Building an Integrated Development Environment (IDE) on top of a Build System

The tale of a Haskell IDE

Neil Mitchell
Facebook
ndmitchell@gmail.com

Moritz Kiefer
Digital Asset
moritz.kiefer@purelyfunctional.org

Pepe Iborra
Facebook
pepeiborra@gmail.com

Luke Lau
Trinity College Dublin
luke_lau@icloud.com

Zubin Duggal
Chennai Mathematical Institute
zubin.duggal@gmail.com

Hannes Siebenhandl
TU Wien
hannes.siebenhandl@posteo.net

Javier Neira Sanchez
UNED, Spain
atreyu.bbb@gmail.com

Matthew Pickering
Well-Typed
matthewtpickering@gmail.com

Alan Zimmerman
Facebook
alan.zimm@gmail.com

Abstract

When developing a Haskell IDE we hit upon an idea – why not base an IDE on an build system? In this paper we'll explain how to go from that idea to a usable IDE, including the difficulties imposed by reusing a build system, and those imposed by technical details specific to Haskell. Our design has been successful, and hopefully provides a blue-print for others writing IDEs.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**.

ACM Reference Format:
Neil Mitchell, Moritz Kiefer, Pepe Iborra, Luke Lau, Zubin Duggal, Hannes Siebenhandl, Javier Neira Sanchez, Matthew Pickering, and Alan Zimmerman. 2020. Building an Integrated Development Environment (IDE) on top of a Build System: The tale of a Haskell IDE. In *Proceedings of Implementation and Application of Functional Languages (IFL'20).* ACM, New York, NY, USA, 12 pages.

1 Introduction

Writing an IDE (Integrated Development Environment) is not as easy as it looks. While there are thousands of papers and university lectures on how to write a compiler, there is much less written about IDEs. We embarked on a project to write a Haskell IDE (originally for the GHC-based DAML language [Digital Asset 2021]), but our first few designs failed. Eventually, we arrived at a design where the heavy-lifting of the IDE was performed by a *build system.* That idea worked well, and is the subject of this paper.

Over the past two years we have continued development and found that the ideas behind a build system are both applicable and natural for an IDE. The result is available as a project named *Haskell Language Server*[1] (HLS)[2].

In this paper we outline the core of our IDE §2, how it is fleshed out into an IDE component §3, and then how we build a complete IDE §4. We look at where the build system both helps and hurts §5. We then look at the ongoing and future work §6 before comparing to related work §7 and concluding §8.

2 Design

In this section we show how to implement an IDE on top of a build system. First we look at what an IDE provides, then what a build system provides, followed by how to combine the two.

2.1 Features on an IDE

To design an IDE, it is worth first reflecting on what features an IDE provides. In our view, the primary features of an IDE can be grouped into three capabilities, in order of how essential they are to a users experience:

Errors/warnings The main benefit of an IDE is to get immediate feedback as the user types. That involves producing errors/warnings on every keystroke. In a language such as Haskell, that involves running the parser and type checker on every keystroke. From these errors/warnings lots of additional features can be built, like the ability to automatically remedy common errors (e.g. inserting missing `LANGUAGE` extension pragmas).

Hover/goto definition The next most important feature is the ability to interrogate the code in front of

[1]https://github.com/haskell/haskell-language-server
[2]Within HLS there lives an internal library named *Ghcide* [Mitchell et al. 2020], which integrates the build system, and used to be a standalone IDE – in this paper we use the term HLS to include the Ghcide library.

IFL'20, September 2–4, 2020, Online
2020.

you. Ways to do that include hovering over an identifier to see its type, and clicking on an identifier to jump to its definition. In a language like Haskell, these features require performing name resolution. Many other features build on this same information, for example auto-completions (which HLS provides as a plugin, §4), case splitting and the ability to automatically insert type signatures.

Find references Finally, the last feature is the ability to find where a symbol is used. This feature requires an understanding of all the code, and the ability to index outward. Lots of refactoring tools require similar knowledge to be available.

While debugging is often part of a fully featured IDE, it is not the focus of this paper, so we mostly ignore it, other than as future work in §6.5.

The design of Haskell is such that to type check a module requires to get its contents, parse it, resolve the imports, type check the imports, and only then type check the module itself. If one of the imports changes, then any module importing it must also be rechecked. That process can happen once per user character press, so is repeated incredibly frequently.

Given the main value of an IDE is the presence/absence of errors, the way such errors are processed should be heavily optimised. In particular, it is important to hide/show an error as soon as possible. Furthermore, errors should persist until they have been corrected.

2.2 Features of a build system

The GHC API is a Haskell API for compiling Haskell files, using the same machinery as the GHC compiler [The GHC Team 2021]. Therefore, to integrate smoothly with the GHC API, it is important to choose a build system that can be used as a Haskell library. Furthermore, since the build graph is incredibly dynamic, potentially changing on every key stroke, it is important to be a monadic build system [Mokhov et al. 2018, §3.5]. Given those constraints, and the presence of an author in common, we chose to use Shake [Mitchell 2012].

The Shake build system is fully featured, including parallelism, incremental evaluation and monadic dependencies. While it has APIs to make file-based operations easy, it is flexible enough to allow defining new types of rules and dependencies which do not use files. At its heart, Shake is a key/value mapping, for many types of key, where the type of the value is determined by the type of the key, and the resulting value may depend on many other keys.

2.3 An IDE based on a build system

Given the IDE and build system features described above, there are some very natural combinations. The monadic dependencies are a perfect fit. Incremental evaluation and parallelism provide good performance. But there are a number of points of divergence which we discuss and overcome below.

2.3.1 Restarting. When a user changes a file (e.g. on every keystroke), if there is still ongoing work (e.g. type-checking a file), there is a choice to be made: do you abort the ongoing work, wait for the ongoing work to complete before dealing with the change, or do both simultaneously. Faster feedback is more useful to the user, so we should prefer dealing with the change immediately. Machine resources are finite, so running lots of things simultaneously is likely to slow everything down and have a corresponding memory cost. However, some jobs take 10 seconds (e.g. type checking a big module), and if aborted every keystroke might not produce results until the user pauses.

