Modularizing GHC

Version 1.0

Sylvain Henry[*], John Ericson[†], Jeffrey M. Young[‡]

2022-05-03

1 Abstract

GHC is the *de facto* main implementation of the Haskell programming language. Over its 30 year history it has served well the needs of pure functional programmers and researchers alike. However, GHC is not exemplary of good large scale system design in a pure function language. Rather ironically, it violates the properties that draw people to functional programming in the first place: immutability, modularity, and composability. These scars have become more noticeable as modern projects currently underway, such as the Haskell Language Server and cross-compilation, aim to fulfill user needs and desires far more diverse than before.

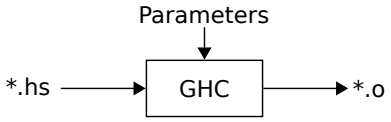
We believe a better GHC is possible. We write this paper to properly situate both the current state of GHC's codebase and that better future state in the design space of large scale, pure, functional systems. Firstly, we document in detail, GHC's architectural problems, such as low coherence and high coupling of mutable state, and their genesis. Secondly, we describe what we believe to be a superior design, drawing heavily on domain-driven design principles. Lastly, we sketch a plan to get this design implemented iteratively and durably, mentioning interactions with other ongoing refactorings (structured errors, Trees That Grow, etc.).

All of this is informed not just by our own experience working on GHC and deep dives into its history, but also by the traditional software engineering literature. The paper is written from an engineering perspective, with the hope that our collection and recapitulation may provide insight into future best practices for other pure functional software engineers.

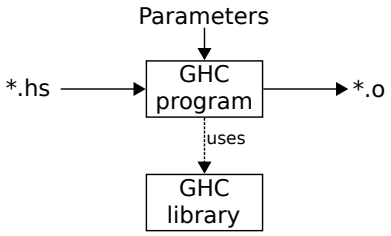
[*] IOG – sylvain.henry@iohk.io – sylvain@haskus.fr
[†] Obsidian Systems – john.ericson@obsidian.systems – inquire@johnericson.me
[‡] IOG – jeffrey.young@iohk.io

2 Introduction

GHC is a venerable project started three decades ago that has been mainly designed as a compiler *program* for the Haskell language of the time. As such and similarly to other compilers (e.g. GCC), it can be modelled as a black box—configured with some command-line parameters—that transforms Haskell source files into object code:



GHC is written in Haskell, with most of its code residing in a Haskell library (ghc-lib*). This design is common in the Haskell world as it allows the library to be reused for other purposes by other clients—which is exactly what happened to GHC too. A more precise representation is thus:



GHC extensions The Haskell language and some of its libraries are defined in the Haskell 98 [1] and Haskell 2010 [2] reports. GHC, however, provides many additional extensions to these languages and new extensions are still regularly proposed and implemented (especially via the ghc-proposals process†).

Furthermore, these extensions are *very* commonly used, so much so that any tool wanting to provide support for modern Haskell programs would have to follow GHC's development closely and reimplement its extensions. This is a lot of work! Thus, in practice, several tools directly use the GHC library (ghc-lib) instead. *GHC has not only become the de facto Haskell compiler but also the de facto Haskell framework used by other tools to manipulate Haskell programs.* For example, this is the case for: documentation generators (haddock), interactive REPL (ghci), linters (hlint), IDEs (haskell-language-server), cross-compilers (GHCJS, Asterius, Eta...), documentation example tester (doctest), etc.

The need for modularity Being written in Haskell, one would hope that ghc-lib would be modular and composable as functional programs tend to be. Disappointingly it is far from it. GHC has

* https://hackage.haskell.org/package/ghc-lib
† https://github.com/ghc-proposals/ghc-proposals

evolved quite a lot over time and, in our opinion, many new features have been force-fitted into the existing code without proper redesign. The resulting ghc-lib is difficult to understand, to use, and to modify, and also fragile (a.k.a. buggy). In Section 3 we describe some of the major deficiencies we are aware of.

While the current design isn't glorious, it works, so we could be tempted to follow the "if it ain't broke, don't fix it" motto. But as we alluded to above, there is now a proliferation of clients which depend on ghc-lib, and some of them have vastly different use cases and requirements compared to the GHC program. For example, consider the requirement distance between the GHC program and an IDE. The GHC program is typically used to execute the whole pipeline (from parsing to native code generation) on a batch of modules while, on the other end of the spectrum, an IDE executes only the frontend (parser, type-checker) possibly after every user keystroke in the code editor: these two use cases have many opposite concerns (one-shot *vs* long-running session, latency impact, cleanly interruptible or not, etc.).

A common use case of the ghc-lib is to develop Haskell cross-compilers, such as, compiling Haskell programs to JavaScript, WebAssembly, JVM bytecode etc. Most cross-compilers (e.g. GHCJS, Asterius, Eta) have been built upon GHC forks because they need to hack around some deficiencies in the stock library. However, as mentioned above, it is a lot of work to keep up with GHC's development pace and indeed these forks lag behind upstream GHC. A much more sustainable approach would be to fix the GHC library itself in order to make it more modular and composable, as required by these other compilers.

"The GHC API was used initially, but the rigidity of the API forced me to inline the entire GHC frontend into the GHCVM [Eta] codebase."—abstract of Rahul Muttineni's HIW2016 talk

Redesigning GHC? Changing GHC's design after so many years seems like a daunting task as the codebase is huge. Table 1 shows the number of lines of Haskell code in ghc-lib at the time of writing. But those numbers don't show how everything is badly entangled. A slightly better metric is the number of transitive module dependencies. For example, the Parser "component", which we have been trying to untangle from the rest of the compiler for quite some time now, still depends on 275 modules out of the 593 modules in the GHC library.‡

On the bright side, GHC is written in Haskell, and this language is particularly well suited to performing massive refactorings with confidence that nothing breaks. Thus, it should be possible to refactor the GHC library towards a more robust and versatile design.

The question becomes: what is the design we are aiming at?

‡ See the test called `CountDepsParser` in GHC's testsuite and `ghc-lib-parser` package (`https://hackage.haskell.org/package/ghc-lib-parser`)

Language	files	blank	comment	code
Haskell	593	65539	136186	214047

Table 1: Haskell code in the GHC library

GHC being a vehicle for research, we don't believe it can reach a fixed point where we would consider it complete/done. Instead, it must stay easy to refactor in order to easily experiment with new ideas. It must stay modular to allow its components to be reused at all, but especially in unanticipated ways. As such, its design is best described as a set of principles.

Our contributions:

- We first present some design flaws in GHC as well as their genesis (Section 3). An alternative title could be "what's so bad about GHC's design and API, and how did we get there in the first place". It motivates the need for changes. We also hope there are some lessons to be learned about iterative software development over long periods of time.
- Then we present design principles that we want to see applied to GHC (Section 4). These principles aren't novel: they come from Domain-Driven Design and have already proved their effectiveness in other programming circles, especially the object-oriented programming community. Our contribution is to present them to an unfamiliar audience by exposing the effects of their application to GHC.
- The proposed changes are invasive and are too numerous to be done all at once. As a consequence, it is necessary to find changes that can be done independently of the other ones. This can be tricky. We give some insights about the method to follow to implement the changes more effectively (Section 5). We also present relations with other ongoing refactorings (structured errors, Trees That Grow, etc.).

Cross-referencing anecdotes Our journey began in earnest with Sylvain wanting to use and to document GHC's internals but being unable to precisely refer to internal components because components had no boundaries. In particular there was no module hierarchy, so that what he decided to fix first, later to be followed by refactoring of the contents of the modules themselves. This proved harder than it sounds, because we didn't realize the full extent to which GHC was entangled with itself.

After some years doing this, Sylvain found Eric Evans's book "Domain-Driven Design: Tackling Complexity in the Heart of Software" [3] which was recommended by Sebastian von Conrad in an unrelated talk about Event Sourcing[1]. Unexpectedly it proved to be a very good resource, describing many issues which matched

1: Go Back to the Future with Event Sourcing and CQRS

our experience of GHC, along with remedies that matched our thoughts on what we wanted to do instead.

Yes, the Haskell community has long been dismiss of the tradition software engineering literature, due to it's overlap with the Object-Oriented Programming community. We don't dispute that famous texts like "Design Patterns" are infamous in these parts for taking a seemingly uncritical lens to OOP language features, putting the means before the ends. But zooming out of the language-specific advise, many of the basic precepts still hold up.

