Haskell Language Server

Zubin Duggal

HLS Architecture

- hie-bios to set up GHC session to interact with build tools
- the GHC API to be able to load and analyse code
- hiedb to manage persistent information across restarts
- lsp library to handle speaking the protocol and communicate with editors
- HLS graph based build system like infrastructure to compute results and manage state
- Facilities for responding to user requests quickly and in the face of failures
- Extensible plugin architecture

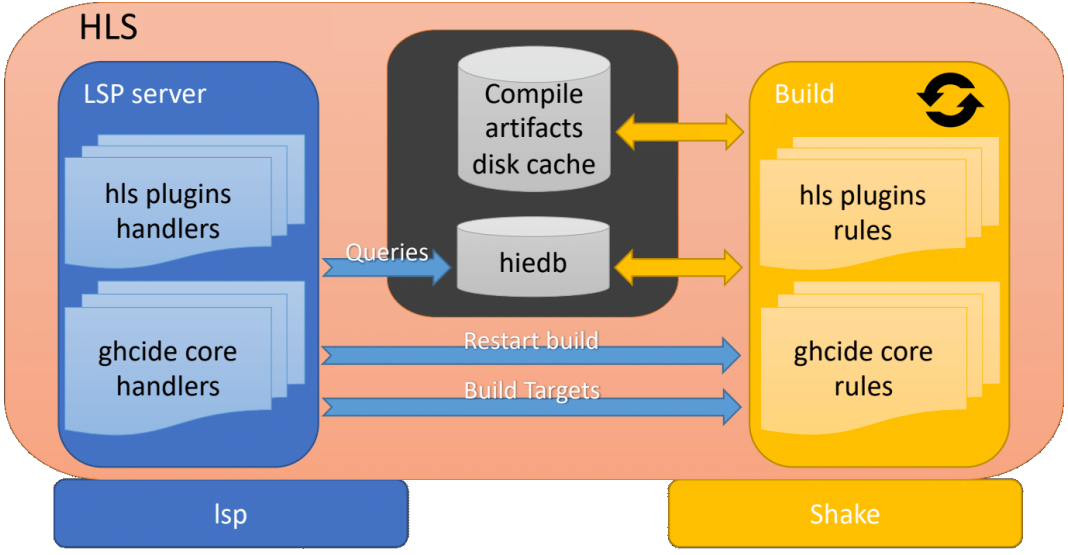


Figure 1: HLS Architecture

Writing a Plugin

```
data PluginDescriptor (ideState :: *) = PluginDescriptor
  { pluginId                   :: !PluginId
  , pluginPriority             :: Natural
  , pluginRules                :: !(Rules ())
  , pluginCommands             :: ![PluginCommand ideState]
  , pluginHandlers             :: PluginHandlers ideState
  , pluginConfigDescriptor     :: ConfigDescriptor
  , pluginNotificationHandlers :: PluginNotificationHandlers ideState
  , pluginModifyDynflags       :: DynFlagsModifications
  , pluginCli                  :: Maybe (ParserInfo (IdeCommand ideState))
  , pluginFileType             :: [T.Text]
  }
```

Hooking into LSP

- Usually, a plugin will wait for a notification or request to arrive from the editor before being triggered.
- A request has a response and can be cancelled
  - Plugin code has to be exception-safe and shouldn't (uninterruptibly) block
  - Responses for requests handled by multiple plugins are combined
- A notification is the client updating us about the state of the world
  - A notification handler cannot be cancelled
  - Plugin notification handlers are called in order of `pluginPriority`
- Both the server and the client have their own requests and notifications methods that they can invoke/handle

Communicating with the editor

```
type PluginMethodHandler a m =
    a -> PluginId -> MessageParams m
  -> LspM Config (Either ResponseError (ResponseResult m))
mkPluginHandler ::
     PluginRequestMethod m
  => SClientMethod m -> PluginMethodHandler ideState m
  -> PluginHandlers ideState
type PluginNotificationMethodHandler a m
  = a -> VFS -> PluginId -> MessageParams m -> LspM Config ()
mkPluginNotificationHandler ::
     PluginNotificationMethod m
  => SClientMethod m -> PluginNotificationMethodHandler ideState m
  -> PluginNotificationHandlers ideState
```

These are `Monoids` and can be combined using semigroup operations

Sending requests and notifications to the client

```
sendRequest :: MonadLsp config f
  => SServerMethod (m :: Method FromServer Request)
  -> MessageParams m
  -> (Either ResponseError (ResponseResult m) -> f ())
  -> f (LspId m)

sendNotification :: MonadLsp config f
  => SServerMethod (m :: Method FromServer Notification)
  -> MessageParams m -> f ()
```