Given the purpose of an IDE is to provide fast feedback, and resource usage (particularly memory) is already a concern (see §5.7), we opt to abort all ongoing work on every change. The downside is that big tasks might repeat the beginning of their work many times, or at worst never complete, although in practice we haven't noticed any significant problems. We discuss possible remedies in §5.3.

A Shake build can be interrupted at any point, and we take the approach that whenever a file changes, e.g. on every keystroke, we interrupt the running Shake build and start a fresh one. While that approach is delightfully simple, it has some engineering concerns. We interrupt using asynchronous exceptions [Peyton Jones 2001], but lots of Haskell code is not properly designed to deal with such exceptions. We had to fix a number of bugs in Shake and other libraries and are fairly certain some still remain.

2.3.2 Errors. In normal Shake execution an error is thrown as an exception which aborts the build. However, for an IDE, errors are a common and expected state. Therefore, it is necessary to make errors first class values. Concretely, instead of the result of a rule such as type checking being a type checked module, we use:

```
([Diagnostic], Maybe TcModuleResult)
```

Where `TcModuleResult` is the type checked module result as provided by the GHC API. The list of diagnostics stores errors and warnings which can occur even if type checking succeeded. The second component represents the result of the rule with `Nothing` meaning that the rule could not be computed either because its dependencies failed, or because it failed itself.

In addition, when an error occurs, it is important to track which file it belongs to, and to determine when the error goes away. To achieve that, we make all Shake keys be a pair of a phase-specific type alongside a `FilePath`. So a type-checked value is indexed by:

```
(TypeCheck, FilePath)
```

where `TypeCheck` is isomorphic to `()`.

The second component of the key determines the file the error will be associated with in the IDE. We cache the error per `FilePath` and phase, and when a `TypeCheck` phase for a given file completes, we overwrite any previous type checking errors that file may have had. By doing so, we can keep an up-to-date copy of what errors are known to exist in a file, and know when they have been resolved.

2.3.3 Performance. Shake runs rules in a random order [Mitchell 2012, §4.3.2]. But as rule authors, we know that some steps like type checking are computationally expensive, while others like finding imports (and thus parsing) cause the graph to fan out. Using that knowledge, we can deprioritise type checking to reduce latency and make better use of multicore machines. To enable that deprioritisation, we added a `reschedule` function to Shake, that reschedules a task with a lower priority.

2.3.4 Memory only. Shake usually operates as a traditional build system, working with files and commands. As standard, it stores its central key/value map in a journal on disk, and rereads it afresh on each run. That caused two problems:

1. Reading the journal each time can take as long as 0.1s. While that is almost irrelevant for a traditional build, for an IDE it is excessive. We solved this problem by adding an option to Shake that retains the key/value map in memory and doesn't write to disk, completely eliminating this delay.
2. Shake serialises all keys and values into the journal, so those types must be serializable. While adding a memory-only journal was feasible, removing the serialisation constraints and eliminating all serialisation would require more significant modifications. Therefore we wrote serialisation methods for all the keys. However, values are often GHC types, and contain embedded types such as `IORef`, making it difficult to serialise them. To avoid the need to use value serialisation, we created a shadow map containing the actual values, and stored dummy values in the Shake map.

The design of Shake is for keys to accumulate and never be removed. However, as the IDE is very dynamic, the relevant set of keys may change regularly. Fortunately, the Shake portion of the key/value is small enough not to worry about, but the shadow map should have unreachable nodes removed in a garbage-collection like process (see §5.6).

2.4 Layering on top of Shake

In order to simplify the design of the rest of the system, we built a layer on top of Shake, which provides the shadow map, the keys with file names, the values with pairs and diagnostics etc. By building upon this layer we get an interface that more closely matches the needs of an IDE. Using this layer, we can define the type checking portion of the IDE as:

```
type instance RuleResult TypeCheck =
    TcModuleResult

typeCheck = define $ \TypeCheck file -> do
    pm <- use_ GetParsedModule file
    deps <- use_ GetDependencies file
    tms <- uses_ TypeCheck $
        transitiveModuleDeps deps
    session <- useNoFile_ GhcSession
    liftIO $ typecheckModule session tms pm
```

Reading this code, we use the `RuleResult` type family [Chakravarty et al. 2005] to declare that the `TypeCheck` phase returns a value of type `TcModuleResult`. We then define a rule `typeCheck` which implements the `TypeCheck` phase. The actual rule itself is declared with `define`, taking the phase and the filename. First, it gets the parsed module, then the dependencies of the parsed module, then the type checked results for the transitive dependencies. It then uses that information along with the GHC API session to call a function `typecheckModule`. To make this code work cleanly, there are a few key functions we build upon:

- We use `define` to define types of rule, taking the phase and the filename to operate on.
- We define `use` and `uses` which take a phase and a file (or lists thereof) and return the result.
- On top of `use` we define `use_` which raises an exception if the requested rule failed. In `define` we catch that exception and switch it for `([], Nothing)` to indicate that a dependency has failed.
- Some items do not have a file associated with them, e.g. there is exactly one GHC session, so we have `useNoFile` (and the underscore variation) for these.
- Finally, the GHC API can be quite complex. There is a GHC provided `typecheckModule`, but it throws exceptions on error, prints warnings to a log, returns too much information for our purposes and operates in the GHC monad. Therefore, we wrap it into a API where all inputs and outputs are explicit, with the signature:

```
typecheckModule
 :: HscEnv
 -> [TcModuleResult]
 -> ParsedModule
 -> IO ([Diagnostic], Maybe TcModuleResult)
```