It's those parts of "Domain-Driven Design" [3] that we want to emphasize here, for when FP and OOP instincts agree, we have a fairly diverse consensus that something is wrong or something would be better. Many of GHC's issues have occurred in some part because GHC developers more often than not tend to spend a huge portion of their time working on GHC rather than other codebases. We want to go out of our way in avoiding the myopia of a single project, and by grounding our analysis in the work of a very different community, we hope we've achieved that.

"Domain-driven design" is a fine phrase for what we propose GHC ought to do, and so we pick up its mantle.

3 Some design defects in GHC

As far as we can tell, the current design of ghc-lib is the result of force fitting changes for three decades into the original simple model. Many changes have been implemented in what was considered the least disruptive way possible to keep the existing model untouched. But the model was touched and the end results are: (1) Code that doesn't reflect what the real model is. (2) People who continue to think that the old model is still relevant.

In this section we present some of the major flaws of ghc-lib in our opinion. Note that these flaws are purely about *software engineering concerns*. We don't comment on anything *research* related (e.g. "System XYZ should be used instead of System FC").

3.1 Shotgun parsing

GHC is particularly subject to a programming antipattern called "shotgun parsing". This antipattern is defined in [4] as:

> Shotgun parsing is a programming antipattern whereby parsing and input-validating code is mixed with and spread across processing code—throwing a cloud of checks at the input, and hoping, without any systematic justification, that one or another would catch all the "bad" cases.

```
Note [Representation of module/name variables]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
In our ICFP'16, we use <A> to represent module holes, and {A.T}
to represent name holes.  This could have been represented by
adding some new cases to the core data types, but this would have
made the existing 'moduleName' and 'moduleUnit' partial, which
would have required a lot of modifications to existing code.

Instead, we use a fake "hole" unit:

    <A>   ===> hole:A
    {A.T} ===> hole:A.T

This encoding is quite convenient, but it is also a bit dangerous
too, because if you have a 'hole:A' you need to know if it's
actually a 'Module' or just a module stored in a 'Name'; these two
cases must be treated differently when doing substitutions.
'renameHoleModule' and 'renameHoleUnit' assume they are NOT
operating on a 'Name'; 'NameShape' handles name substitutions
exclusively.
```

Listing 1: Note about representation of module holes

For instance, when new features are added into GHC, they sometimes extend or refine existing objects of the model (e.g. "Module", "Package", "Name", etc.). Instead of clearly representing the new objects (or variants of existing objects) in the code with novel constructors and types, which could require pervasive refactoring, old model objects are often kept in place but done so by modifying of what they represent.

A concrete example, which we have direct experience with, is Backpack. Since Backpack, package components (libraries, executables) may have module holes that can be instantiated with other modules. The representation of module holes in the code is documented in the Note reproduced in Listing 1.

To avoid modifying too much code by representing real modules and module holes differently, it was preferred to keep the existing `Module` datatype unmodified. Except that now we have to be careful every time we use a `Module` as it may be a hole: the name remained while the concept changed.

This is a perfect example of a change that induces "shotgun parsing". It is also antithetical to the "parse, don't validate" slogan of type-driven design [5]: after this change, every function receiving a `Module` as an argument and expecting it not to be a hole must *validate* its argument (i.e. check if the module is a hole or not and react accordingly). The compiler doesn't help in detecting where these validations must occur because the same types and constructors are used.

A typical validation example for a `Module` argument is shown in Listing 2. In Listing 2, `loadInterface` function first *validates* its

```haskell
loadInterface :: SDoc -> Module -> WhereFrom
              -> IfM lcl (MaybeErr SDoc ModIface)
loadInterface doc_str mod from
  | isHoleModule mod
  -- Hole modules get special treatment
  = do hsc_env <- getTopEnv
       let home_unit = hsc_home_unit hsc_env
       -- Redo search for our local hole module
       loadInterface doc_str
                     (mkHomeModule home_unit (moduleName mod))
                     from
  | otherwise
  = ...
```

Listing 2: Example of a function validating its `Module` argument after Backpack. In this function, the validation occurs in a guard which calls the predicate function `isHoleModule` —a classic example of the validate, don't parse anti-pattern.

input module via a guard to react accordingly to the nature of the module, i.e., whether the module is real or whether it is a hole. Notice that this behavior is not represented at the type level. If we removed this validation code the type-signature would stay the same, but it would likely manifest a *runtime* failure when trying to read an interface for a module coming from the fake `hole` unit.

This is just an example—which in this case happens to be documented—but it is quite illustrative of GHC's design decisions. It results in functions similar to functions written with untyped languages that have to check the types of their arguments at runtime to react accordingly. Writing correct code this way with no help from the compiler to detect unhandled cases is notoriously difficult and it is saddening to see this kind of code in the flagship Haskell codebase.

3.2 Command-line flags (`DynFlags`)

Command-line flags handling is another notorious example of bad design in GHC. After being parsed and loosely validated, command-line flags end up into a huge record whose type is `DynFlags`.

We use "DynFlags" as a singular or plural noun as it represents "the DynFlags record/type" but also a shorter way to write "command-line flags".

The origin of the name comes from 35fb1e38: "CmdLineOpts now separates flags into static flags and dynamic flags; dynamic flags will be passed around explicitly and can therefore change from compilation to compilation." Static flags were passed via global variables. Note that there are still three of them left at the time of writing: `-dppr-debug`, `-dno-debug-output`, and `-fno-state-hack`.

3.2.1 Layering Issues

The first code smell is that `DynFlags` are not confined to the GHC *program*, which is the user interface (UI) of GHC—the GHC *library* (ghc-lib) is `DynFlags`-ridden too! Table 2 shows that the number of

occurrences of the `DynFlags` word in the ghc-lib source is quite high and comparable to the number of occurrences of `Id` and `Var` which are common types that are part of the compiler domain!

GHC version	8.6	8.8	8.10	9.0	9.2
# `DynFlags`	1696	1712	1767	1122	802
# `Id`	1752	1817	1898	2073	2039
# `Var`	1087	1096	1122	1356	1362

Table 2: Number of lines with occurrences of `DynFlags`, `Id` and `Var` in ghc-lib. Notice that it started to decline in GHC 9.0 thanks to our work (more on that in Section 4).

XXX obtained with:

```
grep -r "\<XXX\>" compiler/**/*.hs | wc -l
```

A good code example is the following function to create an `Int#` literal in Core syntax (present up to GHC 8.10):

```
1 -- | Creates a 'Literal' of type @Int#@
2 mkLitInt :: DynFlags -> Integer -> Literal
```

Any user of ghc-lib wanting to manipulate Core syntax has to provide some `DynFlags` value to create an `Int#` literal. The underlying reason is that the size of an `Int#` depends on the target architecture (32-bit, 64-bit) and that GHC learns about the target architecture by reading its settings file whose contents is also stored into the `DynFlags` record.

A user of the API may know that in this specific case GHC only needs information about the target platform, so it could fill the `DynFlags` records with garbage values except for the platform information, but this is an unfriendly user interface. Worse still, this interface is fragile: As we will see in Section 3.2.5, when `DynFlags` went global and led to bugs.

Alternatively the code using this function could itself require its own client code (if any) to pass a `DynFlags` argument. And this is how `DynFlags` ended up being passed almost everywhere in ghc-lib without truly knowing which fields of the record were relevant to the called functions. This is very anti-modular.

3.2.2 Shotgun parsing `DynFlags`

A natural way to handle `DynFlags` would be for the "driver" code, which drives the execution of the compilation pipeline, to parse them and to react accordingly, that is to call sub-components (type-checker, renamer, code generator...) with sub-componentssystem specific options. Unfortunately, that is not the case. The codebase is written with the assumption that the GHC program is the only client of the code. Even GHCi is a second-zone client and has to go through the interface designed for the GHC program.

As another example of "shotgun parsing" (c.f. Section 3.1), most functions whose behavior depends on command-line flags just take a `DynFlags` parameter. These functions then use predicates on

the `DynFlags` argument to adapt their behavior. This is a bad practice both for the function implementer and from the function caller point of view.

