Backpack without symbol tables

Edward Z. Yang

May 11, 2016

Compilers usually have one or more data structures known as *symbol tables*, which are mappings from symbols to information about the symbol. In contrast, GHC avoids symbol tables as much as possible: instead, a symbol is a data structure that *contains* all information about itself, forming a (lazily initialized) immutable *graph* of all symbols known to GHC.

This strategy has a major downside: the fact that the graph is immutable means that we cannot easily *update* the information associated with a symbol.[1] In practice, the only way to update information is to rebuild the graph from scratch. On the up side, GHC rarely needs to update information in the graph; generally, the graph just gets bigger as we typecheck more declarations. On the down side, Backpack makes heavy use of updates—as an example, when we merge an abstract type `data T` with a type synonym `type T = Int`, we must update the type constructor with an updated unfolding of the type synonym.

In principle, the recipe for handling updates to graph is simple: rebuild the graph from scratch with the updated information. In practice, getting this all to work right is an extremely dark corner of GHC. The purpose of this paper is to shed some light on how GHC manages updates to this graph of symbols; we aim to be descriptive with respect to how GHC manages `hs-boot` files, and prescriptive with respect to the implementation of Backpack. Here's the plan:

- We describe GHC's *graph representation*, in which there are no symbol tables, and contrast it with the *interface representation*, which utilizes symbol tables, is serialized to disk, and serves as a template from which the graph can be constructed. We'll describe how to convert between these two representations. Most operations like type equality are implemented only for the graph representation, but operations which involve *updating* information associated with a symbol can only conveniently be done on the interface representation.

- `hs-boot` files pose some difficulty for the graph representation, as the incomplete symbols supplied by the `hs-boot` file must eventually be improved with the information from their implementations in the `hs` file. We describe how GHC ties the knot, so that it can rebuild its graph *at the same time* it is building the type constructors with updated information.

[1] As Haskell is a pure language, it would be highly inconvenient to make this graph mutable.

- What invariants do we expect these graph to uphold? If the user's source code is well-typed, we definitely expect the graph to generally make sense (e.g., be well-typed and well-kinded). But what if the user's code is ill-typed? We might still expect GHC's internal representation of types to make sense. Unfortunately, the eager knot-tying means that GHC constructs malformed types. A relatively harmless example are ill-kinded types; more harmfully, a type synonym loop can cause the compiler to nonterminate. This suggests that perhaps the knot should be tied only after typechecking is completed.
- The typechecking algorithm for Backpack requires a number of operations on GHC's representation of types: *substitution*, *type matching* and *merging*. We carefully spell out what these operations are, and give the strategy for implementing these operations in GHC.

1 Graph representation (without symbol tables)

The *graph representation* for types embeds information about symbols inside the data structure itself. For example, here are some data types for the graph representation from GHC (simplified and abbreviated):

```
data Type  = TyConApp TyCon [Type]  | ...
data TyCon = SynonymTyCon Name Type | ...
data ModDetails = ModDetails [TyCon] ...
```

A type may be an application of a type constructor to some arguments, in which case it contains a type constructor `TyCon`. A type constructor can be a type synonym, in which case it contains the type it expands to. A module `ModDetails` consists of a list of type constructors and other entities that are defined in it. The "graph" is the graph of heap objects which represent these data types.

It is very convenient to write algorithms which operate on the graph representation, as any query which would have required a lookup in a symbol table are reduced to a simple field access, improving efficiency and code clarity. For example, type equality can be defined as a pure function `Type -> Type -> Bool` even in the presence of type synonyms, since the `SynonymTyCon` always carries along the expanded version of the synonym to compare against.

2 Interface representation (with symbol tables)

The *interface representation* for types does not embed information in its types, and thus is suitable for serialization to binary interface files. For example, here are some data types for the interface representation (also simplified):

```
data IfaceType = IfaceTyConApp Name [IfaceType] | ...
data IfaceDecl = IfaceSynonym OccName IfaceType | ...
```

```
data ModIface = ModIface [IfaceDecl] ...
```

Unlike the graph representation, the type constructor application doesn't embed the definition of the type constructor—instead, it records only a `Name` reference to the constructor. To find out more information about the type constructor, you would have to lookup the `IfaceDecl` from the appropriate symbol table; e.g., a module local declaration would be found in the `IfaceDecl` list in `ModIface`.

It is much harder to directly test for type equality on interface types. We can't write a pure function `IfaceType -> IfaceType -> Bool` —in the presence of type synonyms, we may need to do a lookup in a symbol table to determine how the synonym expands. However, it is very easy to update an `IfaceDecl`, since the information associated with it is centralized in one place.

3 Converting between representations

Conversion from graph to interface We convert from the graph representation to the interface representation when we are ready to serialize the results of typechecking to disk. This process is fairly straightforward: all structures in the graph representation record a globally unique name identifying them, so we simply drop the extra information and preserve only the name.