2.5 Stale results

For some operations, getting a result *quickly* is significantly more important than getting a *correct* result. As an example, consider completions – they are only useful if produced almost immediately (before the next keystroke), and a precise answer is nice, but not essential. Our initial approach utilized the `use` function as described in §2.4, but each keystroke necessitates checking the graph is still valid and a reparse of the current file. For completions, that overhead was very noticeable. To address this use case we added a variant of use which can access stale values:

```
useStale :: ...
  => stage -> FilePath
  -> Action (Maybe (v, PositionMapping))
```

The function useStale takes a stage and file, just like normal use, but there are two key differences. Firstly, instead of returning an accurate v, it returns the last successfully computed value of v. The key is only computed afresh if it has never been requested before. A result is only Nothing if *all* previous requests have failed. Secondly, useStale also returns a PositionMapping. The PositionMapping describes how the document has been modified since the result was calculated, allowing operations to map positions from that result to and from the current document – e.g. if a new line is inserted at the beginning of the file, the position mapping will shift all line numbers by one. We also provide a variant of useStale that asynchronously runs the action, so that future stale requests are likely to get a more accurate answer.

This technique provides a significant improvement in the responsiveness, in return for a small sacrifice in correctness. For operations like completions (must be immediate, don't have to be correct), this trade-off was easy to make. For other operations, like hover or go to definition, some changes can have a significant impact (e.g. changing the imports to cause different identifiers to be imported), but most changes are of little consequence, so we use stale information. For actions, like inserting type signatures, incorrect changes made to the users code are very costly, so we do not use stale results.

3 Integration

To go from the core described in §2 to a fully working IDE requires integrating with lots of other projects. In this section we outline some of the most important.

3.1 The GHC API

The GHC API provides access to the internals of GHC and was not originally designed as a public API. This history leads to some design choices where IORef values (mutable references) hide alongside huge blobs of state (e.g. HscEnv, DynFlags). With careful investigation, most pieces can be turned into suitable building blocks for an IDE. Over the past few years the Haskell IDE Engine project [The Haskell IDE Engine Team 2020] has been working with GHC to upstream patches to make more functions take in-memory buffers rather than files, which has been very helpful.

One potentially useful part of the GHC API is the "downsweep" mechanism. In order to find dependencies, GHC first parses the import statements, then sweeps downwards, adding more modules into a dependency graph. The result of downsweep is a static graph indicating how modules are related. Unfortunately, this process is not very incremental, operating on all modules at once. If it fails, the result is a failure rather than a partial success. This whole-graph approach makes it unsuitable for use in an IDE. Therefore, we rewrote the downsweep process in terms of incremental dependencies. The disadvantage is that many things like pre-processing and plugins are also handled by the downsweep, so they had to be dealt with specially. We hope to upstream our incremental downsweep into GHC at some point in the future.

3.1.1 Separate type-checking. In order to achieve good performance in large projects, it is important to cache the results of type-checking individual modules and to avoid repeating the work the next time they are needed, or when loading them for the first time after restarting the IDE. Our IDE leverages two features of GHC that, together, enable fully separate typechecking while preserving all the IDE features mentioned in §2.1

1. Interface files (so called .hi files) are a by-product of module compilation and have been in GHC since the authors can remember. They contain a plethora of information about the associated module. When asking the GHC API to type-check a module M that depends on a module D, one can load a previously obtained D.hi interface file instead of type-checking D, which is much more efficient and avoids duplicating work. Using this file is only correct when D has not changed since D.hi was produced, but happily GHC performs recompilation checks and complains when this assumption is not met.
2. Extended interface files (so called .hie files) are also a by-product of module compilation, recently added to GHC in version 8.8. Extended interface files record the full details of the type-checked AST of the associated module, enabling tools to provide hover and go-to reference functionality without the need to use the GHC API at all. Our IDE mines these files to provide hover and go-to reference for modules that have been loaded from an interface file, and thus not typechecked in the current session.

3.2 Setting up a GHC Session

When using the GHC API, the first challenge is to create a working GHC session. Session construction involves setting the correct DynFlags needed to load and type-check the files in a project. These typically include compilation flags like include paths and what extensions should be enabled, but also includes information about package dependencies, which need to be built beforehand and registered with the command line tool ghc-pkg. Furthermore, these details are all entirely dependent on the build tool: the flags that the build tool Stack passes to GHC to build a project will be different from what the build tool Cabal passes, because each builds and stores package dependencies in different locations and package databases.

Because session creation is so specific to the build tool, setting up the environment and extracting the flags for a Haskell project has traditionally been a very fickle process. A new library called hie-bios [The HIE BIOS Team 2021] was developed to tackle this problem, consolidating efforts into one place. The name comes from the idea that it acts as the first point of entry for setting up the GHC session, much like a system BIOS is the first point of entry for hardware on a computer. Its philosophy is to delegate the responsibility of setting up a session entirely to the build tool — whether that be Cabal, Stack, Hadrian [Mokhov et al. 2016], Bazel [Google 2021], Buck [Facebook 2021] or any other build system that invokes GHC.

The hie-bios library is based around the idea of *cradles* which describe a specific way to set up an environment through a specific build tool. For instance, hie-bios comes with cradles for Stack projects, Cabal projects and standalone Haskell files, but it can interface with other build tools by invoking them and reading the arguments to GHC via `stdout`. These cradles are essentially functions that call the necessary commands on the build tool to build and register any dependencies, and return the flags that would be passed to GHC for a specific file or component. For Cabal and Stack, this information is currently obtained through the `repl` commands. The cradle that should be used for a specific project can be inferred through the presence of build-tool specific files like `cabal.project` and `stack.yaml`. For more complex projects which comprise of multiple directories and packages, the cradles used can be explicitly configured through a `hie.yaml` file to describe exactly what build tool should be used, and what component should be loaded for the GHC session, for each file or directory.

