data definitions in signatures could only be *partial* specifications of their true data types. This seems complicated, which suggests this should not be supported; however, in fact, this sort of definition, while disallowed during type checking, should be *allowed* during shaping. The reason that the shape we abscribe to the signatures `A1` and `A2` are equivalent to the shapes for these which should merge:

```
signature A1(A(foo)) where
    data A = A { foo :: Int, bar :: Bool }

signature A2(A(bar)) where
    data A = A { foo :: Int, bar :: Bool }
```

4.7 Subtyping record selectors as functions

```
signature H(A, foo) where
    data A
    foo :: A -> Int

module M(A, foo) where
    data A = A { foo :: Int, bar :: Bool }
```

Does `M` successfully fill `H`? If so, it means that anywhere a signature requests a function `foo`, we can instead validly provide a record selector. This capability seems quite attractive, although in practice record selectors rarely seem to be abstracted this way: one reason is that `M.foo` still *is* a record selector, and can be used to modify a record. (Many library authors find this suprising!)

Nor does this seem to be an insurmountable instance of the avoidance problem: as a workaround, `H` can equivalently be written as:

```
signature H(foo) where
    data A = A { foo :: Int, bar :: Bool }
```

However, you might not like this, as the otherwise irrelevant `bar` must be mentioned in the definition.

In any case, actually implementing this 'subtyping' is quite complicated, because we can no longer assume that every child name is associated with a parent name. The technical difficulty is that we now need to unify a plain identifier *AvailInfo* (from the signature) with a type constructor *AvailInfo* (from a module.) It is not clear what this should mean. Consider this situation:

```
unit p where
```

```
    signature H(A, foo, bar) where
        data A
        foo :: A -> Int
        bar :: A -> Bool
    module X(A, foo) where
        import H
unit q where
    include p
    signature H(bar) where
        data A = A { foo :: Int, bar :: Bool }
    module Y where
        import X(A(..)) -- ???
```

Should the wildcard import on `X` be allowed? This question is equivalent to whether or not shaping discovers whether or not a function is a record selector and propagates this information elsewhere. If the wildcard is not allowed, here is another situation:

```
unit p where
    -- define without record selectors
    signature X1(A, foo) where
        data A
        foo :: A -> Int
    module M1(A, foo) where
        import X1

unit q where
    -- define with record selectors (X1s unify)
    signature X1(A(..)) where
        data A = A { foo :: Int, bar :: Bool }
    signature X2(A(..)) where
        data A = A { foo :: Int, bar :: Bool }

    -- export some record selectors
    signature Y1(bar) where
        import X1
    signature Y2(bar) where
        import X2

unit r where
    include p
    include q

    -- sharing constraint
    signature Y2(bar) where
        import Y1(bar)

    -- the payload
    module Test where
        import M1(foo)
        import X2(foo)
        ... foo ... -- conflict?
```

Without the sharing constraint, the `foo`s from `M1` and `X2` should conflict. With it, however, we should

conclude that the `foo`s are the same, even though the `foo` from `M1` is *not* considered a child of `A`, and even though in the sharing constraint we *only* unified `bar` (and its parent `A`). To know that `foo` from `M1` should also be unified, we have to know a bit more about `A` when the sharing constraint performs unification; however, the *AvailInfo* will only tell us about what is in-scope, which is *not* enough information.

5 Type checking

PkgType ::= *ModIface*$_0$; ... ; *ModIface*$_n$

Module interface

ModIface ::= `module` *Module* (*mi_exports*) `where`
mi_decls
mi_insts
dep_orphs

mi_exports	::=	*AvailInfo*$_0$, ... , *AvailInfo*$_n$	Export list
mi_decls	::=	*IfaceDecl*$_0$; ... ; *IfaceDecl*$_n$	Defined declarations
mi_insts	::=	*IfaceClsInst*$_0$; ... ; *IfaceClsInst*$_n$	Defined instances
dep_orphs	::=	*Module*$_0$; ... ; *Module*$_n$	Transitive orphan dependencies

Interface declarations

IfaceDecl	::=	*OccName* :: *IfaceId*
	|	`data` *OccName* = *IfaceData*
	|	...
IfaceClsInst		A type-class instance
IfaceId		Interface of top-level binder
IfaceData		Interface of type constructor