An IDE as a build system

- We need to manage a lot of inter-dependent state
- Changes invalidate some of the state but not others
- State updates are often interrupted by new changes
- Solution: Use a build system like architecture (inspired by Shake) and get incremental rebuilds and early cutoff
- On any change (LSP notification), we update the state, cancel any in-progress builds, and kick off a new build
- Why not use Shake?
  - We did! But we changed to a custom re-implementation more suited for our purposes
  - Unlike Shake, we perform most of the building entirely in-memory
  - We also have guarantees to be notified of any changes to the world, so instead of pessimistically checking everything for freshness, we can start rebuilding only the things that we know have changed, and proceed by rebuilding only reverse-dependencies.

The build system in action

```
data TypeCheck = TypeCheck
type instance RuleResult TypeCheck = TcModuleResult
typeCheckRule :: Recorder (WithPriority Log) -> Rules ()
typeCheckRule recorder = define (...) $ \TypeCheck file -> do
    pm <- use_ GetParsedModule file
    hsc  <- hscEnv <$> use_ GhcSessionDeps file
    addUsageDependencies $ typecheckModule hsc pm
  where addUsageDependencies a = do
          r@(_, mtc) <- a
          forM_ mtc $ \tc -> do
            used_files <-
              liftIO $ readIORef $ tcg_dependent_files $ tmrTypechecked tc
            void $ uses_ GetModificationTime
                         (map toNormalizedFilePath' used_files)
          return r
```

Calling into the build system

```
runAction    :: String -> IdeState -> Action a -> IO a
use          :: IdeRule k v => k -> NormalizedFilePath
             -> Action (Maybe v)
useWithStale :: IdeRule k v => k -> NormalizedFilePath
             -> Action (Maybe (v, PositionMapping))

runIdeAction     :: String -> ShakeExtras -> IdeAction a -> IO a
useWithStaleFast :: IdeRule k v => k -> NormalizedFilePath
                 -> IdeAction (Maybe (v, PositionMapping))
```

use and uses

```
use   :: IdeRule k v
      => k -> NormalizedFilePath -> Action (Maybe v)
use_  :: IdeRule k v
      => k -> NormalizedFilePath -> Action v
uses  :: (Traversable f, IdeRule k v)
     => k -> f NormalizedFilePath -> Action (f (Maybe v))
uses_ :: (Traversable f, IdeRule k v)
     => k -> f NormalizedFilePath -> Action (f v)
```

- Use these when you need to guarantee results are up to date
- They block until a fresh result is computed
- It is best to limit these to the body of rule definitions, and for responses where correctness is critical, like edits and refactorings.
- Multiple `use` calls in an `Action` will access a consistent view of the state.

useWithStale and PositionMappings

```
useWithStale :: IdeRule k v => k -> NormalizedFilePath
             -> Action (Maybe (v, PositionMapping))
useWithStale_, usesWithStale, usesWithStale_ :: ...
fromCurrentPosition :: PositionMapping -> Position -> Maybe Position
toCurrentPosition :: PositionMapping -> Position -> Maybe Position
```

- These will also block until a fresh result is computed
- But if the rule fails, they will return an old result for the rule, if one was previously computed.
- They also return a `PositionMapping` specifying how to map between positions in the old version of the document (for which the rule succeeded) and the current version of the document (for which the rule failed)
- Use these for responding to user requests even in the face of failure

useWithStaleFast and IdeActions

- This function (almost) never blocks, and is meant to give you the most recently computed version of the results.
- Queues up a refresh for the rule which will be evaluated asynchronously
- Use this for quickly responding to user requests when correctness/freshness of results is not critical
- Examples: Hover, go to definition etc.
- Multiple calls to `useWithStaleFast` in sequence might result in an inconsistent view of the state.

Defining your own Rules

```
type IdeResult v = ([FileDiagnostic], Maybe v)
data RuleBody k
  = Rule (k -> NormalizedFilePath -> Action (Maybe BS.ByteString, IdeResult v))
  | RuleNoDiagnostics (k -> NormalizedFilePath -> Action (Maybe BS.ByteString, Maybe v))
  | RuleWithCustomNewnessCheck
    { newnessCheck :: BS.ByteString -> BS.ByteString -> Bool
    , build :: k -> NormalizedFilePath -> Action (Maybe BS.ByteString, Maybe v)
    }
  | RuleWithOldValue (k -> NormalizedFilePath -> Value v
                        -> Action (Maybe BS.ByteString, IdeResult v))

defineEarlyCutoff
    :: IdeRule k v
    => Recorder (WithPriority Log)
    -> RuleBody k v
    -> Rules ()
```

hiedb and persistent state

We want restarts of the editor/language server to be fast, and we don't want to recompute everything

- GHC build artifacts are cached in `~/.cache/ghcide`
- We heavily rely on GHC's recompilation avoidance
- For information that is not persistent in GHC's persistent state (interface files), we use a SQLite database called hiedb
- `hiedb` has tables for references, definitions, exports and more
- We ensure it is up to date, but this is done asynchronously
- Most of the information is generated by indexing .hie files, but some is also manually computed by ghcide and inserted into a table.