The caller can't know which fields of the `DynFlags` are used by the callee function without looking at its implementation. In some cases, the call stack is so deep and indirect that it's impossible in practice to understand how the `DynFlags` end up being used. One illustrating example that we found during our work is the following comment:

```
  , hscs_iface_dflags :: !DynFlags
  -- ^ Generate final iface using this DynFlags.
  --
  -- FIXME (osa): I don't understand why this is necessary,
  -- but I spent almost two days trying to figure this out
  -- and I couldn't .. perhaps someone who understands this
  -- code better will remove this later.
```

Some `DynFlags` value had to be stored to be reused later just because understanding how it was used was too difficult (and it really was)! We were finally able to fix this but only after several seemingly unrelated refactorings.

C.f. commit c85f4928

Using `DynFlags` as a function parameter also makes the implementation of the function itself much more difficult and inherently unsafe: the number of cases to handle becomes potentially huge and the compiler can't help in checking that every possible case has been correctly handled. To illustrate this, consider the following `foo` function:

```
foo :: Bool -> Bool -> T  -- exactly 4 cases to handle
foo one_shot safe_haskell = ...
```

Just with its parameter types we know that we only have at most 4 different cases to handle (cardinality of `Bool` is 2 and $2 \times 2 = 4$). Now compare this with the same function but taking a `DynFlags` argument and using predicates on it:

We use the `Bool` type to avoid introducing more custom datatypes even if it would be better to avoid boolean blindness here.

```
foo_dflags :: DynFlags -> T  -- unknown number of cases to handle
foo_dflags dflags = ...
 ... isOneShot (ghcMode dflags) ...
 ... safeLanguageOn dflags ...
```

At the time of writing, `DynFlags` is a record with 145 fields and many of those fields have types with a cardinality greater than 2. For example one field contains the set of general purpose flags which is currently isomorphic to 191 booleans, hence has a cardinality of 2^{191}.

With so many possible cases we lose the benefits of strong typing: The pattern-match checker is of no help to ensure that functions handle all possible cases. These functions become impossible to test automatically as we can't generate a small number of arbitrary yet

meaningful `DynFlags` values to test them. Reusability is reduced, as other clients must deal with these unfriendly and unsafe function interfaces.

But the problems do not end at losing pattern-matching, robust testing, and code reuse. To see why let's put a function like `foo_dynflags` in context.

```haskell
-- | Naive implementation of the fibonacci sequence
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Listing 3: The well known naive implementation of fib in Haskell2010.

Listing 3 shows the well known naive implementation of the Fibonacci sequence in Haskell. Many Haskellers probably viewed this definition with awe at the start of their functional programming journey, and for good reason! This implementation is readable and maintainable *precisely because*: (1) We can argue inductively over the input even though the input has cardinality 2^{32}. (2) From these cases, the behavior of the function is rigorously defined, and thus the semantic domain *is also* rigorously defined. (3) From (1) and (2), our code becomes readable and understandable because the larger conceptual problem (implementing the Fibonacci sequence) has been broken into the *composition* smaller cases (implementing the 1 case of the Fibonacci sequence etc.). Lastly, the cognitive load required to understand the function is reduced because one no longer needs to lookup the predicate functions, such as `isOneShot` (these functions would be `isZero` and `isOne` for `fib`).

Furthermore, observe the asymmetry, and consequently partial ordering, between `foo_dflags` and `foo`. `foo_dflags` could be implemented by calling `foo`, because `foo_dflags` has all the information required to call `foo`. For example:

```haskell
foo_dflags :: DynFlags -> T
foo_dflags dflags
  = foo (isOneShot (ghcMode dflags)) (safeLanguageOn dflags)
```

However, the opposite case is not true. `foo` could not be implemented by calling `foo_dynflags` because the domain of `foo` (i.e., `Bool -> Bool`) *does not contain junk*[1] and thus `foo` cannot supply the types required for `foo_dynflags`.

As we argued earlier in this section, this leads to the proliferation and coupling of state-like records such as `DynFlags`. Our recommendation is to design and implement functions with a *property of least privilege*[2]. To be more precise, we argue that given two possible implementations, `foo` and `foo_dynflags`, one should prefer the implementation that is lower in the ordering of information access, or in other words; choose the implementation with the smallest domain possible.

1: Junk is not used in the mathematical sense, i.e., we are not saying that this violates the no-junk property of some initial algebra, although this may be related it is not inline with the engineering perspective we take throughout this paper. We simply mean to say `DynFlags` contains more information than required to implement `foo`.

2: A corollary to the well known principle of the same name in the security community [6].

3.2.3 When immutable becomes mutable

As we have seen, `DynFlags` has become part of the interface of many functions. Ignoring the aforementioned issues, one might still take a `DynFlags` parameter as input because one might assume that as the command-line flags passed by the user are constant during a GHC session, they are part of an *immutable* global environment and thus relatively safe. [3]

3: This is not true in an interactive GHCi session where a user may pass more command-line flags interactively, more on that later. . .

Unfortunately, this does not hold true in practice. Our experience is that some programs ended up needing to call functions which require `DynFlags` with different parameters than those passed by the user on the command-line. [4] As you might now expect, the least disruptive path was taken: Because there was no information in the type system separating the command-line constructed `DynFlags` from any other, these programs implemented two workarounds. Either they constructed a novel ad-hoc `DynFlags` and then passed this new `DynFlags` along. Or since it is difficult to create a `DynFlags` value from scratch, these programs pushed the responsibility onto the caller by requiring a `DynFlags` argument. This `DynFlags` argument would then be reused to call other functions, but only after the required `DynFlags` fields had been manually altered by the current function, with no further check of their consistency.

4: For example to implement `dynamic-too`, c.f. Section 3.4.4.

This is how the once-upon-a-time immutable `DynFlags` became effectively mutable and used everywhere. Once it contained so much information and had proliferated across the compiler, numerous functions would mutate the fields according to their needs and then pass the record along. As more and more functions implemented this pattern, `DynFlags` becomes effectively mutable. There are still some reminiscence of the good old days when it wasn't mutable. For example in `GHC.Tc.Types` we can still read:

```
1 data Env gbl lcl
2   = Env {
3         env_top  :: !HscEnv, -- Top-level stuff that never changes
```

But the comment is a lie as `HscEnv` has a `DynFlags` field that is modified in several places (see Section 3.3).

3.2.4 Why not make `DynFlags` implicit?

We have thus far described how an immutable state-like record proliferated through the compiler and became mutable as the compiler organically grew. We now provide a case study on an interesting refactoring which refactored the compiler in a *more stateful, mutable* direction.

In 2012 it seemed a good idea to also pass a `DynFlags` value *via* the Reader monad used to build every pretty-printed document in GHC: this is the `SDoc` type.

C.f. commit 330f1541

The reason was that `DynFlags` contains presentation options, for example should the compiler display uniques or not? However as `DynFlags` also happened to contain settings (e.g. the target platform) and compiler state (e.g. information about loaded packages), these fields were available to `SDoc` functions and eventually ended up being used too.

We can see why this is wrong with an example. `SDoc`s may be returned as exception messages, but in order to print the message the client code has to provide some `DynFlags` value. If this value doesn't contain appropriate settings and compiler state (such as, a different target platform), then instead of printing the expected message the output may be slightly different, totally wrong, or the printing process may just crash because of a partial function.

We have spent countless hours of work to revert this. For example we introduced an `OutputableP` type class for types that need some context (like the target platform) to be printed, only later finding that this type class was very similar to the `PlatformOutputable` type class that was removed in 2012.

C.f. commit d06edb8e

3.2.5 The genesis of a global mutable `DynFlags` variable

In the previous section, we described how `DynFlags` slowly became required even to print an `SDoc`, and how its presence in the `SDoc` code led to more coupling. A subsequent issue was that some programs printing `SDoc`s didn't have access to a `DynFlags` value, which was now required! In particular some tracing functions were used in "static" contexts, e.g., to define some CAFs (top-level values).

This was easily solved by making a `DynFlags` value always available:

```
commit ab50c9c527d19f4df7ee6742b6d79c855d57c9b8
Date:   Tue Jun 12 18:52:05 2012 +0100

    Pass DynFlags down to showSDoc

-- tracingDynFlags is a hack, necessary because we need to be
-- able to show SDocs when tracing, but we don't always have
-- DynFlags available. Do not use it if you can help it.
-- It will not reflect options set by the commandline flags,
-- it may have the wrong target platform, etc. Currently it
-- just panics if you try to use it.
tracingDynFlags :: DynFlags
tracingDynFlags = panic "tracingDynFlags used"
```

Now any function which would receive this value as an argument and would try to use it would panic; obviously a fragile design. So this got fixed on the very same day by only panicking for some less commonly used fields of the `DynFlags` record (the settings) and using some default values for the others:

```
commit 37f9861ff65552c2bb6a85c3b27e0228275bc0b6
Date:   Tue Jun 12 23:29:53 2012 +0100

    Make tracingDynFlags slightly more defined

    In particular, fields like 'flags' are now set to the default,
    so at least they will work to some extent.