Figure 5: Module interfaces in GHC

In general terms, type checking an indefinite unit (a unit with holes) involves calculating, for every module, a *ModIface* representing the type/interface of the module in question (which is serialized to disk). The general form of these interface files are described in Figure 5; notably, the interfaces *IfaceId*, *IfaceData*, etc. contain *Name* references, which must be resolved by looking up a *ModIface* corresponding to the *Module* associated with the *Name*. (We will say more about this lookup process shortly.) For example, given:

```
unit p where
    signature H where
        data T
    module A(S, T) where
        import H
        data S = S T
```

the *PkgType* is:

```
module HOLE:H (HOLE:H.T) where
    data T -- abstract type constructor
module THIS:A (THIS:A.S, HOLE:H.T) where
    data S = S HOLE:H.T
-- where THIS = p(H -> HOLE:H)
```

However, while it is true that the *ModIface* is the final result of type checking, we actually are conflating two distinct concepts: the user-visible notion of a *ModuleName*, which, when imported, brings some *Name*s

into scope (or could trigger a deprecation warning, or pull in some orphan instances...), versus the actual declarations, which, while recorded in the *ModIface*, have an independent existence: even if a declaration is not visible for an import, we may internally refer to its *Name*, and need to look it up to find out type information. (A simple case when this can occur is if a module exports a function with type `T -> T`, but doesn't export `T`).

$$
\begin{array}{rcl}
\mathit{ModDetails} & ::= & \langle \mathit{md\_types}\texttt{;}\ \mathit{md\_insts} \rangle \\
\mathit{md\_types} & ::= & \mathit{TyThing}_0\texttt{, ... ,}\ \mathit{TyThing}_n \\
\mathit{md\_insts} & ::= & \mathit{ClsInst}_0\texttt{, ... ,}\ \mathit{ClsInst}_n
\end{array}
$$

Type-checked declarations

TyThing	Type-checked thing with a *Name*
ClsInst	Type-checked type class instance

Figure 6: Semantic objects in GHC

Thus, a *ModIface* can be type-checked into a *ModDetails*, described in Figure 6. Notice that a *ModDetails* is just a bag of type-checkable entities which GHC knows about. We define the *external package state (EPT)* to simply be the union of the *ModDetails* of all external modules.

Type checking is a delicate balancing act between module interfaces and our semantic objects. A *ModIface* may get type-checked multiple times with different hole instantiations to provide multiple *ModDetails*. Furthermore complicating matters is that GHC does this resolution *lazily*: a *ModIface* is only converted to a *ModDetails* when we are looking up the type of a *Name* that is described by the interface; thus, unlike usual theoretical treatments of type checking, we can't eagerly go ahead and perform substitutions on *ModIface*s when they get included.

In a separate compiler like GHC, there are two primary functions we must provide:

ModuleName* to *ModIface Given a *ModuleName* which was explicitly imported by a user, we must produce a *ModIface* that, among other things, specifies what *Name*s are brought into scope. This is used by the renamer to resolve plain references to identifiers to real *Name*s. (By the way, if shaping produced renamed trees, it would not be necessary to do this step!)

***Module* to *ModDetails*/EPT** Given a *Module* which may be a part of a *Name*, we must be able to type check it into a *ModDetails* (usually by reading and typechecking the *ModIface* associated with the *Module*, but this process is involved). This is used by the type checker to find out type information on things.

There are two points in the type checker where these capabilities are exercised:

Source-level imports When a user explicitly imports a module, the *ModuleName* is mapped to a *ModIface* to find out what exports are brought into scope (*mi_exports*) and what orphan instances must be loaded (*dep_orphs*). Additionally, the *Module* is loaded to the EPT to bring instances from the module into scope.

Internal name lookup During type checking, we may have a *Name* for which we need type information (*TyThing*). If it's not already in the EPT, we type check and load into the EPT the *ModDetails* of the *Module* in the *Name*, and then check the EPT again. (`importDecl`)

5.1 *ModName* to *ModIface*

In all cases, the *mi_exports* can be calculated directly from the shaping process, which specifies exactly for each *ModName* in scope what will be brought into scope.

Modules Modules are straightforward, as for any *Module* there is only one possibly *ModIface* associated with it (the *ModIface* for when we type-checked the (unique) `module` declaration.)

Does hiding a signature hide its orphans. Suppose that we have extended Backpack to allow hiding signatures from import.

```
unit p requires (H) where -- H is hidden from import
    module A where
        instance Eq (a -> b) where -- orphan
    signature H {-# DEPRECATED "Don't use me" #-} where
        import A

unit q where
    include p
    signature H where
        data T
    module M where
        import H                -- warn deprecated?
        instance Eq (a -> b)    -- overlap?
```

It is probably the most consistent to not pull in orphan instances and not give the deprecated warning: this corresponds to merging visible *ModIface*s, and ignoring invisible ones.

