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

Matthew Pickering
University of Bristol
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.

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 ([1] is one of the exceptions). We embarked on a project to write a Haskell IDE (originally for the GHC-based DAML language [4]), 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 turned out to be the turning point, and 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 *ghcide*[1], which is then integrated into the *Haskell Language Server*[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 around it using plugins §4. We look at where the build system both helps and hurts §5. We then look at the ongoing and future work §6 before concluding §7.

[1] https://github.com/digital-asset/Ghcide
[2] https://github.com/haskell/haskell-language-server

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

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 priority:

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.

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

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.

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 [17]. 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 [12, §3.5]. Given those constraints, and the presence of an author in common, we chose to use Shake [11].

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 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.

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 problems in practice, and is a significant divergence from the way Shake normally works.

Firstly, we interrupt using asynchronous exceptions [14]. Lots of Haskell code isn't 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.

Secondly, when interrupting a build, some things might be in progress. If type checking a big module takes 10 seconds, and the user presses the key every 1 second, it will keep aborting 1 second through and never complete. In practice, interrupting hasn't been a significant hurdle, although we discuss possible remedies in §5.3.

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, we want 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 [11, §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 nearly nothing for a traditional build, for an IDE that is excessive. We solved this problem by adding a `Database` module to Shake that retains the key/value map in memory.
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 [2] 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 don't 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 "pure" API (where the output is based on the inputs), with the signature:

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

2.5 Error tolerance

An IDE needs to be 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. We employ a variety of mechanisms to achieve this goal:

- GHC's `-fdefer-type-errors` and `-fdefer-out-of-scope-variables` flags turn type errors and out of scope variable errors into warnings, and let it proceed to typecheck and return usable artifacts to the IDE. This 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 define the `useWithStale` function to get the most recent, *successfully* computed value of a key, even if it was for a older version of the source.
- The function `useWithStale` has return type `Maybe (v, PositionMapping)` where `v` is the return type of the rule, and the type `PositionMapping` is a set of functions that help us convert source locations in the current version of a document back into the version of the document for which the rule was last computed successfully, and vice versa. For example, if the user inserts a line at the beginning of the file, the reported source locations of all the definitions in the file need to be moved one line down. Similarly, when we are querying the earlier version of the document for the symbol under a cursor, we must remember to shift the position of the cursor up by one line. We maintain this mapping between source locations for all versions of a file for which we have artifacts older than the current version of the document.

2.6 Responsiveness

An IDE needs to return results quickly in order to be helpful. However, we found that running all the Shake rules to check for freshness and recompute results on every single request was not satisfactory with regards to IDE responsiveness. This problem was particularly evident for completions, which need to show up quickly in order to be useful. However, each

keystroke made by a user invalidates the Shake store, which needs to be recomputed.

For this reason, we added an alternative mechanism to directly query the computed store of results without rerunning all the Shake rules. We defined a function `useWithStaleFast` for this purpose, with a signature like `useWithStale`. This function first asynchronously fires a request to refresh the Shake store. Immediately afterwards, it checks to see if the result has already been computed in the store. If it has, it immediately returns this result, along with the `PositionMapping` for the version of the document this result was computed for, as described in the previous section. If the result has never been computed before, it waits for recomputation request to Shake to finish, and then returns its result.

This technique provides a significant improvement in the responsiveness of requests like hovering, go to definition, and completions, in return for a small sacrifice in correctness.

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 [18] project 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 preprocessing 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's 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 hasn't changed since `D.hi` was produced, but happily GHC performs recompilation checks and complains when this assumption isn't 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. This 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 they also include information about package dependencies, which need to be built beforehand and registered with *ghc-pkg*. Furthermore, these details are all entirely dependent on the build tool: The flags that *Stack* passes to GHC to build a project will be different from what *Cabal* passes, because each builds and stores package dependencies in different locations and package databases.

Because this whole process 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 *hie-bios* [19] 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 [13], Bazel [7] or any other build system that invokes GHC.

hie-bios 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 functions 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.

3.3 Handling multiple components in one session

Haskell projects are often separated into multiple packages, and when using Cabal [9], 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. Consequentially, using the IDE on a big project with multiple subprojects should work as expected.

However, the GHC API is designed to only handle a single component at a time. This limitation is 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, assumes that any dependencies are stored on disk and won't 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.

3.4 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) [10] for this, 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, *ghcide* deals 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 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.

3.5 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 *lsp-test*, a testing framework for LSP servers. *lsp-test* 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 "ghcide" 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 *ghcide* to open up a document, and then ignore any notifications it may send with `skipMany anyNotification`. A session is actually a parser combinator [8] 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.

An additional benefit of having testing at the transport level is that we can reuse much of the test suite in IDEs building on top of Ghcide for free, since we only need to swap out what server the tests should be run on. *lsp-test* is also used not only for testing, but also for automating benchmarks (See §5.7).

