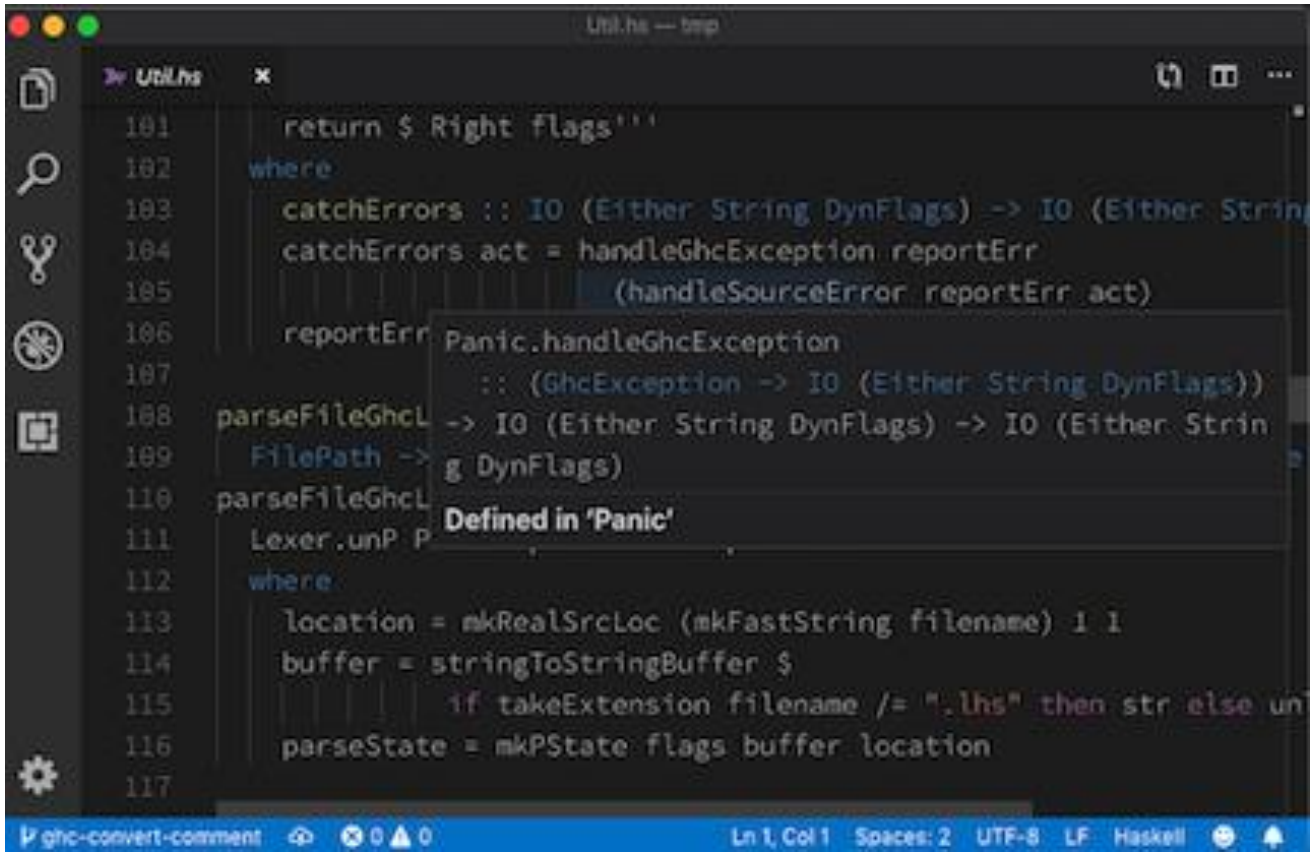
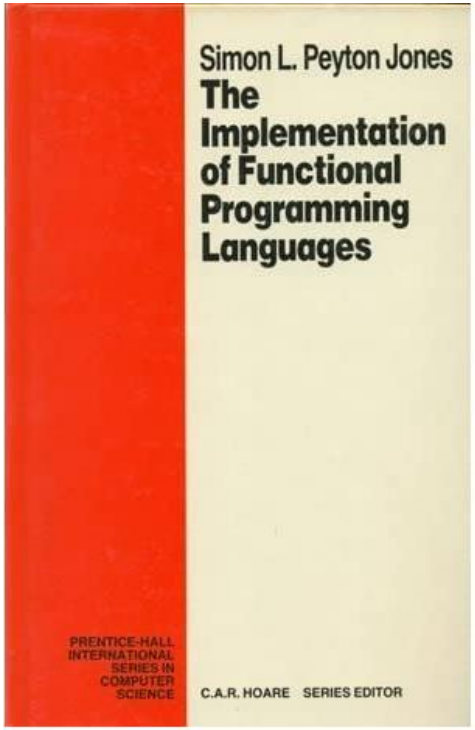
Building an IDE on top of a Build System