Signatures For signatures, there may be multiple *ModIface*s associated with a *ModName* in scope, e.g. in this situation:

```
unit p where
    signature S where
        data A
unit q where
    include p
    signature S where
        data B
    module M where
        import S
```

Each literal `signature` has a *ModIface* associated with it; and the import of `S` in `M`, we want to see the *merged ModIface*s. We can determine the *mi_exports* from the shape, but we also need to pull in orphan instances for each signature, and produce a warning for each deprecated signature.

5.2 *Module* to *ModDetails*

Modules For modules, we have a *Module* of the form p`(`m `->` *Module*`, ...)`, and we also have a unique *ModIface*, where each hole instantiation is `HOLE:`m.

To generate the *ModDetails* associated with the specific instantiation, we have to type-check the *ModIface* with the following adjustments:

1. Perform a *Module* substitution according to the instantiation of the *ModIface*'s *Module*. (NB: we *do* substitute `HOLE:A.x` to `HOLE:B.x` if we instantiated `A -> HOLE:B`, *unlike* the disjoint substitutions applied by shaping.)

2. Perform a *Name* substitution as follows: for any name with a unit key that is a `HOLE`, substitute with the recorded *Name* in the requirements of the shape. Otherwise, look up the (unique) *ModIface* for the *Module*, and subsitute with the corresponding *Name* in the *mi_exports*.

Signatures For signatures, we have a *Module* of the form `HOLE:`m. Unlike modules, there are multiple *ModIface*s associated with a hole. We distinguish each separate *ModIface* by considering the full *UnitKey* it was defined in, e.g. `p(A -> HOLE:C, B -> q():B)`; call this the hole's *defining unit key*; the set of *ModIface*s for a hole and their defining unit keys can easily be calculated during shaping.

To generate the *ModDetails* associated with a hole, we type-check each *ModIface*, with the following adjustments:

1. Perform a *Module* substitution according to the instantiation of the defining unit key. (NB: This may rename the hole itself!)
2. Perform a *Name* substitution as follows, in the same manner as would be done in the case of modules.
3. When these *ModDetails* are merged into the EPT, some merging of duplicate types may occur; a type may be defined multiple times, in which case we check that each definition is compatible with the previous ones. A concrete type is always compatible with an abstract type.

Invariants When we perform *Name* substitutions, we must be sure that we can always find out the correct *Name* to substitute to. This isn't obviously true, consider:

```
unit p where
    signature S(foo) where
        data T
        foo :: T
    module M(bar) where
        import S
        bar = foo
unit q where
    module A(T(..)) where
        data T = T
        foo = T
    module S(foo) where
        import A
    include p
    module A where
        import M
        ... bar ...
```

When we type check `p`, we get the *ModIface*s:

```
module HOLE:S(HOLE:S.foo) where
    data T
    foo :: HOLE:S.T
module THIS:M(THIS:M.bar) where
    bar :: HOLE:S.T
```

Now, when we type check `A`, we pull on the *Name* `p(S -> q():S):M.bar`, which means we have to type check the *ModIface* for `p(S -> q():S):M`. The un-substituted type of `bar` has a reference to `HOLE:S.T`; this should be substituted to `q():S.T`. But how do we discover this? We know that `HOLE:S` was instantiated to `q():S`, so we might try and look for `q():S.T`. However, this *Name* does not exist because the `module S` reexports the selector from `A`! Nor can we consult the (unique) *ModIface* for the module, as it doesn't reexport the relevant type.

The conclusion, then, is that a module written this way should be disallowed. Specifically, the correctness condition for a signature is this: *Any Name mentioned in the ModIface of a signature must either be from an external module, or be exported by the signature.*

Special case export rule for record selectors. Here is the analogous case for record selectors:

```
unit p where
    signature S(foo) where
        data T = T { foo :: Int }
    module M(bar) where
        import S
        bar = foo
unit q where
    module A(T(..)) where
        data T = T { foo :: Int }
    module S(foo) where
        import A
    include p
    module A where
        import M
        ... bar ...
```

We could reject this, but technically we can find the right substitution for `T`, because the export of `foo` is an *AvailTC* which does mention `T`.