GHC version support and CPP

- To use HLS it must be compiled with the exact same version of GHC that your project is using
- We support a lot of different GHC versions (current 8.10 onwards)
- This requires a *lot* of CPP to ensure compatibilty
- We try to maintain a compatibilty layer in the `Development.IDE.GHC.Compat(.*)` modules
- Most CPP should be restricted to these modules
- However, when there are major logic changes in between versions, it is better to have the CPP inline where it is being used

GHC Exactprint

- We use ghc-exactprint for providing refactorings and edits for various interactions and requests
- It uses annotations in the AST to provide faithful pretty printing support
- Earlier versions of ghc-exactprint (prior to GHC 9.2) used out of tree annotation, so we use copious amounts of CPP to support those

Darks Arts of the GHC API

GHC provides mechanisms to hook into various parts of the compiler, in the form of `Plugins` and `Hooks`.

Various parts of HLS use these to access and replace parts of the compiler normally not accessible to API users.

```
data Hooks = Hooks
  { ...
  , hscCompileCoreExprHook :: !(Maybe (HscEnv -> SrcSpan -> CoreExpr
                                  -> IO (ForeignHValue, [Linkable], PkgsLoaded)))
  , runMetaHook            :: !(Maybe (MetaHook TcM))
  ...
  }
data Plugins = Plugins
  { staticPlugins :: ![StaticPlugin]
  ...
  }
```

Type information on hover for splices

```
$(deriveJSON 'Foo)
==>
instance ToJSON Foo where ...
```

- The typechecked AST we get back from GHC after typechecking has all splices expanded out
- We would still like to provide types/documentation on hover, go to definition support and other features for symbols used in the splice pre-expansion
- We can install a `runMetaHook` to capture the AST of the splice prior to expansion and inject it into the AST so that these features work
- We store the AST in an IORef that we read out after typechecking returns

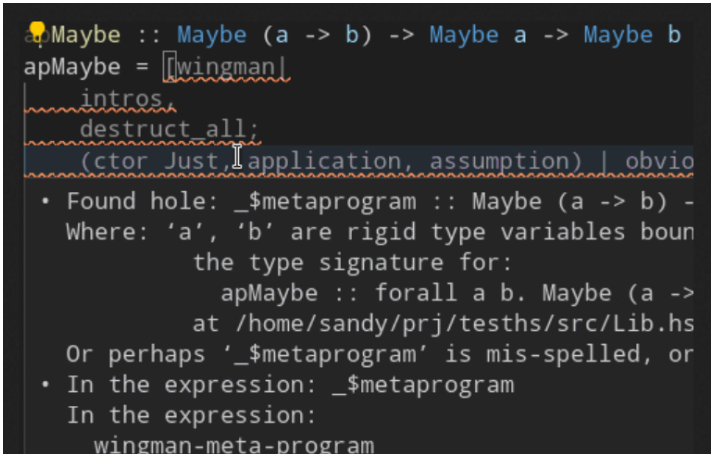
Lazy on-demand generation of bytecode

- Usually we don't want to desugar/compile code, we only want to run the compiler frontend
- But when Template Haskell is used, we must desugar code for all dependencies/imports so that we can evaluate splices
- However, most dependencies/imports are usually not used in a splice!
- We still have to desugar all dependencies as we don't want to keep the AST around for all of them as this uses a lot of memory, and we don't want to retypecheck them if it turns out we actually need to compile them
- But we can avoid some wasted work by only compiling the desugared code to bytecode for dependencies that are *actually* used in a splice
- To do this, we hook into `hscCompileCoreExprHook` to intercept splices just before they are compiled to bytecode and executed.
- We analyse the expression to compute all things it depends on, and requst the build system to compile *only* those modules to bytecode
- This results in a nice performance improvement for projects with a lot of TH.

Adding custom sytax using plugins (Wingman)

Sometimes, LSP is not expressive enought for the kinds of interactions we would like to support.

For instance, the Wingman plugin provides tacting metaprogramming using special syntax added using a GHC plugin.



Links

- LSP specification
- Blog post on recompilation aviodance for TH