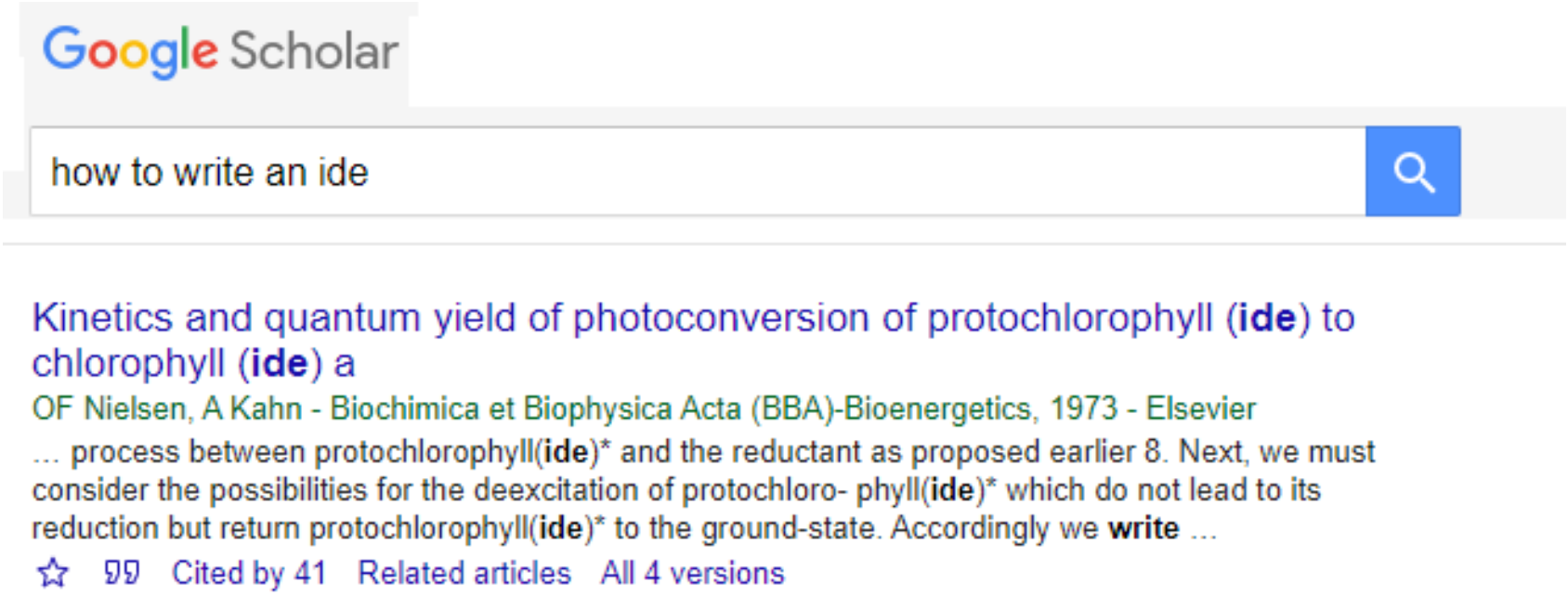
The tale of a Haskell IDE

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?



