A stronger foundation for interactive Haskell tooling

Daniel Gröber

Motivation
Inevitably new and prospective Haskell users will attempt to obtain editor/IDE integration for Haskell but unfortunately they are much more likely to fail than to succeed due to the less than ideal state of the tools.

Haskell downstream tooling is simply in a bit of a bad place at the moment. Things have always moved fast in GHC land but recently things got a lot worse for tools when the release frequency increased from every-two-years to every-six-months.

As if that weren’t enough, Cabal – the main Haskell build tool – is almost ready to switch over to the long awaited Nix-style new-build commands. This switch brings with it a major change to how tooling has to interact with the build system.

Bad tooling is a major source of frustration for new and experienced Haskell users alike. This proposal will substantially improve the reliability, performance and maintainability of tooling efforts.

About me
I’ve been working in the Haskell tooling space for around 6 years now, mainly as a developer and later the maintainer of ghc-mod and cabal-helper. I have contributed to GHC, Cabal and Stack in the past all in the interest of better tooling.

Overview
This proposal consists of three main areas:

1. Improvements in GHC to reduce friction for downstream tooling efforts
2. Work on cabal-helper to enable easy new-build support
3. Integration of the above into Haskell IDE Engine

Tasks
1. Improve GHC API for tooling use-cases

There are a number of minor problems in the interface between GHC and tooling. In this task I plan on making it possible to remove a lot of hacks and workarounds from downstream projects to make them more reliable and performant.

1.1. Support loading in-memory files with GHC API

The Language Server Protocol as implemented by Haskell IDE Engine allows users to perform queries on unsaved files. To implement this with the GHC API HIE currently uses ghc-mod’s file-mapping functionality which is in turn emulated by writing unsaved changes to temporary files. This has proven to be a major source of bugs and really needs to be replaced.
In principle GHC already exposes this functionality in its API through the `targetContents` field (Hackage docs). However using it is currently not supported when when any preprocessors like CPP are involved in the module graph.

I am going to remove the preprocessor limitation and integrate the now working `targetContents` feature into Haskell IDE Engine.

ETA: 1 week

1.2. Make a partial module graph accessible even when some modules fail to parse

Currently GHC’s `depanal` function (Hackage docs) only returns a module graph when all modules headers have been successfully parsed, however experience with ghc-mod based tooling shows that there are cases when even just a partial module graph is useful.

The idea here is that when a user is in the middle of performing a complicated change on multiple Haskell modules it is very bad UX for tooling to stop working on an unrelated module when *any* module in the module graph currently has a module header that doesn’t parse.

This represents just a small step towards making GHC-based tooling more robust in the face of user introducing errors into the source.

ETA: 1 week

2. Add support for multi-session GHC API usage

At the moment the GHC API really only reliably supports having one session, and correspondingly one Cabal component, loaded in a process at a time. A component is for example a library, executable or test suite. However Haskell projects often consist of many such components that are all developed in concert. With Cabal new-build or Stack we can even have multiple packages, themselves with many components, involved in a development session.

Tooling such as ghc-mod and Haskell IDE Engine currently handles this by only loading one component at a time and discarding the GHC session – including all the precious compilation products – when a different component needs to be made active, which is obviously a huge drain on performance when switching between components frequently.

Even worse, some use-cases really do need access to multiple components at a time to have any reasonable performance at all. For example global refactoring with HaRe would immensely benefit from a multi-session GHC API.

In the past the idea of having multiple GHC sessions in a single process was infeasible because widely used language features such as Template Haskell require dynamic linking in the compiler process, but the state of the dynamic linker was global in a process.

However this particular problem has recently been resolved through unrelated work on allowing GHC to work in a cross-compilation context, namely `-fexternal-interpreter`. There is also work on simply allowing multiple linker instances in one process (PR #388).

I believe it is high time to revisit this issue as it would vastly improve the performance and user experience of Haskell tooling.

2.1. Test and document multi-session GHC API usage

I have performed some preliminary testing and there don’t seem to be any glaring problems with doing this in the API, so this task is all about developing tests for all the functionality required for tooling to function well, fixing what (hopefully) little is broken and documenting the fact that it works reliably.
2.2. Improve multi-session memory usage

Once we can reliably use multiple GHC sessions in a single process it becomes possible to reduce the overall memory consumption by sharing resources across sessions. GHC caches a lot of essentially read-only data per-session which should be easy to share.

I plan to measure which resources are consuming the most memory in the context of Haskell IDE Engine. My hunch is that GHC’s \texttt{ModIface} cache, corresponding to \texttt{.hi} files on disk, would be a good target as components from the same package are very likely to share a significant number of package dependencies. Hence sharing module interfaces is almost guaranteed to give us a nice reduction in memory usage.

ETA: 2 weeks

3. Finish and release cabal-helper

For the past eight months I’ve been working on a re-design of my \texttt{cabal-helper} library, with the intention of using it as the build tool abstraction layer for Haskell development tools with the changes brought by Cabal’s new-build taken into account.

This library essentially provides the missing piece of infrastructure to allow tools to easily interact with any \texttt{lib:Cabal}-based build system (currently Cabal and Stack). This work is nearly complete and only needs a final concentrated push over the finish line.

There is currently ongoing work in Cabal to add the \texttt{show-build-info} command, the integration of which into cabal-helper would allow improving performance while allowing it to work well in more complex environments.

I plan on producing a Hackage release of cabal-helper-1.0 incorporating functionality required to support new-build. If there is time I will work on making sure \texttt{show-build-info} can be used in cabal-helper as well as integrating it.

ETA: 2 weeks

4. Integrate previous work into Haskell IDE Engine

Haskell IDE Engine has emerged as a community hot spot for interactive tooling. It has demonstrated an ability to attract and retain contributors and I think it’s the right place to bring a lot of the functionality we all want in our development environments together.

The economics of the Language Server Protocol protocol mean that supporting a wide array of clients is easy, because most of the hard lifting is done by a client specific implementation which can be shared across all language servers.

Current releases of HIE still depend on ghc-mod for its workarounds and hacks on top of the GHC API as well as its build tool abstraction functionality. With my work in task 1 the former should become redundant. Furthermore task 3 goes a long way towards making interacting with different build tools easy, though not quite as easy as with ghc-mod yet.

Recently my mentor Matthew Pickering has started working on simplifying the interface between HIE and GHC (PR #1126), particularly by taking ghc-mod out of the equation. His branch implements a simpler but less powerful way of interacting with the underlying build system together with core improvements in the way of making HIE less entangled with ghc-mod.
I plan on building on Matthew’s work to integrate my work from the other three tasks, as well as implementing the pieces missing to bring us to feature parity with ghc-mod’s build system abstraction, namely automatic Cabal component selection.

ETA: 4 weeks

Contact

- E-mail: Daniel Gröber <gsoc@dxld.at>
- IRC: dxld @ Freenode: #ghc-mod, #haskell-ide-engine
- GitHub: https://github.com/DanielG