4 Plugins

The IDE described in §3 corresponds to the Haskell library Ghcide, which is currently used in at least four different roles:

- With a thin wrapper as a stand alone IDE for GHC.
- As the engine powering the IDE for DAML.
- As the foundation of a compiler for DAML.
- As the GHC layer for a more full-featured Haskell IDE (Haskell Language Server, HLS).

The key to supporting all these use cases is a rich plugin mechanism.

4.1 LSP extensibility

The Language Server Protocol is extensible, in that it provides sets of messages that provide a (sub) protocol for delivering IDE features. Examples include:

- Context aware code completion
- 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 should be possible to supplement these with any other information from a different analysis tool. Such as `hlint`, or `liquid haskell`.
- 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.
- 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 ghcide these are used to display inferred type signatures for functions, and allow you 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 should be easy to add on the server side.

4.2 Ghcide plugins

Internally, *ghcide* is two things, a rule engine, and an interface to the Language Server Protocol (§3.4).

So to be extensible, there must be a way to add rules to the rule database, and additional message handlers to the LSP message processing.

A plugin in *ghcide* is thus defined as a data structure having `Rules` and `PartialHandlers`

A *Monoid* class is provided for these, meaning they can be freely combined. There is one caveat, in that order matters for the `PartialHandlers`, so the last message handler for a particular message wins.

In practical terms the plugin uses these features as follows

- It provides rules to generate additional artefacts and add them to the Shake graph if needed. For most plugins this is unnecessary, as the full output of the underlying compiler is available. Typical use-cases for this would be to trigger additional processing for diagnostics, such as for `hlint` or similar external analysis. Note that care must be taken with adding rules, as it affects both memory usage and processing time.
- It provides handlers for the specific LSP messages needed to provide its feature(s).

This is a fairly low-level capability, but it is sufficient to provide the plugins built in to *ghcide*, and serve as a building block for the *Haskell Language Server*.

4.3 Haskell Language Server plugins

The *Haskell Language Server* makes use of *ghcide* as its IDE engine, relying on it to provide fast, accurate, up to date information on the project being developed by a user.

Where *ghcide* is intended to do one thing well, *Haskell Language Server* is targeted at being the "batteries included" starter IDE for any Haskell user. *HLS* is the family car where *ghcide* is the sports model.

We will describe here its approach to plugins.

Firstly, a design goal for *HLS* is to be able mix and match any set of plugins. The current version (0.3) has a set built in, but the road map calls for the ability to provide a tiny custom `Main` module that imports a set of plugins, puts them in a structure and passes them in to the existing main programme.

To enable this, it 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 `[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.

So 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.

4.4 Haskell Language Server plugin processing

The *HLS* engine converts the *HLS*-specific plugin structures to a single *ghcide* plugin.

It simply combines the `Rules` monoidally, but does some specific processing for the other message handlers.

The key difference is that *HLS* processes the entire set of `PluginHandlers` at once, rather than using the pairwise `mappend` operation.

This means that when a hover request comes in, it can call *all* the hover providers from all the configured plugins, combine the results and send a single combined reply to the original request.

The same technique is used as appropriate for each of the message handlers.

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 didn't use, but were able to rapidly experiment with. However, the interesting part of the evaluation is what *doesn't* 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's 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 is the first port of call for ghcide, so if this step fails then every other feature will fail. The diversity of project setups in the wild is astounding, and without explicit configuration hie-bios struggles to detect the correct cradles for the correct files (see §3.2). It is a difficult problem, and the plethora of Haskell build tools out there only exacerbates it further. Tools such as Nix [5] are especially common and problematic.

Work is currently underway to push the effort upstream from hie-bios into the build tools themselves, to expose more information and provide a more reliable interface for setting up sessions: Recently a `show-build-info` command has been developed for *cabal-install* that builds package dependencies and returns information about how Cabal would build the project in a machine readable format.

In addition, some projects require more than one GHC session to load all modules — we are still experimenting with solutions for this problem.

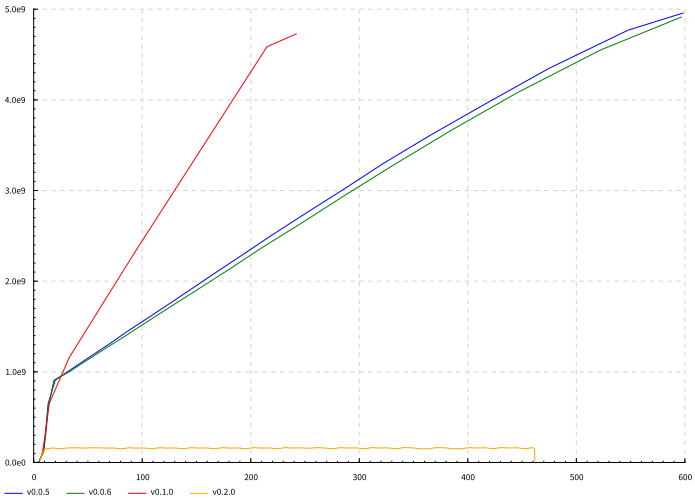
5.3 Cancellation

While regularly cancelling builds doesn't 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 [6] might offer a better foundation, but we were unable to identify a suitable existing library for Haskell (most cannot deal with parallelism). 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 [15]. The mechanisms within GHC for runtime evaluation are improving with every release, but still cause many problems.