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

Demo

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

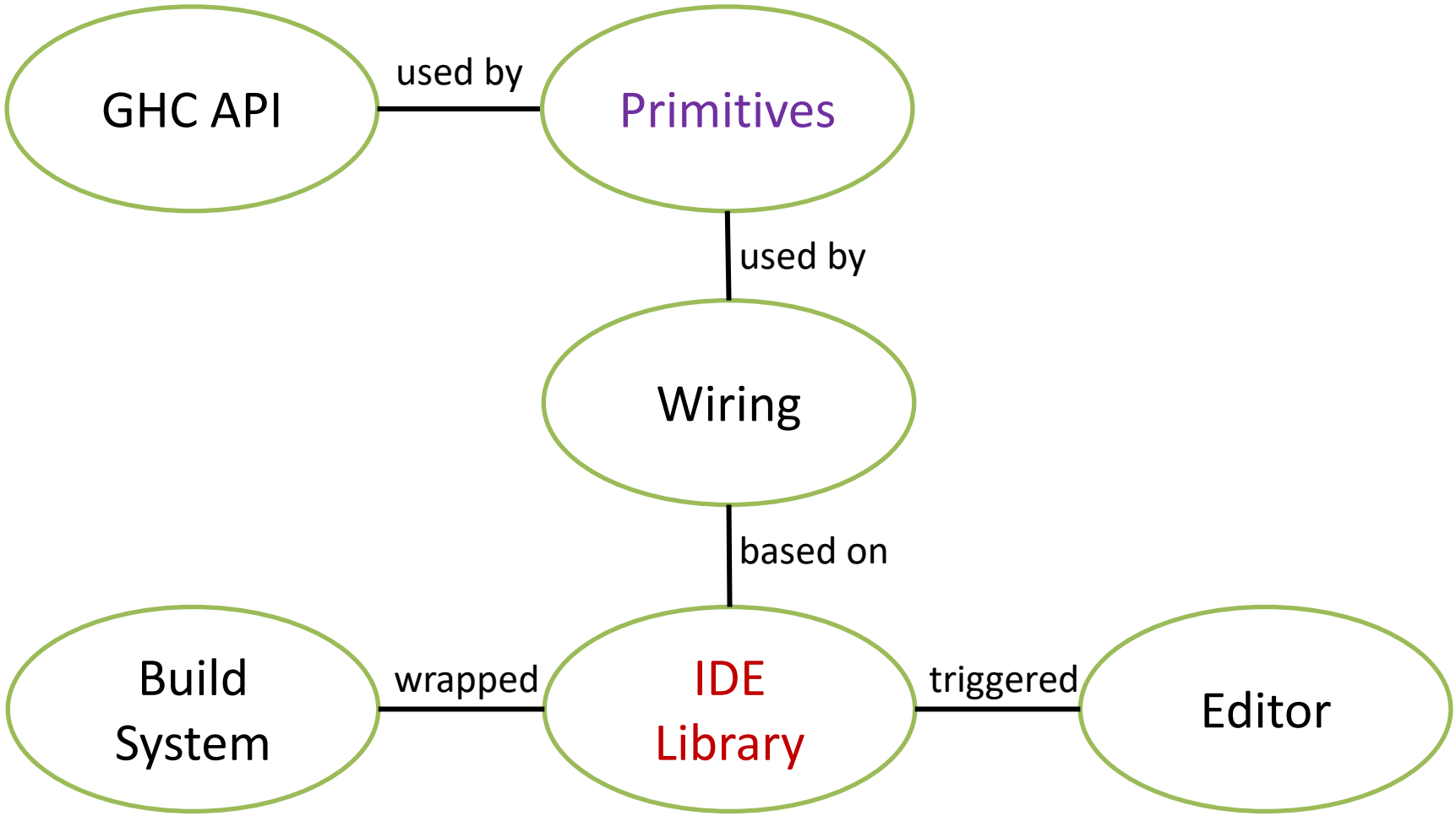
```
typecheckModule
    :: HscEnv
    -> [TcModuleResult]
    -> ParsedModule
    -> IO
       ([Diagnostic]
       , Maybe TcModuleResult)
```