-- Do not use tracingDynFlags!
-- tracingDynFlags is a hack, necessary because we need to be
-- able to show SDocs when tracing, but we don't always have
-- DynFlags available. Do not use it if you can help it.
-- It will not reflect options set by the commandline flags,
-- and all fields may be either wrong or undefined.
tracingDynFlags :: DynFlags
tracingDynFlags = defaultDynFlags tracingSettings
    where tracingSettings = panic "Settings not defined in
     tracingDynFlags"
```

It turned out some function probably tried to use the settings from their given `DynFlags` argument and panicked. This was fixed in the similar manner, with less partial fields:

```
commit cfb038de5df3fd2521987c143b3e5257d5d20055
Date:   Fri Jul 20 19:10:14 2012 +0100

    Make tracingSettings have just enough information to get
    debug output printed

    I suspect I have done the wrong thing; I hope someone can
    improve.

{-# OPTIONS_GHC -fno-warn-missing-fields #-}
-- So that tracingSettings works properly

tracingDynFlags :: DynFlags
tracingDynFlags = defaultDynFlags tracingSettings

tracingSettings :: Settings
tracingSettings = Settings { sTargetPlatform = tracingPlatform }

tracingPlatform :: Platform
tracingPlatform = Platform { platformWordSize = 4
                           , platformOS = OSUnknown }
```

The important bit is the `-fno-warn-missing-fields` option passed to GHC, which hides the warning mentioning that `tracingSettings` is still a partially defined record value. As you might expect, it turned out that some function tried to use one of the partial fields (namely the platform constants in the settings) and panicked.[5]

As there were no default platform constants to use to fill the `tracingSettings`, it was decided to bite the bullet and treat `DynFlags` according to what it had become: a global variable:

```
commit f7cd14fd30d40ae7e904a533804f43d43dd8f439
Date:   Mon Oct 8 21:55:23 2012 +0100
```

5: Tracked in GHC issue #7304 — arm-linux: Missing field in record construction `DynFlags.sPlatformConstants`

```
    Put the DynFlags in a global variable for tracing; fixes #7304

    This is an ugly kludge to make a DynFlags value available for
    the 'trace' functions. It may not be the value we really ought
    to use, but it'll be good enough for the pretty-printer to use

    Ideally we'd pass the real DynFlags down to all the trace
    calls, but this will do for now at least.

-- Do not use unsafeGlobalDynFlags!
--
-- unsafeGlobalDynFlags is a hack, necessary because we need to be
-- able to show SDocs when tracing, but we don't always have
-- DynFlags available.
--
-- Do not use it if you can help it. You may get the wrong value!

GLOBAL_VAR(v_unsafeGlobalDynFlags,
  panic "v_unsafeGlobalDynFlags: not initialised", DynFlags)

unsafeGlobalDynFlags :: DynFlags
unsafeGlobalDynFlags = unsafePerformIO $ readIORef
    v_unsafeGlobalDynFlags

setUnsafeGlobalDynFlags :: DynFlags -> IO ()
setUnsafeGlobalDynFlags = writeIORef v_unsafeGlobalDynFlags
```

Notice that the default value of the global variable is a panic again! This means that our the story isn't over yet:

```
commit 5166ee94e439375a4e6acb80f88ec6ee65476bbd
Date:   Mon Mar 16 18:36:59 2015 +0100

    Dont call unsafeGlobalDynFlags if it is not set

    Parsing of static and mode flags happens before any session
    is started, i.e., before the first call to 'GHC.withGhc'.
    Therefore, to report errors for invalid usage of these
    two types of flags, we can not call any function that needs
    DynFlags, as there are no DynFlags available yet
    (unsafeGlobalDynFlags is not set either). So we always print
    "on the commandline" as the location, which is true
    except for Api users, which is probably ok.

    When reporting errors for invalid usage of dynamic flags
    we /can/ make use of DynFlags, and we do so explicitly in
    DynFlags.parseDynamicFlagsFull.

    Before, we called unsafeGlobalDynFlags when an invalid
    (combination of) flag(s) was given on the commandline,
    resulting in panics (#9963).
```

And the expected fix similar to the ones above:

```
commit 4e98b4ff98e127aa9ef4fa1e85bdf0efa41f0902
Date:   Sat Mar 26 00:42:11 2016 +0100

    DynFlags: Initialize unsafeGlobalDynFlags enough to be useful
```

```
5
6     Previously unsafeGlobalDynFlags would bottom if used prior to
7     initialization. This meant that any attempt to use the
8     pretty-printer early in the initialization process of the
9     compiler would fail. This is quite inconvenient.
10
11    Here we initialize unsafeGlobalDynFlags with defaultDynFlags,
12    bottoming only if settings is accessed.
13
14    See #11755.
15
16 -- | This is the value that 'unsafeGlobalDynFlags' takes before
17 -- it is initialized.
18 defaultGlobalDynFlags :: DynFlags
19 defaultGlobalDynFlags =
20     (defaultDynFlags settings) { verbosity = 2 }
21   where
22     settings = panic "v_unsafeGlobalDynFlags: not initialised"
23
24 GLOBAL_VAR(v_unsafeGlobalDynFlags, defaultGlobalDynFlags,DynFlags)
```

As you might expect, some users reported unexpected panics [6]because some code tried to access the settings field. At this point you may be relieved to learn that our work allowed us to replace this global `DynFlags` variable with 3 global `Bool` variables [7] that never panic. Ideally we would get rid of them too, but it requires a lot more work.

Like most aspects of design, the better design was only realized by breaking up a large element in the design space (`DynFlags`) into understandable constituent parts, and then adding logic to handle those parts individually.

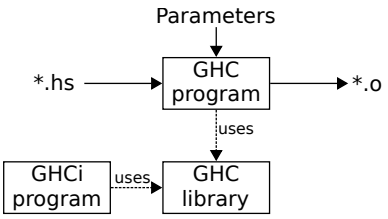
6: in GHC issue #18339 — A plugin's DynFlags not properly shared with GHC under Windows?

7: These 3 global variables correspond to 3 of the 4 "static flags" that were combined with the "dynamic flags" in commit bbd3c399: `-fno-state-hack`, `-dppr-debug` and `-dno-debug-output`. [Due to the use of global variable to store the `DynFlags`, there became no difference between dynamic and static flags.]

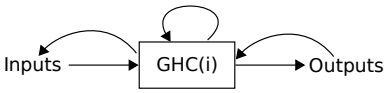
3.2.6 When immutable *really* becomes mutable: GHCi

The GHC library has mostly been designed to serve the GHC program. Hence it was architected to follow a one-shot use case which consists in executing the compiler program for a short period of time over a given set of input files to produce the output files (object code, libraries, executables. . .):

In this model, the command-line flags are constant during the whole execution (ignoring the caveats mentioned in previous sections). However, the implementation of the interactive REPL (GHCi) broke this model! One would think that GHCi would be implemented as an independent client of ghc-lib:



But instead GHCi has been implemented directly *within both* ghc-lib and the GHC program, simply with a `--interactive` command-line flag to enable it. The trouble is that GHCi provides a command to set and unset some command-line flags interactively. Thus the design now looks more like this:



That is, a very stateful design:

- inputs (command-line flags) are part of the state and are modified during the execution of the session
- outputs are also read back: e.g. produced object code is interactively linked with the GHC process itself
- the session is no longer one-shot: it lasts until the user quits the interpreter.

This is a large departure from a straightforward one-shot model. Doing some code archaeology, we can see that the feature to set the `DynFlags` interactively was broken from the very beginning (Listing 4):

Listing 4: Commit adding GHCi's setOptions feature

```
commit 459e7bd4622ea5bb8e90511b5fc6c7d8058dbd5f
Date:   Fri Nov 17 16:53:28 2000 +0000

      ...