Conversion from interface to graph How do we convert from the interface representation to the graph representation? An important point is that this conversion process doesn't happen in isolation. You can't just simply convert an `IfaceType` into a `Type`: where should a `Name` reference be resolved to? You need a symbol table—one which records the graph nodes of any `Name` you may refer to. Thus, this conversion is always done with respect to some graph; GHC largely assumes that there is only one such graph, and that it is populated with all dependencies. GHC gives this illusion by lazily *loading* declarations into the graph when needed; indeed, this is the usual reason an interface is being converted into graph representation.

In the GHC codebase, the process of converting from interfaces to the graph representation is called *type checking* (though this is not a very good name, since this process never fails—the interface file is assumed to be well-typed).

4 Handling hs-boot files

`hs-boot` files pose a special problem for the graph representation, because they require an *update* to the graph representation. For example, suppose we are typechecking the source files `A.hs-boot`, `B.hs` and `A.hs`:

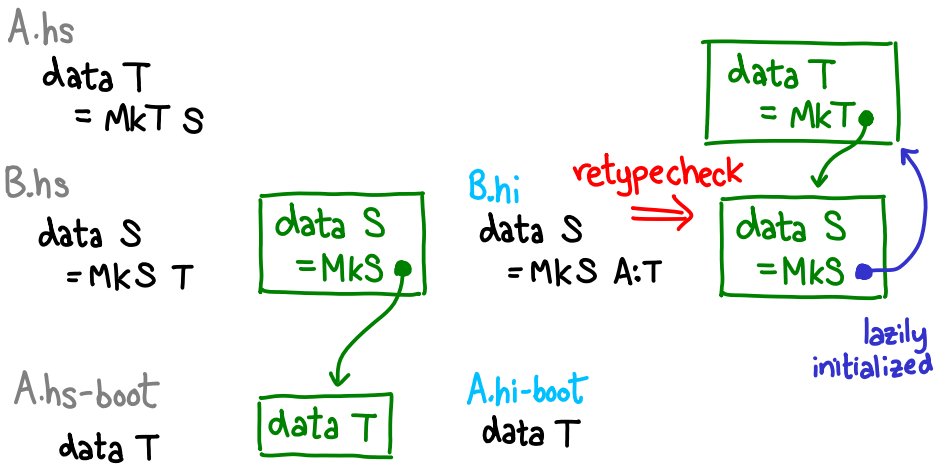
1. We typecheck `A.hs-boot`, producing an *incomplete* graph for the types and values defined in this file. This graph is incomplete because its type constructors may lack information about data type constructors (the user

defined an abstract data type); similarly, its values may lack unfoldings (there is no source code in an `hs-boot` file).

2. We typecheck `B.hs`. The graph we build for this module refers to the incomplete graph from `A.hs-boot`.

3. We typecheck `A.hs`, producing the complete graph for these types and values, while simultaneously updating the references on all the modules from step (2) to point to the new, complete graph. This update is called *re-typechecking*, for reasons that will become clear soon.

The evolution of the graph representation is shown diagramatically below. In the first column, we have the source code being compiled (bottom-first). We typecheck `A.hs-boot` and `B.hs`, forming an incomplete graph (second column), as well as some interface files (third column). When we finally typecheck `A.hs`, we simultaneously retypecheck the interface file `B.hi`, allowing us to tie the knot between the graph representations for `T` and `S`, discarding the old graph (which is now out of date.)



Tying the knot is easy to do if we have an up-to-date copy of `A.hi` in hand (not shown), but what if we are typechecking `A.hs` to generate the interface file in the first place? Here lies a dark and twisty corner of GHC's implementation: we need the updated graph for `data S` in order to typecheck `A.hs`, but prior to typechecking `A.hs`, we don't actually have an object representing `data T`.

The solution is to be lazy: we just need to delay committing to the destination of `data S`'s pointer until we have created the graph node for `data T`. Thus, the pointer is initialized with an unsafe thunk which queries the local type environment for the necessary graph node when it is forced. This is very delicate: if we force the thunk too early, typechecking may not have built the structure we

need and we'll get an error. This is a big source of bugs![2] In fact, `ghc --make` doesn't tie the knot at all; the behavior in these two cases is inconsistent.

5 Graph well-formedness and hs-boot

When studying type-checking algorithms, one is usually concerned with the properties one gets when the provided programs are *well-typed*. But of course, real world programs are often ill-typed, and any real implementation also cares about the behavior of their type-checker on these programs. Here is one *property* which we would like our graph representation to uphold: *all types in the graph are well-formed*, no matter what inputs the user gives us. If a user provides us something bad, we should refuse to build a graph for it.