3.2.1 Handling multiple components in one session.

Haskell projects are often separated into multiple packages, and when using Cabal [Jones 2005], a package consists of multiple components. These components might be a library, executable, test-suite or a benchmark. Each of the components might require a different set of compilation options and they might depend on each other. Ideally, we want to be able to use the IDE on all components at the same time, so that features like goto-definition and refactoring work sensibly. Using the IDE on a big project with multiple subprojects should work equivalently to a single component project.

However, the GHC API is designed to only handle a single component at a time. This limitation is currently hard-coded in multiple locations within the GHC code-base. As it can only handle a single component, GHC only checks whether any modules have changed for this single component, assuming that dependencies are stored on disk and will not change during the compilation. However, in our dynamic usage, local dependencies might change!

The same problematic behaviour can be found in everyday usage of an interactive GHC session. Loading an executable into the interactive session, and applying changes to the library the executable depends on, will not cause any recompilation in the interactive session. For any of the changes to take effect, the user needs to entirely shut-down the interactive GHC session and reload it. In the IDE context, if the library component changes the executable component will not be recompiled, as GHC does not notice that a dependency has changed and diagnostics for the executable component become stale.

To work around these limitations, we handle components in-memory and modify the GHC session ad-hoc. Whenever the IDE encounters a new component, we calculate the global module graph of all components that are in-memory. With this graph, we can handle module updates ourselves and load multiple components in a single GHC session. However, such approaches are fighting against GHC, are less efficient than a native GHC approach would be, and there are many bugs in the corner cases – some of which appear impossible to eliminate. The real solution is to extend GHC with multiple home components, which we discuss in §6.3.

3.2.2 Error tolerance.

An IDE needs to be tolerant to errors in the source code, and must continue to aid the developer while the source code is incomplete and does not parse or typecheck, as this state is the default while source code it is being edited. Importantly, the further we can proceed in the compilation pipeline, the more information we can offer the user; e.g. if we can type check some of the file, we can provide better types on hover, even if the whole file does not type check. We employ a variety of mechanisms to achieve this goal:

- GHC has two flags named `-fdefer-type-errors` and `-fdefer-out-of-scope-variables` that turn type errors and out of scope variable errors into warnings, and let it proceed to typecheck and return usable artifacts to the IDE. These flag leads to GHC downgrading the errors produced to warnings, so we must promote such warnings back into errors before reporting them to the user.
- If the code still fails to typecheck (for example due to a parse error, or multiple declarations of a function etc.), we still need to be able to return results to the user. Therefore, we use `useStale` from §2.5 to the most recent successful type checked results, even if it was for an older version of the source.
- We continue to work with the GHC team, suggesting further places where errors can be downgraded to warnings - e.g. the parse error `do a <- x` could be treated as a warning.

3.3 Language Server Protocol (LSP)

In order to actually work as an IDE, we need to communicate with a text editor. We use the Language Server Protocol (LSP) [Microsoft 2021b] for communication, which is supported by most popular text editors and clients, either natively or through plugins and extensions. LSP is a JSON-RPC based protocol that works by sending messages between the editor and a *language server*. Messages are either *requests*, which expect a *response* to be sent back in reply, or *notifications* which do not expect any. For example, the editor (client) might send notifications that some file has been updated, or requests for code completions to display to the user at a given source location. The language server may then send back responses answering those requests and notifications that provide diagnostics.

To bridge the gap between messages and the build graph, we deal with the types of incoming messages differently:

- When a notification arrives from LSP that a document has been edited, we modify the nodes that have changed, e.g., the content of the modified files, and immediately start a rebuild in order to produce diagnostics.
- When a request for some specific language feature arrives, we append a target to the ongoing build asking for whatever information is required to answer that request. For example, if a hover request arrives, we ask for the set of type-checked spans corresponding to that file. Importantly, this operation does not cause a rebuild.
- When the graph computes that the diagnostics for a particular file have changed, we send a notification to the client to show updated diagnostics.

3.4 Testing

Our IDE implements a large part of the LSP specification, and has to operate on a large range of possible projects with all sorts of edge cases. We protect against regressions from these edge cases with a functional test suite built upon the library lsp-test, a testing framework for LSP servers. This lsp-test library acts as a client which language servers can talk to, simulating a session from start to finish at the transport level. The library allows tests to specify what messages the client should send to the server, and what messages should be received back from the server.

Functional testing turns out to be rather important in this scenario as the RPC-based protocol is in practice, highly asynchronous, something which unit tests often fail to account for. Clients can make multiple requests in flight and Shake runs multiple worker threads, so the order in which messages are delivered is non-deterministic. Because of this fact, a typical test might look like:

```
test :: IO ()
test = runSession "hls" fullCaps "test" $ do
  doc <- openDoc "Foo.hs" "haskell"
  skipMany anyNotification
  let prms = DocumentSymbolParams doc
  rsp <- request TextDocumentDocumentSymbol prms
  liftIO $ rsp ^. result `shouldNotSatisfy` null
```

In this session, lsp-test tells HLS to open up a document, and then ignore any notifications it may send with `skipMany anyNotification`. A session is actually a parser combinator [Hutton and Meijer 1996] operating on incoming messages under the hood, which allows the expected messages from the server to be specified in a flexible way that can handle non-deterministic ordering. It then sends a request to the server to retrieve the symbols in a document, waits for the response and finally makes some assertion about the response.

We also use lsp-test to drive benchmarks and memory leak detection (see §5.7).

4 Extensibility

The IDE described in §3 corresponds to the bare bones of a Haskell IDE, on which many features need adding. As examples, HLS integrates with 5 different formatters, a linting tool (HLint), a refactoring tool (Retrie), a runtime evaluator, documentation comment generator and many more besides. The key to supporting all these use cases, without adding additional complexity to the core, is an extensible core and a rich plugin mechanism.

