HIE files in GHC 8.8

Zubin Duggal

A brief history

- I started working on Haskell IDE Engine in 2017 as part of Summer of Code
- Came up with .hie files as a way to make HIE work better
  - Implemented in Summer of Code 2018
  - First proof-of-concept application was Haddocks hyperlinked sources
  - Merged in GHC 8.8 (released in Sept 2019) with a lot of help from Alec Theriault
- HIE files now power most of the code navigation in `ghcide` and `haskell-language-server`
  - Go to definition/type definition
  - Type on hover
  - In-file references
  - Scope aware completions
  - Project-wide references are WIP

Some problems faced by Haskell tooling

- Need to recompile source to get access to semantic information
- There is no way to persist such information
- So it is completely unavailable for code that doesn't compile.
- Setting up a GHC session that can load what you want it to load is hard
  - Ecosystem fragmentation - Cabal/Stack/Hadrian/Nix/Bazel/Obelisk etc.
  - Your tooling has to hook into the build system somehow
- No reasonable way to get information about code in dependencies
  - The above problem is compounded - now you need know how to compile each and every dependency
  - Both Stack and Cabal don't keep the original source files around, only final build products like `.o` and `.hi` files
- Complexity and churn in internal GHC representation add a lot of maintenance burden

HIE Files to the rescue!

Idea: When GHC is done with typechecking, serialize some of the output of the renamer and typechecker to the disk, so that tooling can easily access it.

Now, we don't need deep hooks into the build system, just need to instruct it to compile everything with an additional GHC option.

What can HIE Files do for us? (Demo)

- `ghcide` with `hiedb`
  - Hover, go to definition, references
  - Scope-aware completions for local variables
- Haddock's hyperlinked source (with jump to instance declaration sneak preview)
- Weeder (dead code elimination)
- Stan (static analysis/linting)

Lifecycle of a Haskell source file

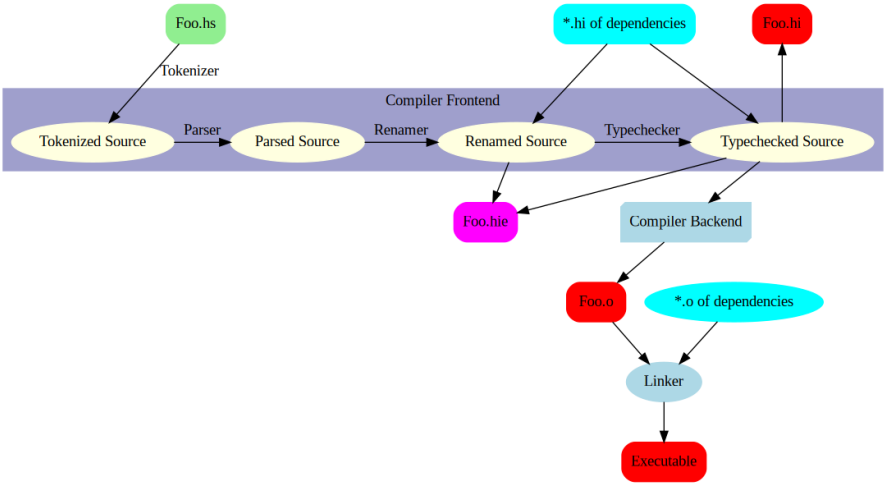


Figure 1: The GHC compilation pipeline

Hooking into the correct place

- The renamer is responsible for resolving names
- The typechecker is responsible for assigning types to expressions and solving constraints
- We hook into the results of both the renamer and typecheker, and traverse the ASTs outputted to gather the info we need for our HIE file

What information do we save?

The GHC AST is **large**!

```
data HsExpr p
  = HsVar (XVar p) (Located (IdP p))
  | HsUnboundVar (XUnboundVar p) UnboundVar
  | HsConLikeOut (XConLikeOut p) ConLike
  | HsRecFld (XRecFld p) (AmbiguousFieldOcc p)
  | HsOverLabel (XOverLabel p) (Maybe (IdP p)) FastString
  ...
```

`HsExpr` alone has more than 30 constructors!

Moreover, the internal representation is constantly changing as features are added or as the code is refactored.

We don't want to burden tooling developers with all this complexity and churn.

So, we want a *simplified* representation of the internal ASTs used by the GHC frontend

Furthermore, it should:

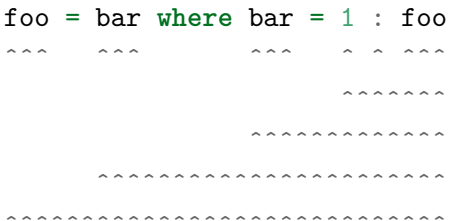
- Be relatively stable
- Be easy to consume
- Map directly to the source code the developer wrote
  - It should be simple to pick a point in the file and answer the following questions:
    - What is this? (Class declaration, binding, operator, variable usage, pattern, etc.)
    - What is its type?
    - Where does it come from?
    - Where else is it used?
  - GHC Core is not suitable for this reason, since it it is too far removed from Surface Haskell