TypeCheck wiring

```
type instance RuleResult TypeCheck = TcModuleResult

define $ \TypeCheck file -> do
    pm <- use_ GetParsedModule file
    deps <- use_ GetDependencies file
    tms <- uses_ TypeCheck (transitiveModuleDeps deps)
    packageState <- useNoFile_ GhcSession
    liftIO $ typecheckModule packageState tms pm
```

Architecture of an IDE



*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 $ \Phase file -> do
    use Phase file -- return the real value
    use_ Phase file -- fail if Nothing
    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?

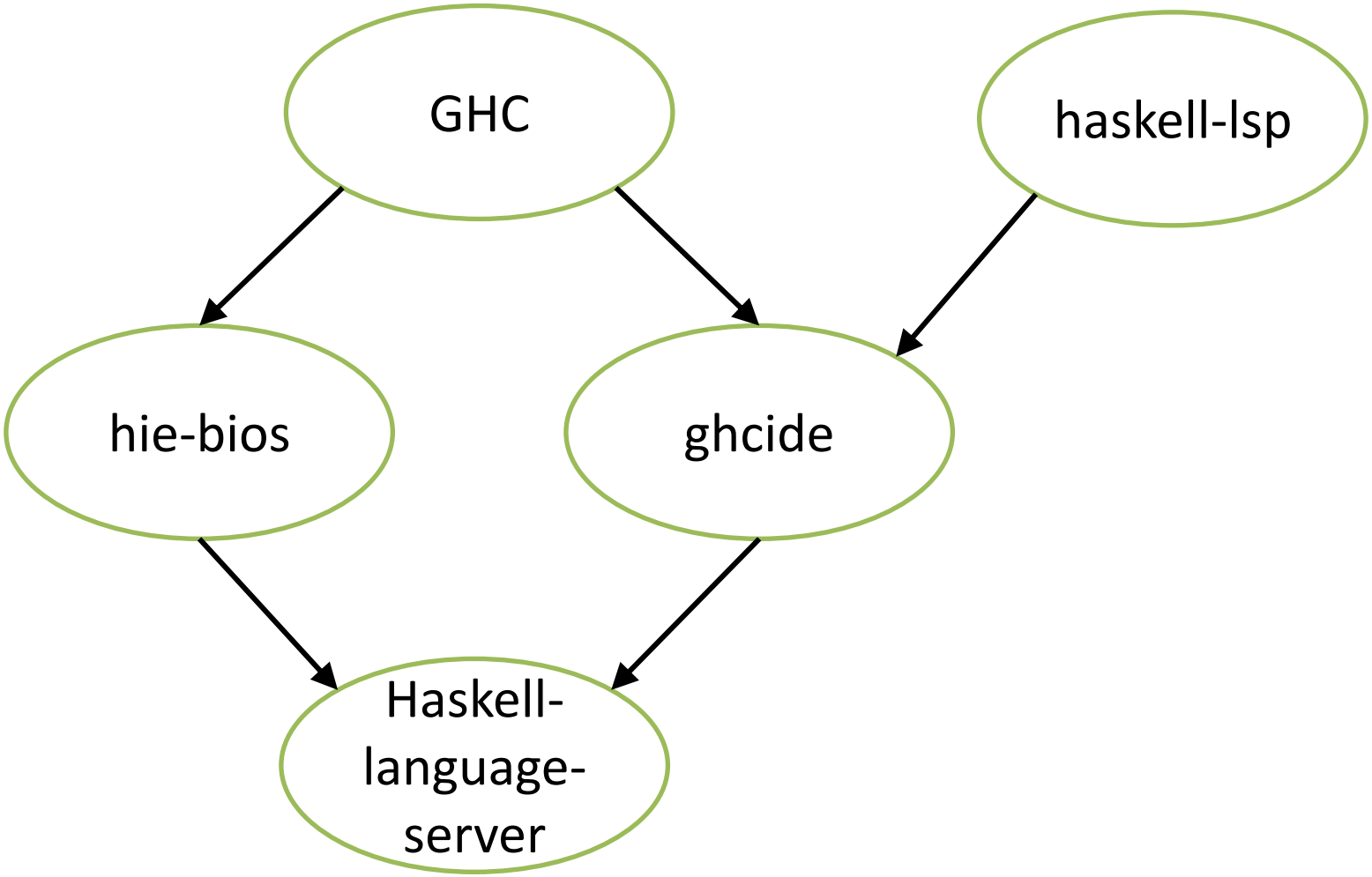