4.1 LSP extensibility

The Language Server Protocol is extensible, in that it defines messages for various features that an IDE may or may not implement. Examples include:

- Context aware code completion. These suggestions have information about the current position of the cursor within the document, and so can provide suggestions that are semantically meaningful. For example, suggesting variable names that are available within the current scope or record fields matching the current type.
- Hover information. This is context-specific information provided as a separate floating window based on the cursor position. Additional analysis sources should be able to seamlessly add to the set of information provided.
- Diagnostics. The GHC compiler provides warnings and errors. It is possible to supplement these with any other information from a different analysis tool. We have already integrated HLint [Breitner et al. 2013] for lint-style suggestions and would like to also support tools such as Liquid Haskell [Vazou et al. 2014].

- Code Actions. These are context-specific actions that are provided based on the current cursor location. Typical uses are to provide actions to fix simple compiler errors reported, e.g. adding a missing language pragma or import. But they can also provide more advanced functionality, like suggesting refactorings of the code, including based on HLint suggestions.
- Code Lenses. These operate on the whole file, and offer a way to display annotations to a given piece of code, which can optionally be clicked on to trigger a code action to perform some function. In HLS these are used to display inferred type signatures for functions, and to add them to the code with one click.

The standardised messaging allows uniform processing on the client side for features, but also means new features are easy to add on the server side.

4.2 Core extensibility

Seen at the lowest level, our dependency library from §2 provides two things: a rule engine and an interface to LSP (§3.3). Therefore, to retain extensibility, there must be a way to add rules to the rule engine, and additional message handlers to the LSP message processing.

The Shake build system is designed to be extensible, so new types of rules can be defined, much the same way the core rules are defined in §2.4. These rules can depend on existing stages of the compiler with `use`, and then produce new values which are properly cached and invalidated. The biggest concern when adding such rules is that the results occupy memory on the build graph, which increases memory consumption by the IDE. By using an extensible build system we get some aspects of extensibility for free.

To extend the LSP message processing, we defined a type called `PartialHandlers`, which provides a function that can respond to certain messages. Using the `Monoid` type class such handlers can be combined, with the caveat that only one handler can process each message, so the last one wins. As a consequence, there can only be one handler responding to completions, whereas in reality there might be several complementary approaches to completions.

4.3 HLS plugins

While the core extensibility allows one handler for each type of message, the HLS plugins are intended to be more compositional. The aim is that most functionality (outside the core business of parsing and type checking source files) should live in plugins, and that users can freely compose these plugins. Therefore, HLS has a plugin descriptor which looks like:

```
data PluginDescriptor =
  PluginDescriptor
  { pluginId
      :: !PluginId
  , pluginRules
      :: !(Rules ())
  , pluginCommands
      :: ![PluginCommand]
  , pluginCodeActionProvider
      :: !(Maybe CodeActionProvider)
  , pluginCodeLensProvider
      :: !(Maybe CodeLensProvider)
  , pluginHoverProvider
      :: !(Maybe HoverProvider)
  ...
  }
```

The `pluginId` is used to make sure that if more than one plugin provides a *Code Action* with the same command name, HLS can choose the right one to process it. The field `[PluginCommand]` is a possibly empty list of commands that can be invoked in code actions. The rest of the fields can be filled in with just the capabilities the plugin provides.

As an example, a plugin providing additional hover information based on analysis of the existing GHC output would only fill in the `pluginId` and `pluginHoverProvider` fields, leaving the rest at their defaults. If two plugins providing hover information are both used, then the hover information from *both* will be combined in the response to LSP.

To evaluate the HLS `Plugin` values, the `pluginRules` are joined together and given as database rules. The provider fields are joined together into a single `PartialHandlers`, allowing their outputs to be combined as required.

5 Evaluation

We released our IDE and it has become an important part of the Haskell tools ecosystem. When it works, the IDE provides fast feedback with increasingly more features by the day. Building on top of a build system gave us a suitable foundation for expressing the right things easily. Building on top of Shake gave us a well tested and battle hardened library with lots of additional features we did not use, but were able to rapidly experiment with. However, the interesting part of the evaluation is what *does not* work.

5.1 Asynchronous exceptions are hard

Shake had been designed to deal with asynchronous exceptions, and had a full test suite to show it worked with them. However, in practice, we keep coming up with new problems that bite in corner cases. Programming defensively with asynchronous exceptions is made far harder by the fact that even `finally` constructions can actually be aborted, as there are two levels of exception interrupt. We suspect that in time we'll learn enough tricks to solve all the bugs, but it is a very error prone approach, and one where Haskell's historically strong static checks are non-existent.

5.2 Session setup

The majority of issues reported by users are come from the failure to setup a valid GHC session – this task is the first performed by HLS, and if this step fails, then every other feature will fail too. The diversity of project setups in the wild is astounding, which makes progress difficult. Haskell projects using Nix [Dolstra et al. 2004] are both common, and cause many problems.

In initial versions of HLS, we recommended that users write custom configuration files to accurately describe the setup of their project to hie-bios (see §3.2). However, as time has gone by, approaches for automatically detecting sensible configuration have improved and we now recommend most people use auto-detection where possible.

Work is currently underway to push logic that extracts setup information upstream from hie-bios into the build tools themselves, to expose more information and provide a more reliable interface for setting up sessions. As one example, a `show-build-info` command has recently been developed for Cabal that builds package dependencies and returns information about how Cabal would build the project in a machine readable format.

Finally, many projects require more than one GHC session to load all modules. While we have solutions §3.2.1, they are often a cause of problems, and we are working on more principled approaches §6.3.

5.3 Cancellation

While regularly cancelling builds does not seem to be a problem in practice, it would be better if the partial work started before a cancellation could be resumed. A solution like FRP [Elliott and Hudak 1997] might offer a better foundation, but we were unable to identify a suitable existing library for Haskell (most cannot deal with parallelism). We have performed initial experiments using the Haskell FRP library Reflex, which offered some benefits (lower overhead), but the lack of parallelism was problematic. Alternatively, a build system based on a model of continuous change rather than batched restarts might be another option. We expect the current solution using Shake to be sufficient for at least another year, but not another decade.