Figure 1. Heap usage over successive versions of Ghcide



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. However, we have not yet done so.

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 [4], but not yet for the Haskell IDE.

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 [3] for the garbage collector 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 wasn't until a laziness related space leak was identified and fixed in the Haskell library `unordered-containers` library that we observed a material improvement. Figure 1 shows the heap usage of a replayed Ghcide session over time, for various versions of Ghcide, where we can see that for versions prior to 0.2.0 it would grow linearly and without bounds until running out of memory.

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

6 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. Some of the items listed in this section are currently under active development, while other are more aspirational in nature.

6.1 hiedb

`hiedb`[4] is a tool to index and query GHC extended interface (`.hie`) files. It reads `.hie` files 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, storing it in a SQLite database for fast and easy querying.

Integrating hiedb with Ghcide has many obvious benefits. For example, we can finally add support for "find references", as well as allowing you to search across all the symbols defined in your project.

In addition, the hiedb database serves as an effective way to persist information across Ghcide runs, allowing greater responsiveness, ease of use and flexibility to queries. hiedb works well for saving information that is not local to a particular file, like definitions, documentation, types of exported symbols and so on.

A branch of Ghcide integrating it with hiedb is under active development.

Ghcide acts as an indexing service for hiedb, generating .hi and .hie files which are indexed and saved in the database, available for all future queries, even across restarts. A local cache of .hie files/type-checked modules is maintained on

[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.

[4] https://github.com/wz1000/HieDb

top of this to answer queries for the files the user is currently editing, while non-local information about other files in the project is accessed through the database.

6.2 Replacing Shake

As we observed in §5, a build system is a good fit for an IDE, but not a perfect fit. Using the abstractions we built for our IDE, we have experimented with replacing Shake for a library based on Functional Reactive Programming [6], specifically the Haskell library `Reflex`. Initial results are promising in some dimensions (seems to be lower overhead), but lacking (no parallelism). We continue to experiment in this space.

6.3 Multiple Home Unit in GHC

As described in §3.3, there are limitations in the GHC API that force us to handle the module graph in-memory. This is error-prone and complicates the IDE quite a lot. Moving this code into GHC improves the performance and simplify support for multiple GHC versions. Moreover, it might prove useful for follow up contributions to enable GHC to work as a build server. As such, it can compile multiple units in parallel without being restarted, while using less memory in the process.

7 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.

The idea of using a build system to drive a compiler is becoming more widespread, e.g. in Stratego [16] and experiments with replacing GHC `--make` [20]. By going one step further, we can build the entire IDE on top of a build system. The closest other IDE following a similar pattern is the Rust Analyser IDE [3], which uses a custom recomputation library, not dissimilar to a build system. 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

[1] 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.

[2] Manuel MT Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–13.

[3] 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.

[4] Digital Asset. 2020. DAML Programming Language. (2020). https://www.daml.com/.

[5] 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.

[6] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming*.

[7] Google. 2020. Bazel. (2020). http://bazel.io/.

[8] Graham Hutton and Erik Meijer. 1996. Monadic Parser Combinators.

[9] Isaac Jones. 2005. The Haskell Cabal: A Common Architecture for Building Applications and Libraries, Marko van Eekelen (Ed.). 340–354.

[10] Microsoft. 2020. Language Server Protocol. (2020). https://microsoft.github.io/language-server-protocol/.

[11] Neil Mitchell. 2012. Shake before building: Replacing Make with Haskell. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 55–66.

[12] 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.

[13] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-recursive Make Considered Harmful - Build Systems at Scale. In *Haskell 2016: Proceedings of the ACM SIGPLAN symposium on Haskell*. 55–66.

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

[15] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 Haskell Workshop, Pittsburgh*. 1–16.

[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). arXiv:2002.06183

[17] The GHC Team. 2020. The GHC Compiler, Version 8.8.3. (2020). https://www.haskell.org/ghc/.

[18] The haskell-ide-engine Team. 2020. haskell-ide-engine. (2020). https://github.com/haskell/haskell-ide-engine.

[19] The hie-bios Team. 2020. hie-bios. (2020). https://github.com/mpickering/hie-bios.

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