      - :set sort of works - you can do ":set -dshow-passes", for
    example

+-- set options in the interpreter.  Syntax is exactly the same as
+-- the ghc command line, except that certain options aren't
+-- available (-C, -E etc.)
+--
+-- This is pretty fragile: most options won't work as expected.
+-- ToDo: figure out which ones & disallow them.
 setOptions :: String -> GHCi ()
-setOptions = panic "setOptions"
+setOptions str =
+   io (do leftovers <- processArgs static_flags (words str) []
+         dyn_flags <- readIORef v_InitDynFlags
+         writeIORef v_DynFlags dyn_flags
+         leftovers <- processArgs dynamic_flags leftovers []
+         dyn_flags <- readIORef v_DynFlags
+         writeIORef v_InitDynFlags dyn_flags
+          if (not (null leftovers))
+               then throwDyn (OtherError ("unrecognised flags: "
```

```
+                                    ++ unwords leftovers))
+               else return ()
+   )
```

A testimony of the fragility of the design is that the feature is still brokenin the same way 20 years later in GHC 8.10.5, as we show in Listing 5.

Tracked in GHC issue #19299 — GHCi crashes with external interpreter and user defined prompt function

Listing 5: Modifying some unexpected DynFlags interactively in GHCi 8.10.5

```
$ ghc-8.10.5 --interactive
GHCi, version 8.10.5: https://www.haskell.org/ghc/  :? for help
> :set -fexternal-interpreter
> 1
ghc: ghc-iserv terminated (-11)    <-- segmentation fault
Leaving GHCi.
```

The truth is that most systems with shared mutable state like this are very difficult to program correctly[7] and GHC, even with the safety guarentees provided by Haskell, is no exception.

To add to the difficulty, GHCi requires two sets of input flags—one for the code we type in the REPL and another for the code we load from source files—which hopefully stay coherent one relative to the other.

Additionally, as of the 9.4 release, GHC partially supports multiple home units—independent packages loaded at the same time in a compiler session—which is a welcome feature. For example it allows loading several units corresponding to several packages under development into GHCi (e.g. a package and its testsuite). However in current GHC implementation each home-unit needs to have its own set of `DynFlags`. The burden of providing compatible flags is, once again, upon the user as the compiler can only loosely check that different `DynFlags` are compatible due to their ripple effects.

This is yet another example of how modern demands placed upon GHC—in this case the simultaneous development of multiple packages—both increase the benefits of modularity and increase the costs of entanglement alike.

Note that the multiple home unit feature is yet to be made *fully* compatible with several others (GHCi, Backpack, . . .). For example, most GHCi commands aren't supported yet.

Ultimately we hope that this feature will subsume several hacks currently used, e.g. to implement Backpack support of `.bkp` files. But in the short term, it can be considered as yet another force-fitted feature making the use of `DynFlags` even more pervasive.

In a correctly layered codebase, this feature would have been implemented as an additional layer in the driver code, making use of the code already in place that handles a single home-unit at a time, and generalizing it to handle several home-units. Cf Section 4.2.3.

3.3 Top-level session state (`HscEnv`)

`HscEnv` is a datatype representing the top-level session state in ghc-lib. It is defined as follows:

```
newtype Ghc a = Ghc { unGhc :: Session -> IO a }

-- | The Session is a handle to the complete state of a
-- compilation session. A compilation session consists of
-- a set of modules constituting the current program or
-- library, the context for interactive evaluation, and
-- various caches.
data Session = Session !(IORef HscEnv)

-- | HscEnv is like Session', except that some of the fields are
```

```haskell
-- immutable.
--
-- An HscEnv is used to compile a single module from plain Haskell
-- source code (after preprocessing) to either C, assembly or C--.
-- It's also used to store the dynamic linker state to allow for
-- multiple linkers in the same address space. Things like the
-- module graph don't change during a single compilation.
--
-- Historical note: \"hsc\" used to be the name of the compiler
-- binary, when there was a separate driver and compiler.
-- To compile a single module, the driver would invoke hsc on
-- the source code... so nowadays we think of hsc as the layer
-- of the compiler that deals with compiling a single module.
data HscEnv = HscEnv
  { hsc_dflags :: DynFlags
        -- ^ The dynamic flag settings
  , hsc_IC :: InteractiveContext
        -- ^ The context for evaluating interactive statements
  , ...
```

These comments are misleading because the compilation manager (a.k.a. `--make` mode) and the interactive UI (GHCi) have been merged with the one-shot mode in 2005. Since then `HscEnv` is no longer only "used to compile a single module" but to represent the global mutable state of a GHC session (interactive or not). In particular it contains the command-line flags (`DynFlags`) and the GHCi state (`InteractiveContext`), which contains its own set of `DynFlags`.

C.f. commit 069370a5

3.3.1 `HscEnv`'s `DynFlags`

`DynFlags` stored into `HscEnv` can be modified with GHCi commands and we have already seen in Section 3.2.6 an example of incorrect `DynFlags` handling in GHCi 8.10.5.

Similarly, `HscEnv`'s `DynFlags` can also be modified before each module compilation to take into account the `OPTIONS_GHC` pragma which allows some GHC command-line flags to be set per module. This pragma in itself is a layering violation as some command-line flags are meant to be global (i.e. must apply to all modules compiled during the session) while others can really be set per module.

We return to this point in Section 4.2.4.

Listing 6 is an example of a dubious `OPTIONS_GHC` pragma, and listing 7 shows its incorrect handling still present in GHC 9.2. It shows GHC expecting an interface for a module built with the "vanilla" (static) way while trying to load it from an interface file for the "dynamic" way (`.dyn_hi`), which is nonsense. Ways are described in more details in Section 3.4.3.

```haskell
{-# OPTIONS_GHC -static #-}
module Test where

main :: IO ()
main = putStrLn "Hello World"
```

Listing 6: Dubious `OPTIONS_GHC` usage

```
$ ghc-9.2 Test.hs -dynamic
[1 of 1] Compiling Test
Test.hs:2:8:
  error: Bad interface file: .../base-4.16.0.0/Prelude.dyn_hi
  mismatched interface file profile tag (wanted "", got "dyn")
```

Listing 7: Incorrect `OPTIONS_GHC` handling in GHC 9.2 of the code in Listing 6

3.3.2 `HscEnv`'s caches

`HscEnv` is used as a shared global mutable store. In particular it contains several caches for module interfaces read from disk (external package state, EPS) or generated during the session (home package table, HPT).

One major issue is that there is only one such module environment. This is a problem with any kind of cross or multi-target compilation where we would like to distinguish modules belonging to different environments (e.g. host *vs* target, profiling *vs* non-profiling, dynamic *vs* non-dynamic).

As functional programmers are well aware, performing implicit side-effects on a single shared global mutable environment is difficult to do correctly. For example, the order used to read interfaces matters, but it shouldn't: the caches contain either too few, too much or incorrect information. We describe these problems below and provide a list of issues which result from each. We begin with "too-few-information":

too little information If the first read of a module interface is performed for a module having the `-fignore-interface-pragmas` flag set (or compiled with `-O0` as the optimization level implicitly disables/enables the flag), the interface file will be read partially (for performance reasons) and stored in the cache. The next module reading the interface will get the partial information even if it doesn't use the flag.

- #8635 — GHC optimisation flag ignored when importing a local module with derived type classes
- #9370 — unfolding info as seen when building a module depends on flags in a previously-compiled module
- #13002 — `:set -O` does not work in `.ghci` file
- #20021 — Optimization options (esp. `-O2`) in `OPTIONS_GHC` pragma can cause frustrating behavior
- #20056 — `-fignore-interface-pragmas` doesn't work well with `--make`

too much information some information retrieved to compile one module leak to compile other modules while they shouldn't.

- #2182 — GHC sessions (`--make`, `--interactive`, GHC API) erroneously retain instances
- #8427 — GHC accepts invalid program because of EPS

poisoning
- #9422 — EPT caching on `--make` can make spurious instances visible
- #13102 — orphan family instances can leak through the EPS in `--make` mode

incorrect information some information is no longer correct as the implicit ordering has been violated (e.g. after a reload in GHCi) but the cache isn't properly updated to reflect this.

- #2404 — GHCi `:r` does not reset imported class instances
- #9729 — GHCi accepts invalid programs when recompiling
- #10420 — "Care with plugin imports" is wrong / orphan `RULE` visibility (rewrite rules defined into plugins leaking into compiled modules!)

The list of tickets is not exhaustive, but it shows that issues of this kind have been bedeviling GHC users for quite some time. Some of them aren't fixed at the time of writing.

3.3.3 Code reuse

Similarly to `DynFlags` (Section 3.2), the `HscEnv` datatype isn't confined into top-level driver modules but passed to many sub-components: the type-checker, renamer, desugarer (HsToCore), Core optimizer, most code generators. Only the parser has been spared. The trouble is that it makes these sub-components much more difficult or even impossible to reuse.

For example, suppose that some user wants to compile a Haskell module into ByteCode and print it (e.g. for debugging or pedagogical reasons). The interface to generate ByteCode isn't documented (another common issue) but looks reasonable (listing 8):