5.4 Runtime evaluation

Some features of Haskell involve compiling and running code at runtime. One such culprit is Template Haskell [Sheard and Peyton Jones 2002]. The mechanisms within GHC for runtime evaluation are improving with every release, but still cause many problems. Examples of problems we have observed are asynchronous exceptions giving problems (§5.3), segfaults and use of stale code.

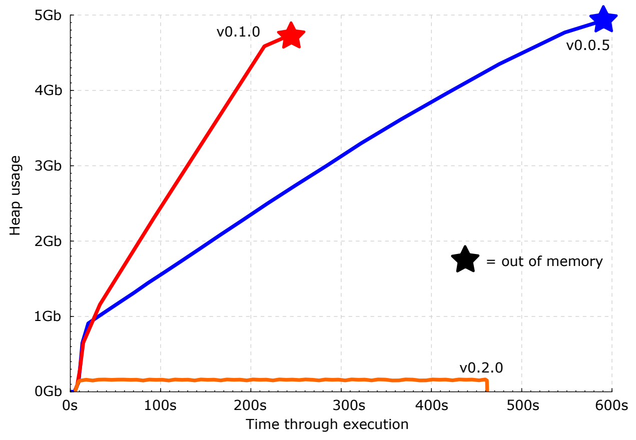

Figure 1. Heap usage for successive versions of HLS, replaying a consistent trace.

5.5 References

As stated in §2.1, an IDE offers three fundamental features – diagnostics, hover/goto-definition and find references. Our IDE offers the first two, but not the third. If the IDE was aware of the roots of the project (e.g. the `Main` module for a program) we could use the graph to build up a list of references. This work is ongoing, as we describe in §6.1, using the build system to produce information that can then be queried from a more traditional relational database.

5.6 Garbage collection

Currently, once a file has been opened, it remains in memory indefinitely. Frustratingly, if a temporary file with errors is opened, those errors will remain in the users diagnostics pane even if the file is shut. It is possible to clean up such references using a pass akin to garbage collection, removing modules not reachable from currently open files. We have implemented that feature for the DAML Language IDE [Digital Asset 2021], but not yet for HLS.

5.7 Memory leaks

A recurring complaint of our users is the amount of memory used. Indeed, one of the authors witnessed >70GB resident set sizes on multiple occasions on medium/large codebases. This memory consumption was not only ridiculously inefficient but also a source of severe responsiveness issues while waiting for the garbage collector[3] to waddle through the mud of an oversized heap.

Our initial efforts focused on architectural improvements like separate type-checking and a frugal discipline on what gets stored in the Shake graph. But it was not until a laziness related space leak was identified and fixed in the Haskell

[3]By default the GHC runtime will trigger a major collection after 0.3 seconds of idleness; thankfully this can be customized along with many other GC settings.

unordered-containers library[4] that we observed a material improvement. Figure 1 shows the heap usage in Gb (Y axis) of a replayed HLS session over time in seconds (X axis), for various versions of HLS. Versions 0.0.5 and 0.1.0 grow linearly without bound until running out of memory and failing. Notable is how the leak became more pronounced in 0.1.0 - this was only due to performance improvements that made HLS run faster, and thus leak memory faster too. Version 0.2.0 finally addressed the issue.

Given how much effort and luck it took to clear out the space leak, and the lack of methods or tooling for diagnosing leaks induced by laziness, we have installed mechanisms to prevent new leaks from going undetected:

1. A benchmark suite that replays various scenarios while collecting space and time statistics.
2. An experiment tool that runs benchmarks for a set of commits and compares the results, highlighting regressions.

Monitoring performance and preventing regressions is always a good practice, but absolutely essential when using a lazy language, due to the rather unpredictable dynamic semantics.

6 Ongoing and future work

Since the IDE was released, a number of volunteer contributors have been developing and extending the project in numerous directions. In addition, some teams in commercial companies have starting adopting the IDE for their projects.

6.1 Find References

Our approach to the problem of find references is to integrate with the hiedb library[5]. The hiedb library reads the `.hie` files produced by GHC, and extracts all sorts of useful information from them, such as references to names and types, the definition and declaration spans, documentation and types of top level symbols. All the information is stored in an SQLite database for fast and easy querying. Integrating this project with HLS provides find references, symbol search and opportunities to target refactorings.

Integrating hiedb with HLS is under active development, with many technical issues to be worked out, but the fundamental approach works well. The build system dependencies are used to generate the `.hie` files, and invalidate them when necessary, and that information is passed on to hiedb. For answering queries, a cache of the `.hie` file that the user is editing is used for queries about that file, while all other queries are performed against the database.

In addition to performing find reference and other queries, the hiedb database serves as an effective way to persist information across HLS runs. When a user first opens a project, while the build system is busy type checking a potentially large number of files, results from hiedb can be used to provide stale but immediate results to queries such as go to definition or hover.

6.2 Scalability

Our goal is to allow HLS to scale to codebases with more than 20,000 source files. Such a size, coupled with the relatively slow type checking speed of Haskell, requires that anything we do on all source files must be very cheap, while anything expensive must only be done on a subset of files. Similarly, any information stored per file must be compact – storing a type checked AST for each file would be prohibitively expensive.

Given a project with a set of N source files, where only a subset O are open in the editor, the following information is computed by the IDE:

1. $N * 3$ file exist checks, for every source file and the corresponding two interface files (.hi and .hie).
2. $N * 3$ timestamp checks, to compare every source file with the corresponding two interface files.
3. A global module graph with N nodes
4. T invocations to the type checker pipeline, which can be as few as O if all the interface files are up-to-date, or up to N otherwise.

Items 1 and 2 take linear time and cannot be avoided at startup, but after that they can be replaced by file system subscriptions. HLS already uses this approach for 1 but we have not yet adopted it for 2.

The global module graph is also a one-off cost in time, but a linear cost in space if kept in memory. Ideally, this graph should be stored in a disk-backed graph database with fast querying, although we do not foresee this being a bottleneck for codebases with less than 1,000,000 source files.