Contents of .hie files

- The original source of the file we compiled
- An interval tree, where each interval corresponds to a span in the source

```
data HieAST =
  Node { nodeInfo :: NodeInfo ty
       , nodeSpan :: RealSrcSpan
       , nodeChildren :: [HieAST ty]
       }
```

Structure of the HieAST



Information captured in nodes

- The type(s) assigned by GHC to this Node, if any
- The name of the GHC AST constructors that this Node corresponds to
- The identifiers that occur at this span
- Information about the identifiers, like:
  - Are they names or modules
  - Their type, if any
  - The context in which they occur: Are they being defined or used, imported, exported, declared etc..
  - If they are being defined, the full span of the definition, and an approximated scope over which their definition can be used.
- Types (stored in a hash consed representation)

Generating .hie files

- Pass the option `-fwrite-ide-info` to ghc to generate them next to .hi/.o files
- Can control path with `-hiedir`
- Cabal/Stack and other build tools need to learn to manage these.

Put the following in your `cabal.project`:

```
package *
  ghc-options: -fwrite-ide-info -hiedir /some/directory/here
```

Consuming .hie files

```
readHieFile :: NameCacheUpdater -> FilePath -> IO HieFileResult
-- ^ Needs to update a NameCache so it can
-- play nice with existing GHC sessions
generateReferencesMap
  :: HieASTs a
  -> M.Map Identifier [(Span, IdentifierDetails a)]
selectSmallestContaining
  :: Span -> HieAST a -> Maybe (HieAST a)
selectLargestContainedBy
  :: Span -> HieAST a -> Maybe (HieAST a)
```

.hie files vs .hi files

- `.hi` files contain the information GHC needs to typecheck other files
  - The functions exported and their types
  - any types, classes defined
  - other stuff like fixities, unfoldings of inlinable functions etc.
- `.hie` files contain information about the source code that developers/tooling authors would like
  - The structure of the AST, along with the type of each node in the tree
  - For each symbol that occurs in the program, along with its source span:
    - The kind of thing (is it a `data`, `class`, value etc)
    - Its type (if it has one)
    - The parts of the source where it is in scope

hiedb - Index and search all of your code!

`hiedb` is a library and command line tool to index and query HIE files

- Index .hie files to get reference information for your whole source tree!
- Extremely fast searching and indexing thanks to SQLite
- Supports many project-wide queries on .hie files
  - References (including references to all variables of a given type)
  - Call graph construction
  - Dead code detection
  - Definitions
  - Search
- Integration with `ghcide` is close to ready
- Can index your dependencies too!

Coming in GHC 9.0: Demystifying typeclass evidence with HIE files

- Typeclass evidence is a bit opaque
  - GHC implicitly generates and inserts evidence for solved constraints in the program
  - However, there is no easy way for the developer to know how exactly GHC did this
  - This is frustrating, since the compiler is essentially writing parts of our program for us, yet we have no way of knowing what it has written.
- We now record information about typeclass evidence into .hie files
- Will allow us to answer questions like:
  - What instances are being used at this point? (Jump to implementation)
  - How are these instances composed?
  - Where all in my program is this instance being used?

```haskell
class C a where
  f :: a -> Char

instance C Char where
  f x = x

instance C a => C [a] where
  f x = 'a'

foo :: C a => a -> Char
foo x = f [x]
--        ^ (31,9)
```

```
═══════════════════════
At point (31,9), we found:
═══════════════════════

┌ $dC at HieQueries.hs:31:1-13, of type: C [a]
│     is an evidence variable bound by a let, depending on: [$fC[], $dC]
│           with scope: LocalScope HieQueries.hs:31:1-13
│           bound at: HieQueries.hs:31:1-13
│     Defined at <no location info>
└
│
+- ┌ $fC[] at HieQueries.hs:27:10-21, of type: forall a. C a ⇒ C [a]
│  │     is an evidence variable bound by an instance of class C
│  │           with scope: ModuleScope
│  │
│  │     Defined at HieQueries.hs:27:10
│  └
│
`- ┌ $dC at HieQueries.hs:31:1-13, of type: C a
   │     is an evidence variable bound by a HsWrapper
   │           with scope: LocalScope HieQueries.hs:31:1-13
   │           bound at: HieQueries.hs:31:1-13
   │     Defined at <no location info>
   └
```

Future developments

Merge with .hi files

- Tooling integration for free
- One place to look for everything GHC knows about haskell source
- Extensible HI files proposal

Capturing more type information

- Typechecked AST doesn't actually contain type information for all nodes
- Until now every tool desugared *every single subexpression* in the AST to get its type
- Because of this, we only save type information in the `HieAST` when the node has it available or it is a leaf node
- WIP patch to add an efficient means to get the type of every expression.

Capturing more type information

- Typechecked AST doesn't actually contain type information for all nodes
- Until now every tool desugared *every single subexpression* in the AST to get its type
- Because of this, we only save type information in the `HieAST` when the node has it available or it is a leaf node
- WIP patch to add an efficient means to get the type of every expression.

Links

- hiedb: https://github.com/wz1000/HieDb
- More information on .hie files: https://gitlab.haskell.org/ghc/ghc/wikis/hie-files
- PR to integrate hiedb with ghcide: https://github.com/haskell/ghcide/pull/898