```
byteCodeGen :: HscEnv
            -> Module
            -> [StgTopBinding]
            -> [TyCon]
            -> Maybe ModBreaks
            -> IO CompiledByteCode
```

Listing 8: ByteCode generation code interface

`Module` must be the module to generate ByteCode for; `[StgTopBinding]` must be the list of top-level bindings of the module in STG representation; `[TyCon]` must be the list of top-level type constructors of the module; `Maybe ModBreaks` is weird, probably something related to breakpoints so we can pass `Nothing` for now; `CompiledByteCode` looks like what we want to obtain.

So far so good, however there are two issues: why does it need a `HscEnv` argument and why is it running in the `IO` monad? It turns

out that `HscEnv` argument is used to get:

- the `Logger` that controls logging on stdout/stderr and generation of dump files
- the `DynFlags` that are used to get some information about the target: OS, way (profiling or not), word size, stack size limit, stack size, architecture registers...
- the `Interpreter` that is used to determine the interpreter way (profiling or not, redundant with the `DynFlags` above as both ways must match), allocate string literals (`MallocStrings` command), allocate wrappers for foreign calls (`PrepFFI` command).

The only way to discover which fields of the `HscEnv` argument are used is to peruse the function's code and likewise that of recursively called functions to which the `HscEnv` value is passed.

In this specific case the code generator is tied to the interpreter so that it's impossible to generate ByteCode without providing an interpreter, even if the code isn't meant to be executed. This is very anti-modular.[8] If an `Interpreter` argument was explicitly required, the clumsiness of the design would have been obvious because the types would tell us so!

8: Even more if you know that there is no interpreter command to free the allocated strings...

3.4 Interpreter

The interpreter was originally designed for the interactive user interface of GHC (GHCi) to *execute* some Haskell code. It is another component of GHC that has seen its usage extended without a proper redesign. In addition to GHCi, the interpreter was later reused to implement Template Haskell and compiler plugins (discussed in Section 3.5).

The interpreter supports executing Haskell programs compiled into both ByteCode or *into native code* (`.o`, `.a`, `.so`, `.dll`). The latter is much more difficult to do because native code is platform specific (e.g. x86-64 *vs* AArch64) and GHC supports several ABIs for the same platform (e.g. with profiling enabled or not, dynamically linked or not, etc.).

3.4.1 Internal interpreter

Historically GHC only had a single kind of interpreter, that we now call the *internal interpreter*. The internal interpreter allows the execution of native object code by loading them into the running GHC process before calling into them.

Loading is called "runtime linking" in GHC and it is implemented in the runtime system (RTS).

In order to do this, the system is subject to a major constraint: the *native object code must be ABI compatible with the compiler itself*. As such, the internal interpreter cannot be used with cross-compiled Haskell code, nor with object code produced with a

slight ABI change (e.g. with profiling enabled). In the rest of this section, we document the impact of this design decision, including workarounds and issues that have arisen from it. It is likely that some of these issues are familiar with the general Haskell community.

3.4.2 Avoiding the use of the interpreter

A trivial non-solution to the internal interpreter limitations is to avoid using features that rely on the interpreter (Template Haskell and compiler plugins). This is the solution used by GHC itself.

Suppose we have a GHC compiler program (`ghc-stage0`) that produces object code with ABI, `old_abi`. Now we use `ghc-stage0` to build a new GHC compiler program (`ghc-stage1`) that produces object code with ABI, `new_abi`. `ghc-stage1` can't support the internal interpreter because it has been built using `old_abi` but builds object code using `new_abi`: the object code it builds aren't ABI compatible with its own, so it can't load them!

The fact that this restricted `ghc-stage1` compiler doesn't support the interpreter isn't an issue because it is typically only used to build *another* GHC (`ghc-stage2`) from the same sources (hence also producing object code with ABI `new_abi`) and building GHC doesn't require an interpreter. `ghc-stage2` is built with `new_abi` and produces object code with `new_abi`, thus it supports the internal interpreter. It is the compiler that is distributed in GHC binary distributions.

As long as `stage1` compilers are used as placenta to give birth to `stage2` compilers, and they don't outlive this bootstrapping phase, then they only impose a burden on GHC developers. If they are distributed more widely, however, their limitations may backfire.

As a concrete example, cross-compilers don't support compiler plugins (#14335). !7377 implements a promising method to workaround this limitation, only requiring the presence of the internal interpreter, hence a `stage2` compiler. Sadly, at the time of writing, cross-compilers are built and distributed as `stage1` compilers (#19174) for no good reason—mostly because GHC's build system would need to be adapted to do this and its implementation makes this utterly not trivial—hence we can't use this workaround.

3.4.3 Working around "ways"

As mentioned above, GHC can produce object code with different ABIs depending on some options. For example, it can produce objects that:

- use dynamic linking or not
- support profiling (cost-centres, etc.) or not
- use additional debug assertions or not
- use different heap object representation (e.g. `tables_next_to_code`)

Some of these options are set at GHC compilation time and we don't consider them further here (e.g. `tables_next_to_code`). Others are configurable at runtime via command-line flags and are called "ways".

The trouble is that when GHC builds some object code with a different way than its own, it is actually performing some kind of cross-compilation! As we have seen above, this means that it can't use the internal interpreter to load object code it has built into its own process to execute them. That inability to load is inconvenient

because it means, in turn, that Template Haskell cannot be used. Fortunately, a workaround has been implemented.

First, GHC had to distinguish files (object code, interfaces, archives etc.) generated with different ways. For this purpose it uses tags, for example "p" for the profiled way, "debug" for the debug way and "dyn" for the dynamically linked way. Ways can be combined, and so tags can also be combined. For example, object code built with dynamic + profiling ways would have the filename extension `.dyn_p.o`.

The workaround is for the interpreter to load object code that is not the object code that is used to build the real build product. For example, supposing GHC isn't built with the profiled way but is asked to build profiled object code (via `-prof` command-line flag), if Template Haskell, GHCi or plugins are used, GHC will use both non-profiled objects (in the interpreter) and profiled objects (to build the real product).[9]

9: However, installed packages may not provide objects for all the possible way combinations as it would make compilation times and occupied disk space explode, so this scheme is fragile. E.g. #15394 — GHC doesn't come with dynamic object files/libraries compiled with profiling .

As GHC only has a single global unit environment (stored in `HscEnv`, c.f. Section 3.3), it doesn't separate interfaces read for the interpreter (non-profiled) and interfaces read for the final product (profiled). Quoting the GHC wiki[§]:

> The way this is done currently is inherently unsafe, because we use the profiled `.hi` files with the unprofiled object files, and hope that the two are in sync.

It also led to several bugs because GHC has to juggle with ways for target code, plugins, Template Haskell and GHCi, and sometimes fails to do it correctly.

E.g. #15492 — Plugin recompilation check fails when profiling is enabled

3.4.4 `-dynamic-too`

On another front, to avoid compilation times doubling because of GHC's need of two sorts of object code (one for its own way and another for the actual user-selected way), another hack was added: `-dynamic-too`. With this flag, GHC acts like a multi-target compiler and produces both static and dynamic object code in the same session.

The trouble is that the infrastructure of the compiler wasn't meant to support multi-target compilation and has not be redesigned to do so. Quoting the wiki again[¶]:

> `-dynamic-too` is buggy, slow, and has an ugly implementation

In addition, it is a very limited and ad-hoc multi-way support because it is only available for the dynamic way, not the other ones. For example there is no equivalent `-profiling-too` flag.

[§] https://gitlab.haskell.org/ghc/ghc/wikis/remote-GHCi

[¶] https://gitlab.haskell.org/ghc/ghc/-/wikis/dynamic-linking-debate

Putting it all together, listing 9 shows an actual excerpt of GHC code obtained by combining bug-driven development[‖], shotgun parsing and mutation of `DynFlags` (Section 3.2), ad-hoc (and dubious) handling of compiler ways, automatic activation of a buggy feature (`-dynamic-too`). Notice that this code also mentions the external interpreter which we discuss in the next section.