Item 4, running the parsing and typechecking pipeline, is unsurprisingly the biggest cost in terms of time. HLS can make use of interface files to skip this step, so if the file system has been suitably "primed" then only O compilation artifacts must be kept in memory. These interface files can be generated in advance e.g. by a cloud build system or as a by-product of a previous IDE run.

Unfortunately, when Template Haskell [Sheard and Peyton Jones 2002] is involved, it is necessary to keep many additional artifacts in memory – the number of artifacts becomes proportional to N in the worst case. There is a fair amount of engineering work required, both in GHC and in our IDE, to reduce the memory usage when Template Haskell is involved.

Over time the ability of HLS to scale has improved markedly – initial versions got sluggish at 200 source files, but now 10,000 can be made to work (with appropriate setup). Such progress requires eliminating all bottlenecks, of which the above are only the most serious.

[4] https://github.com/haskell-unordered-containers/unordered-containers/issues/254
[5] https://github.com/wz1000/HieDb

6.3 Multiple Home Unit in GHC

As described in §3.2.1, GHC has the concept of a home component, and assumes that all other components are on disk and do not change. If you wish to open both a library and an executable that depends on it simultaneously, that causes problems when the library changes but GHC does not notice. While we have implemented workarounds, the correct solution is to make GHC aware of *multiple* home units simultaneously.

The work to expand GHC to multiple home units is in progress, and involves expanding GHC's `DynFlags` data type to go from having the notion of one home unit, to many, with one of those many being the "current" home unit. Such a patch has already been written, as described in several blog posts[6]. There are three remaining limitations:

1. The code has not yet been merged into GHC. Until that happens, it cannot be used by HLS and by our users.
2. Modules in components can be hidden from each other. However, which modules are visible is not actually something GHC knows about, but is controlled by the Cabal project description. Since GHC does not know about Cabal projects, and only Cabal knows about visibility, the information is not yet visible when GHC is processing multiple components. As a consequence, an executable can use a hidden module in a library, and that error will only be detected when the project is next compiled, not in the IDE.
3. GHC has a feature allowing package imports, where both the module name and package of an import are specified. Normally the name of packages are read from the package database, but when GHC compiles multiple components in memory, such information is not available.

The arrival of multiple home units will simplify HLS, fix some bugs, reduce memory consumption, remove unnecessary invalidations and improve parallelism. We consider such a project one of the most meaningful improvements to HLS to be found on the GHC side.

6.4 Parsing

We currently reuse the standard GHC parser, which operates on a whole file, and returns either an error or a parse tree. Neither of those attributes are desirable.

Parsing the file at once. GHC parses the whole file at once. However, given the model of LSP, we have a diff to the file – we can identify precisely which part changed. But GHC cannot make use of that information. In practice, given that parsing is fast and type-checking is comparatively slow, it is not clear that a diff-based parsing algorithm would provide many benefits, unless combined with diff-based type-checking [Busi et al. 2019].

Single error. If GHC detects a parse-error in a file, it reports the entire file as being invalid. In practice, while a user is editing an expression, often the rest of the program will be well-formed, but we cannot use GHC to extract any information beyond the parse error. Worse, the Haskell grammar was not designed with incremental development in mind. Consider the monadic expression:

```
main = do
  x <- getLine
```

This program results in a parse error, meaning we cannot obtain information about the name resolution or definition of `getLine`. GHC has gained the ability to defer type errors [Vytiniotis et al. 2012], and some users have proposed deferring some class of parse errors like the one above[7]. Going further, the popular Tree Sitter library can define a parser that always produces a parse tree, potentially with a number of errors embedded within it, using parsing with error recovery [Wagner and Graham 1997]. Such an approach would provide more accurate information during edits.

6.5 Debugging

Many IDE's handle both text editing facilities and debugging. However, the LSP specification [Microsoft 2021b] sticks strictly to the text editing, with the companion Debug Adaptor Protocol (DAP) [Microsoft 2021a] providing cross-editor hooks for debugging. We consider extending HLS to deal with debugging to be important, but a *lot* of difficult engineering work, for a few reasons:

1. In comparison to LSP, DAP is a lot newer technology, so is not as well supported or stabilised between editors.
2. Haskell has a fairly unique lazy evaluation model, which causes some difficulties when integrating in to traditional debugging frameworks. Concepts like stepping to the next line are possible to map to Haskell, but not trivial.
3. Our experience with the runtime evaluation mechanisms of Haskell has found that they are significantly less robust, see §5.4.

However, none of these are likely to be fatal impediments. There is an evaluation plugin for HLS, which allows evaluating snippets written in documentation comments, which requires some of the same underlying pieces. There is also an interactive debugger provided by the interactive GHC environment [Marlow et al. 2007], which we should be able

[6]https://mpickering.github.io/ide/posts/2020-10-12-multiple-home-units.html

[7]https://github.com/ghc-proposals/ghc-proposals/pull/333

to take advantage of. Importantly, the underlying architecture of HLS seems to fit with a debugger, all that is left is a substantial amount of engineering.

7 Related work

The idea of building compilers and IDE's on top of build systems is starting to gain traction, but it is still far from the standard approach. One term which is sometimes applied to this idea is *query-based*, referencing the notion that nodes on the dependency graph are "queries" of their dependencies.

The IDE which is most similar to our design is the Rust Analyser IDE [Rust IDE Contributors 2020]. Rust Analyser uses a custom library called Salsa [Salsa Contributors 2021], which is described as an incremental computation library. Salsa is defined in terms of inputs, and functions that produce outputs – much like Shake. Looking at Salsa through the lens of a build system [Mokhov et al. 2018], it is monadic with early cut-off. The experience of Rust Analyser is that the benefits are "generality and correctness" and that "you are immune to cache invalidation bugs", while the downside is "extra complexity, slower performance". We agree with their assessment.