Unfortunately, this property is not upheld by the current `hs-boot` implementation. Consider this example:

```
-- A.hs-boot
module A where
    data T
    f :: T -> T
-- B.hs
module B where
    import {-# SOURCE #-} A
-- A.hs
module A where
    import B
    data T a = T a
    f :: T a -> T a
    f x = x
```

GHC will stodgily report that the `hs-boot` file's `f :: T -> T` does not match `f :: T a -> T a`. Worse yet, due to the knot tying trick, the `T` in `f :: T -> T` refers to the type constructor for `data T a = T a`; i.e., the type is ill-kinded! The problem is that we must commit to knot-tying `T` to the type presently being typechecked prior to knowing whether or not the `T`'s kind even matches the kind in the `hs-boot` file.

Worse yet, you can convince GHC to construct a type synonym loop and then infinite loop:

```
-- A.hs-boot
module A where
data S
type R = S
-- B.hs
module B (module A, module B) where
```

[2] `https://ghc.haskell.org/trac/ghc/search?q=tcifaceglobal`

```
import {-# SOURCE #-} A
type U = S
-- A.hs
module A where
import qualified B
type S = B.R
type R = B.U
```

Here, the type synonym `S` in `A.hs` is eagerly wired up with the occurrences of `S` in `B.hi`, even though a type synonym is not a valid replacement for an abstract data type.

So what should we do? There seem to be two possibilities:

Accept that sometimes the graph will be malformed We could say that there is no problem with GHC today; sometimes the graph will be ill-kinded or the compiler will loop but the cure is worse than the disease. Before we typechecking finishes, we will discover the problem and bale out (or infinite loop, as the case may be).

Don't tie the knot until after all typechecking is finished Intuitively, the reason why we build the bad graph is because we eagerly rebuild the graph to point to the graph we are typechecking, before we even know whether or not it makes sense or not. So an alternate solution is to *not* tie the knot until we know if the modification makes sense. The very simplest implementation of this strategy would be to just turn off all knot tying, and retypecheck only after typechecking is completely finished.

There is a decent amount of evidence that this would be a good idea:

- `ghc --make`, by far the most used mode of GHC, doesn't correctly tie the knot, and no one has really complained; subsequently, some bugs[3][4] don't affect `ghc --make` (but do affect `ghc -c`).
- When Simon Marlow introduced the knot-tying code for one-shot mode in commit `d83e1ac43a` to fix a subtle "bug", he observed that this bug didn't actually break real code, it just caused some fingerprints to wobble.
- Tying the knot for `Id`s is not very useful, because the optimizer is going to update the unfoldings, etc.

One downside is that without tying the knot, in a hypothetical extension of `hs-boot` files to support implement abstract data types with type synonyms, we encounter the double vision problem:

```
-- A.hs-boot
module A where
```

[3] https://ghc.haskell.org/trac/ghc/ticket/12042
[4] https://ghc.haskell.org/trac/ghc/ticket/10083

```
    data A
-- B.hs
module B where
    import {-# SOURCE #-} A
    f :: A -> A
    f x = x
-- A.hs
module A where
    import B
    type A = Int
    y = f (2 :: Int) -- error!
```

Although we have defined `type A = Int`, `y` will fail to typecheck because the `TyCon` in `f`'s type is abstract and doesn't contain the necessary unfolding.

6 The operations Backpack requires

Let us now characterize the operations Backpack requires on GHC's representations of types. For concreteness, we'll consider how to typecheck the following include statement from Backpack'14:

```
package p where
    A :: [ data T ]
    B :: [ import A; data S = MkS T ]
package q where
    A = [ data T = MkT ]
    B :: [ data S; f :: S -> S ]
    include p -- *
```

The procedure is as follows:

1. The package we are including is associated with some module types which are universally quantified over the identities of its holes. We first apply a **substitution** to these module types, updating these identities to the actual names the holes are instantiated with. In the above example, we substitute `T` in `p` so that it refers to the `T` implemented in `q`. (In Backpack'14, this would be substituting $\beta_{AT} \mapsto \nu_A$; in Backpack'16 this would be substituting $\{\texttt{A.T}\} \mapsto \texttt{q}[\texttt{B} = \langle \texttt{B} \rangle]\texttt{:A.T}$.)

2. The resulting module types are **merged** into the existing types in the context. In the above example, the `data S` from `q`'s signature is merged with `data S = MkS T` from `p`'s signature. This merge operation may fail, e.g., if the types or kinds of the merged declarations do not match; thus, there is implicitly a **type matching** operation which tests if the merge will be successful. (The reason for this separation will become clear shortly.)

So, which representations are most suitable for these operations?

- **Substitution** is reasonably simple enough to carry out on types in either representation. However, substituting over type constructors is most easily done in the *interface representation*, because it represents an update to information which may be embedded elsewhere in the graph representation.
- **Type matching** involves type equality tests, thus it should be done in the *graph representation*.
- **Merging** can cause the information for type constructors to be updated; thus, it should be done in the *interface representation*.

The technical difficulty arises from the fact that, although most operations are most naturally carried out on the interface representation, to test for type matching, we need to convert to the graph representation so that we can test for type equality.