Building an IDE on top of a Build System

The tale of a Haskell IDE
How to write a compiler?

+ 1000’s of papers, on every single aspect
+ A course at most universities
+ Blog posts galore
How to write an IDE?

Kinetics and quantum yield of photoconversion of protochlorophyll (ide) to chlorophyll (ide)

OF Nielsen, A Kahn - Biochimica et Biophysica Acta (BBA)-Bioenergetics, 1973 - Elsevier

... process between protochlorophyll(ide)* and the reductant as proposed earlier 8. Next, we must consider the possibilities for the deexcitation of protochlorophyll(ide)* which do not lead to its reduction but return protochlorophyll(ide)* to the ground-state. Accordingly we write ...

🌟 99 Cited by 41 Related articles All 4 versions
Base it on a build system!
The tale of a Haskell IDE

• First implemented by Digital Asset for DAML language (Haskell on a distributed ledger)
• Split out as ghcide, for Haskell
• Integrated into haskell-language-server

Now: A workable Haskell IDE

https://github.com/haskell/haskell-language-server
https://www.youtube.com/watch?v=WBYWtrKjKcE
Why does a build system feel right?

• Lots of *dependencies*
  – Contents > Parse > TypeCheck
  – TypeCheck also depends on the transitive import type checks

• Lots of *invalidation*
  – If source changes, invalidate Parsing + TypeCheck

Build *primitives*, then *wire* them together!
TypeCheck primitive

typecheckModule :: HscEnv
-> [TcModuleResult]
-> ParsedModule
-> IO
  ([Diagnostic]
   , Maybe TcModuleModuleResult)
TypeCheck wiring

type instance RuleResult TypeCheck = TcModuleResult

define $ \text{TypeCheck file } -> \text{ do}

\text{pm } <- \text{ use_ GetParsedModule file}

\text{deps } <- \text{ use_ GetDependencies file}

\text{tms } <- \text{ uses_ TypeCheck (transitiveModuleDeps deps)}

\text{packageState } <- \text{ useNoFile_ GhcSession}

\text{liftIO } $ \text{ typecheckModule packageState tms pm}
Architecture of an IDE

- GHC API
- Primitives (used by)
- Wiring (used by)
- IDE Library (based on)
- Build System (wrapped)
- Editor (triggered)

The diagram illustrates the components and their interactions within an IDE architecture.
Build an IDE library,
that does whatever an IDE requires,
on top of a build system
What does an IDE do?

Lots, but three “core” features.

• Errors/warnings – show the current state of the code as you type.
• Hover/goto-definition – give information about the code in front of you.
• Find references – tell you where an identifier is used.
What does a build system do?

- Maps keys to values through computations
- Computations depend on other keys

- We use Shake, because:
  - Has monadic dependencies (an IDE is not static)
  - Written in Haskell, easy integration with GHC API
  - Allows fully custom rules
IDE Library

• A wrapper over Shake
• Set up dependencies
  – FilePath > Contents > Parse > Imports > TypeCheck
• Every time anything changes (e.g. keystroke)
  – Abort whatever is ongoing
  – Restart from scratch, skipping things that haven’t changed
• Report errors as you get them
IDE Library features

Easy

- Parallelism
- Incrementality
- Dependencies
- Monadic
- Well-engineered

Less-easy

- Error reporting
- Restarting
- Performance
Error Reporting

• Keys are (Phase, FilePath)
  – (Parse, Foo.hs), (TypeCheck, Foo.hs)
• Values contain errors as first-class info
  – ([Diagnostic], Maybe r)
  – (xs, Nothing), I raised an error
  – (xs, Just v), I raised some warnings
  – ([], Nothing), my dependency failed
• Collect warnings for all phases for a file
IDE Library primitives

define $ \backslash \text{Phase file} -> \text{do}
   \text{use Phase file -- return the real value}
   \text{use_ Phase file -- fail if Nothing}
   \text{uses_ Phase files -- parallel use_}
Restarting

• On change:
  – Abort, with asynchronous exception
  – Restart

• Rules are cached. In-progress actions are lost.

• Don’t underestimate the engineering effort in async exceptions

• Would a GHC suspend primitive work?
Performance

• Build systems are about *files*
  – We contributed an in-memory API for Shake
• IDEs might restart 200 times per minute
  – Scanning a large graph can get expensive
  – Some optimisation work, some GHC bugs
  – Ongoing effort
• Would an FRP-like solution work better?
Connecting to the IDE

• Key/Value mappings which depend on each other
  – Wiring GHC functions and types into a graph

• Request comes in from IDE
  – Modify the input values
  – Compute some values from keys
  – Format that information appropriately

• Lots of plumbing
Shake was a good idea

• IDE is a very natural dependency problem
• Robust parallelism
• Thoroughly debugged for exception handling
  – GHC API has a few issues in corner cases here
• Has good profiling (caught a few issues)
• Has lots of features – we could replicate the end state, but not the path there
Full IDE

GHC

hie-bios

ghcide

Haskell-language-server

https://github.com/haskell/haskell-language-server

haskell-lsp
It works!

• 524 stars, 85 forks, 399 pull requests, 62 contributors, 4K VS Code installs (at least)
• Can edit the GHC codebase (~500 modules)
• Used by several companies
• Still the basis of the DAML IDE
How to write an IDE?

lots more details, including:

• What garbage collection means
• How to put plugins over the top
• How we test it
• Memory leaks we’ve had
• .hi files
Authors

Additional Credits

Digital Asset, ZuriHac, MuniHac, many others...
What does LSP do?

• Language Server Protocol (LSP)
• Communication protocol for VS Code, Vim, Emacs etc.
• Tell the editor when diagnostics change
• Be told when a file changes
What does the GHC API do?

• GHC is the Haskell compiler

• GHC API exposes most of that as a library
  – Type checking, parsing, loading packages
  – .hi files, .hie files
  – Lots of building blocks, which are hard to use

• Also provides a dependency tracker
  – Which is mostly useless to an IDE
  – Not incremental (we had to write our own)
GHC downsweep

- GHC dependency graph is not incremental
  - Give it all files, get all results

- We want to get the dependencies of a file ourselves
  - If there are cycles, we want to still work elsewhere
  - Don’t want to have to do everything up front
  - Con: Makes TH, CPP etc harder

- Needs abstracting and sending upstream
The GHC API

• A scary place
• IORef’s hide everywhere
• Huge blobs of state (HscEnv, DynFlags)
• The GHC Monad
• Lots of odd corners
• Lots of stuff that is not fit for IDE (e.g. downsweep)
<rant />

- Warnings from the type checker
data HscEnv = HscEnv
  {hsc_dflags :: DynFlags -- 148 fields
   ,hsc_targets :: [Target]
   ,hsc_mod_graph :: ModuleGraph
   ,hsc_IC :: InteractiveContext
   ,hsc_HPT :: HomePackageTable
   ,hsc_EPS :: IORef ExternalPackageState
   ,hsc_NC :: IORef NameCache
   ,hsc_FC :: IORef FinderCache
   ,hsc_type_env_var :: Maybe (Module, IORef TypeEnv)
   ,hsc_iserv :: MVar (Maybe IServ)
  }

Wrap the GHC API Cleanly

• We want “pure” functions (morally)

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