While building a dependency graph IDE on top of a standard compiler is a viable approach, pushing the dependency graph deeper inside the compiler can give further benefits. As examples, both C# [Hejlsberg 2016] and Rock [Fredriksson 2020] feature dependency graphs at a finer level, where individual functions in the compiler are separate nodes. As a consequence, when editing a large file, only a small portion of the file needs to be invalidated. Equally important, if one part of the file contains an error, the remainder of the file, and even those files depending on that file, can continue to work (much easier and more robust than our approach from §3.2.2). We expect that as an IDE becomes a standard part of language tooling, the compiler will be increasingly designed as an IDE first.

While some compilers have been designed around build systems at their lowest level, other compilers have observed that even without the IDE focus, there are problems within compilers that are naturally solved by off-the-shelf build systems. Examples include in Stratego [Smits et al. 2020] and experiments with replacing GHC `--make` with Shake [Yang 2016].

Finally, there are relatively few reports on the technical challenges associated with writing an IDE. The Merlin IDE [Bour et al. 2018] for OCaml is one of the exceptions. Some of the difficulties they encountered are remarkably similar to ours – OCaml has a variety of different project formats (just like Haskell, see §3.2) and scalability is a constant concern (see §6.2). Much of their paper is dedicated to improving parsing results in the presence of errors – something we have not yet explored, and would definitely benefit from.

8 Conclusion

We implemented an IDE for Haskell on top of the build system Shake. The result is an effective IDE, with a clean architectural design, which has been easy to extend and adapt. We consider both the project and the design a success. Our design separates an IDE into an incrementality engine (in our case based on a build system), rules which describe dependencies and an integration layer with LSP – we believe this design is a good fit for many programming language IDE/LSP integrations. Build systems offer a powerful abstraction whose use in the compiler/IDE space is likely to become increasingly prevalent.

Acknowledgments

Thanks to everyone who contributed to the IDE. The list is long, but includes the Digital Asset team (who did the initial development), the Haskell IDE engine team (who improved the GHC API and lead the trail), and the hie-bios team (who made it feasible to target real Haskell projects). In addition, many open source contributors have stepped up with bug reports and significant improvements. Truly a team effort.

References

Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–15.

Joachim Breitner, Brian Huffman, Neil Mitchell, and Christian Sternagel. 2013. Certified HLints with Isabelle/HOLCF-Prelude. In *Haskell And Rewriting Techniques (HART)*.

Matteo Busi, Pierpaolo Degano, and Letterio Galletta. 2019. Using standard typing algorithms incrementally. In *NASA Formal Methods Symposium*. Springer, 106–122.

Manuel MT Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *POPL*. 1–13.

Digital Asset. 2021. DAML Programming Language. (2021). https://www.daml.com/.

Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *LISA*, Vol. 4. 79–92.

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *ICFP*.

Facebook. 2021. Buck. (2021). https://buck.build/.

Olle Fredriksson. 2020. Query-based compiler architectures. (25 June 2020). https://ollef.github.io/blog/posts/query-based-compilers.html.

Google. 2021. Bazel. (2021). https://bazel.build/.

Anders Hejlsberg. 2016. Modern Compiler Construction. (12 May 2016). https://channel9.msdn.com/Blogs/Seth-Juarez/Anders-Hejlsberg-on-Modern-Compiler-Construction.

Graham Hutton and Erik Meijer. 1996. Monadic Parser Combinators.

Isaac Jones. 2005. The Haskell Cabal: A Common Architecture for Building Applications and Libraries. In *Trends in Functional Programming*. 340–354.

Simon Marlow, José Iborra, Bernard Pope, and Andy Gill. 2007. A Lightweight Interactive Debugger for Haskell. In *Haskell Workshop*. 13–24.

Microsoft. 2021a. Debug Adaptor Protocol. (2021). https://microsoft.github.io/debug-adapter-protocol/.

Microsoft. 2021b. Language Server Protocol. (2021). https://microsoft.github.io/language-server-protocol/.

Neil Mitchell. 2012. Shake before building: Replacing Make with Haskell. In *ICFP*. ACM.

Neil Mitchell, Moritz Kiefer, Pepe Iborra, Luke Lau, Zubin Duggal, Hannes Siebenhandl, Matthew Pickering, and Alan Zimmerman. 2020. Building an Integrated Development Environment (IDE) on top of a Build System. In *Draft Proceedings of the 32nd International Symposium on Implementation and Application of Functional Languages (IFL 2020)*. 222–230.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proceedings ACM Programing Languages* 2, Article 79, 79:1–79:29 pages.

Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-recursive Make Considered Harmful - Build Systems at Scale. In *Haskell 2016*. 55–66.

Simon Peyton Jones. 2001. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*. IOS Press, 47–96.

Rust IDE Contributors. 2020. Three Architectures for a Responsive IDE. (20 July 2020). https://rust-analyzer.github.io/blog/2020/07/20/three-architectures-for-responsive-ide.html.

Salsa Contributors. 2021. About Salsa. (2021). https://salsa-rs.github.io/salsa/.

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Haskell Workshop*. 1–16.

Jeff Smits, Gabriël D. P. Konat, and Eelco Visser. 2020. Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System. *CoRR* (2020).

The GHC Team. 2021. The GHC Compiler, Version 8.10.3. (2021). https://www.haskell.org/ghc/.

The Haskell IDE Engine Team. 2020. haskell-ide-engine. (2020). https://github.com/haskell/haskell-ide-engine.

The HIE BIOS Team. 2021. hie-bios. (2021). https://github.com/mpickering/hie-bios.

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Refinement types for Haskell. In *ICFP*. 269–282.

Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. 2012. Equality Proofs and Deferred Type Errors: A Compiler Pearl. In *ICFP*. 341–352.

T. Wagner and S. Graham. 1997. Incremental analysis of real programming languages. In *PLDI '97*.

Edward Yang. 2016. ghc –make reimplemented with Shake. (2016). https://github.com/ezyang/ghc-shake.