Connecting to the IDE

- 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

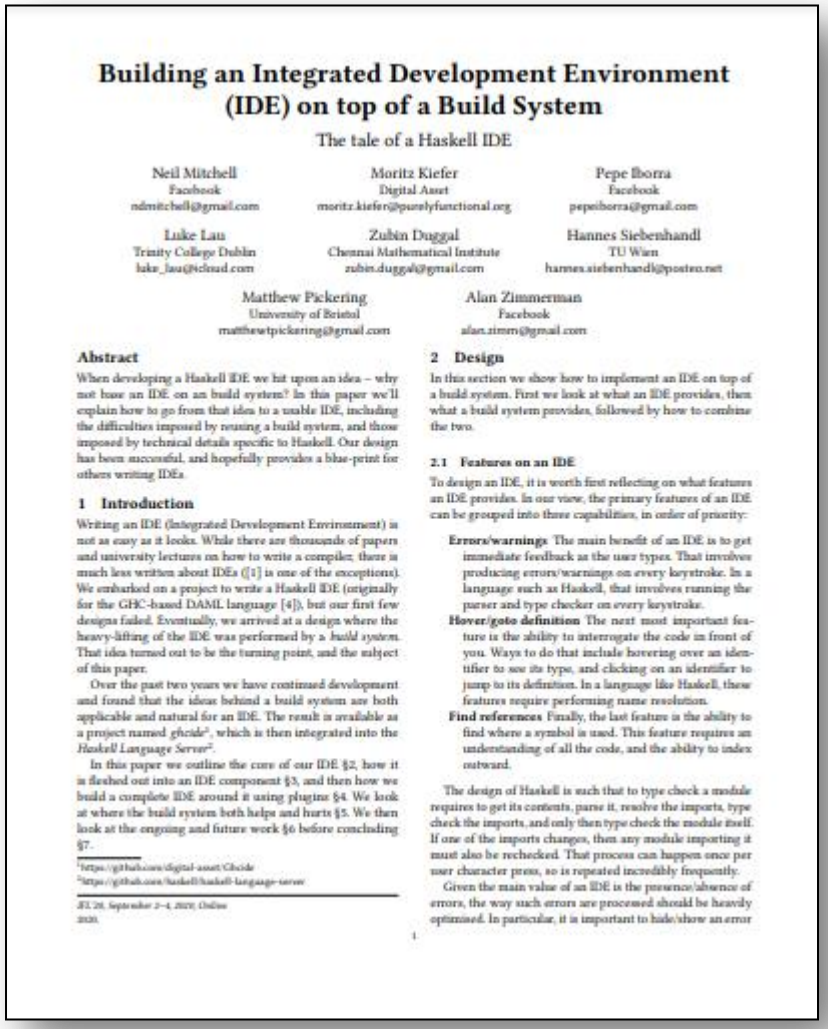


https://github.com/haskell/haskell-language-server

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?



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

IU '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 of 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

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

Neil Mitchell, Moritz Kiefer, Pepe Iborra,
Luke Lau, Zubin Duggal, Hannes Siebenhandl,
Matthew Pickering, Alan Zimmerman

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)
```