```
-- #8180 - when using TemplateHaskell, switch on -dynamic-too so
-- the linker can correctly load the object files.  This isn't
-- necessary when using -fexternal-interpreter.
dflags1 = if hostIsDynamic && internalInterpreter &&
             not isDynWay && not isProfWay && needsLinker
          then gopt_set lcl_dflags Opt_BuildDynamicToo
          else lcl_dflags

-- #16331 - when no "internal interpreter" is available but we
-- need to process some TemplateHaskell or QuasiQuotes, we
-- automatically turn on -fexternal-interpreter.
dflags2 = if not internalInterpreter && needsLinker
          then gopt_set dflags1 Opt_ExternalInterpreter
          else dflags1
```

Listing 9: Automatically enabling -dynamic-too

3.4.5 External interpreter

The main issue with the internal interpreter is that it can only load object code that is ABI compatible with itself. So the idea behind the external interpreter is to delegate the execution of the code to another process (called `iserv`).

The `iserv` process may use a different ABI than the compiler, allowing it to load object code that the compiler can't! It can even itself delegate execution of the object code to another process in a virtual machine (e.g. `qemu`, `wine`) or on a remote machine with an appropriate architecture.

Ideally, for each way combination a corresponding `iserv` program should be available. By default, GHC tries to spawn a different `iserv` process depending on the selected ways: `ghc-iserv-prof`, `ghc-iserv-dyn`, etc. It could perhaps build these programs on demand but it's not currently done. Alternatively a custom external interpreter program can be specified with the `-pgmi` command-line option.

The major advantage of the external interpreter design is that it cleanly decouples the compiler and the interpreter: both use separate processes with different runtime systems, loader states, etc. However, this comes with a different, albeit more acceptable, major constraint compared to the internal interpreter: *a communication protocol is needed between the compiler and the interpreter*. In particular

[‖] https://en.wikipedia.org/wiki/Tester-driven_development

transferred data must be serialized, hence be serializable.

Plugins It has been possible to work out a protocol for GHCi and Template Haskell. However, it hasn't been possible to devise a protocol to make the external interpreter support plugins. Some of the reasons are:

- some plugins have access to `HscEnv` (Section 3.3) which would have to be serialized and this would be very difficult
- Core representation is cyclic (c.f. tying-the-knot**) hence making it more difficult/costly to serialize, e.g. for plugins manipulating Core.
- We would have to deal with ABI differences such as different word sizes which would be quite difficult: a Word in the compiler may not have the same size as a Word in the interpreter.
- GHC stores global variables (for string interning, for unique number generation) into the runtime system itself. So we would have to work out a way to synchronize both compiler and interpreter runtime systems.

As a consequence, *plugins aren't supported when the external interpreter is used*.

#14335 — Plugins don't work with -fexternal-interpreter

A theoretically easy way to fix this would be to use the external interpreter for everything except for plugins which would still use the internal interpreter. However, as GHC has grown with the implicit assumption that there was a single interpreter, this fix is in fact far from trivial to implement, and was one of the first motivations for writing this paper.

3.5 Plugins and Hooks

Plugins were already mentioned several times in previous sections. To recap, plugins use the same interpreter as Template Haskell and GHCi. As GHC currently only supports a single interpreter instance at a time, plugins don't work when the external interpreter is used (Section 3.4.5). As GHC only supports a single unit environment (EPS/HPT etc.), module interfaces loaded for plugins are mixed with target code module interfaces with risks of unsound intercontamination (c.f. Section 3.3.2).

The initial plugin implementation was for a single kind of plugin (custom Core-to-Core pass[10]). It then grew up "organically"[11] from this point, without redesign, to support several kinds of plugins: type-checker, renamer, interface loader, Template Haskell splice modifier, even "`DynFlags` plugins" whose initial purpose was to setup hooks (c.f. "hooks" later in this section).

10: commit 592def09

11: The suboptimal "organic growth" was discussed after Moritz Angerman's "More Powerful GHC Plugins" talk at the Haskell Implementors Workshop (HIW) in 2016.

** `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/tying-the-knot`

As a result, we still use *a single record* (`data Plugin`) to support plugins for very different parts of the compiler. As you might imagine, this datatype is directly used by each of those parts of the compiler, creating artificial dependencies between parts that ought to be independent of each other. [12]

12: C.f. #18921 — Refactor hooks and plugins

Hooks The "hooks" mechanism provided by the GHC library is very similar to plugins. Hooks also allow the customisation of some compiler operations. They have been introduced to accommodate GHCJS's needs.[13]

13: GHCJS is a Haskell to JavaScript compiler based on GHC. Hooks have been added in commit 6f799899

Hooks are very similar to plugins but they can't be setup similarly to plugins (i.e. via `-fplugin` command-line flag). They can only be set via direct use of the ghc-lib API.

Hooks were of course stored into the `DynFlags`. Hence the next obvious step was to add a new kind of plugins: `DynFlags` modifying plugins which allow custom modification of the `DynFlags` when plugins are loaded... [14]

14: 900cf195

Our work led us to store hooks with the rest of the session state in `HscEnv`.[15] We replaced `DynFlags` plugins with "driver" plugins—that allow arbitrary modification of the `HscEnv`—to still allow plugins to set hooks. In retrospect, a better solution would have been to only allow plugins to set hooks instead of giving them access to the whole `DynFlags` or `HscEnv`. By giving them access to the whole of `DynFlags` or `HscEnv` we risk calcifying and coupling `DynFlags` or `HscEnv` just as `SDoc` became accidentally coupled to `DynFlags` (Section 3.5). Fixing this would be good, but it could break existing plugins that have already realized this coupling and are doing more than setting hooks. Or hooks could be transformed into proper plugins as GHC also supports "static plugins"—plugins that can only be set via the GHC API[16]—making the hooks mechanism redundant.

15: ecfd0278

16: da05d79d

3.6 Template Haskell

Template Haskell (TH) allows the execution of Haskell code at compilation time to transform the abstract syntax tree of a Haskell module when it is compiled.

TH has been implemented by reusing the interpreter already used by GHCi. The following citation from the first Template Haskell paper [8] shows that the (internal) interpreter has been reused:

> "When a compile-time function is invoked, the compiler finds its previously-compiled executable and dynamically links it (and all the modules and packages it imports) into the running compiler.
>
> A module consisting completely of meta-functions need not be linked into the executable built by the final

> link step (although `ghc --make` is not yet clever enough to figure this out)."

GHC still isn't clever enough after 19 years, but it might soon: see the ExplicitSpliceImports GHC Proposal 412 with which GHC could distinguish compile-time and runtime module dependencies.

As mentioned in the quote above, Template Haskell doesn't distinguish modules that are only used at compile time (Haskell code executed with the interpreter) from modules needed at runtime. As such, modules used only for Template Haskell at compile time are conservatively linked with the final binary product, making it slower to load for no good reason.

In addition, we can't use two different versions of the same package — one for Template Haskell and the other for the final build product — because GHC uses of a single module environment (c.f. Section 3.3.2).

Side effects Template Haskell splices can execute any IO action. This ability is a security concern because it allows arbitrary code to be executed at compilation time. It is also an issue when cross-compiling and using the external interpreter (Section 3.4.5) because the environment into which IO actions are executed isn't well specified. For example, a splice could try to read a file that is not available from the external interpreter environment (e.g. due to sandboxing, remote execution, or execution in a VM).

3.7 The Driver

In GHC the "Driver" is responsible for orchestrating other compilers and linkers:

- HSC: which was the old name for the Haskell to C compiler
- GCC: to compile C files into object files
- Linker: to link object files

Over time the driver was extended to support compiling multiple modules ("managed or `--make`" mode), to support GHCi interactive context, etc.

"A lot of code has been added to the Main component; this is partly because there was previously a 3,000-line Perl script called the "driver" that was rewritten in Haskell and moved into GHC proper, and also because support for compiling multiple modules was added." [9]

The Driver's main datatype is `HscEnv`: it's the top-level session state. As we already discussed in Section 3.3, its main issue is that it is leaking into many unrelated components of the compiler. For example, the type-checker directly manipulates `HscEnv`.

This lack of abstraction is the main issue but it's not the only one. We identified the following other issues:

It isn't independent of GHC-the-program command-line interface most driver functions require a `HscEnv` argument which can only be created with an undocumented function in the `GHC.Driver.Main` module:

```
1 newHscEnv :: DynFlags -> IO HscEnv
```

It isn't self-consistent

(1) passing a valid `DynFlags` value is difficult as its "settings" field has to be properly setup. Most users probably rely on `initGhcMonad :: GhcMonad m => Maybe FilePath -> m ()` in the `GHC` top-level module or duplicate its code to avoid dealing with the `GhcMonad` abstraction.

(2) the `HscEnv` created by this function is useless for most purposes because several fields (unit env, interpreter...) have to be properly initialized, which can only be done with `setSessionDynFlags ::GhcMonad m => DynFlags -> m ()` or the similar `setProgramDynFlags`) also in the `GHC` module. Or by duplicating their code.

Documentation is often missing, outdated, or incomplete

(1) the `GHC.Driver.Main` module is documented as the "Main API for compiling plain Haskell source code.", however, as we have seen, it can't really be used alone and a module further up in the hierarchy has to be used.

(2) the undocumented use of `HscEnv`'s fields by each function make their behavior impossible to predict.

The interface is inherently unsafe setting `DynFlags` is the main way to tweak `HscEnv`. However `DynFlags` isn't a set of flags but a record, hence "flag" validation is very weak. As a result, users of the API can't know which flags they are allowed to modify at any point during a session, nor if a flag has been taken into account or ignored.

It isn't full-featured some clients of the GHC API (e.g. HLS, Haddock) have to (re)implement a complex driver of their own, often duplicating code from GHC's one.

4 Refactoring GHC using Domain-Driven Design

So far, in the previous, we've described key problems with GHC. Here, we describe what we want to do to remedy then. We will quote several excerpts from "Domain-Driven Design: Tackling Complexity in heart of Software" by Eric Evans [3]. This book introduced "Domain-Driven Design" concept and principles which prove to be very relevant to the issues we want to solve in GHC.

There are many principles in domain-driven design, but we believe that even just getting the main ones[1] implemented into GHC would already be a major improvement over the status quo. Namely we will describe the benefits and the required changes to follow these domain-driven design principles:

1: Some domain-driven design principles are already at the core of Haskell programming (pure functions, value objects, smart constructors as `factories`, etc.) and we don't mention them at all.

- Ubiquitous language: use consistent and precise terminology in code and documentation, and make it apparent at type-level (Section 4.1)
- Domain isolation with layered architecture: use layering to avoid spaghetti or lasagna code[††] (Section 4.2)
- Supple design: make code "a pleasure to work with, inviting to change" (Section 4.3)

Each subsection goes into more details for each principle and relates it to GHC issues mentioned in Section 3. We also present the work already done to implement them and the work left to be done.

4.1 Ubiquitous Language and Type-Driven Design

The *ubiquitous language* principle consists in using *precise* domain terminology *consistently* in code, in documentation, and in speech. In Haskell we can extend this principle to require that domain terminology be represented at type-level in the code: *type-driven design*. By doing this, functions have *intention revealing interfaces*, expressed in the ubiquitous language, and checked by the type-checker.

"A project faces serious problems when its language is fractured. [..] The terminology of day-to-day discussions is disconnected from the terminology embedded in the code" [3] p. 25

Ubiquitous language in GHC GHC lacks an ubiquitous language: some words used to describe the domain model are ambiguous or are not used consistently, even among GHC developers.

As an example, the concept of a "package" changed from being "a set of modules whose corresponding code objects are bundled into a single library" to something much more fuzzy: A Cabal package may now contain several library components and each of them can be compiled into different units (depending on compilation options, dependencies, etc.); modules of a unit are bundled into a single library. Nowadays GHC mostly deals with units instead of packages but the code was and still is using the old terminology in many places.

In this example, using correct terminology would allow us to make obvious that GHC's user interface is ambiguous (e.g. `-package-name` may refer to different units), that GHC's "package-qualified imports" feature is similarly ambiguous, that some cabal-install's "projects" could perhaps be better subsumed by a hierarchical package component namespace, etc.

"Persistent use of the ubiquitous language will force the model's weaknesses into the open." [3] p. 26

Type-driven design in GHC GHC also doesn't fully exploit type-driven design: a lot of functions are partial; domain concepts aren't always represented at type-level; shotgun parsing is used a lot.

[††] https://en.wikipedia.org/wiki/Spaghetti_code

As an example, in Section 3.1 we showed that the concept of a "module" got extended for Backpack to support "module holes". This wasn't reflected neither in the language, nor in the model, nor in the code.

Suppose we had introduced separate types to distinguish concrete modules from "holey" modules, then we wouldn't need to intertwine hole module resolution with concrete module interface loading in Listing 2 (an example of "shotgun parsing"): module interface loading only makes sense for concrete modules and we could express it in the type of `loadInterface`.

Recommended concrete action In light of these problems, our recommended concrete action is *to make naming precise, consistent, documented, and checked by the type system*, adding new datatypes and refactoring the existing ones as needed.

It should be no surprise to Haskellers that adding more information into the type system yields numerous benefits: it helps create maintainable and checkable code, it makes obvious and explicit the interactions between different features and phases of the compiler, and it allows GHC developers to more frequently *think with types* during GHC's development.

We recommend that GHC's developers get acquainted with the "Parse, don't validate" slogan coined by Alexis King in [5], and with the type-driven design principles underlying it. This cited paper also gives several advices that we encourage every Haskell developer to follow.

4.2 Layering and Componentization

An important domain-driven design principle is to divide complex software into conceptual layers.

> "The essential principle is that any element of a layer depends only on other elements in the same layer or on elements of the layer 'beneath' it. Communication upward must pass through some indirect mechanism" [3] p. 69

There are four fairly standard layers. Adapting layer's descriptions from [3] p.70, they can be described from top to bottom as follows:

- **User Interface** (or **Presentation Layer**): responsible for interaction with the user (or with another system)
- **Application Layer**: defines the jobs the software is supposed to do and directs the expressive domain components to work out problems. Thin layer that doesn't contain business rules or knowledge.
- **Domain Layer** (or **Model Layer**): responsible for representing

concepts, rules, and information about the business. State that reflects the business situation is controlled and used here, even through the technical details of storing it are delegated to the infrastructure. *This layer is the heart of business software.*
- **Infrastructure Layer**: provides generic technical capabilities that support the higher layers.

"Partition a complex program into layers. Develop a design within each layer that is cohesive and that depends on the layers below. Follow standard architectural patterns to provide loose coupling to the layers above. Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code." [3] p. 70

As a first approximation, we can map these layers onto GHC like this:

- **User Interface**: code in GHC-the-program, ghci, HLS, or any other client of GHC-the-library.
- **Application Layer**: "driver" code that deals with constructing and executing build plans, with recompilation avoidance, with interactive contexts, etc.
- **Domain Layer**: code responsible for representing concepts, information, and rules about compiling Haskell programs: that's the commonly represented pipeline with its different IRs (Haskell syntax, Core, Stg, Cmm...) and their transformers.
- **Infrastructure Layer**: code supporting technical operations: logging, dealing with file-systems (finder, interface loader, etc.), reporting errors (panics)...

The most important layer to isolate from the rest is unsurprisingly the domain layer. It's the heart of GHC as it contains the business code that makes GHC useful in the first place: the code to compile Haskell sources into something else. This layer implements parsing of Haskell source files, type-checking, compilation into several intermediate representations (Core, STG, Cmm, ByteCode), code optimizations, production of machine code... It's the code documented in research papers and books. [2]

2: Other than this very paper, no one ever discusses `DynFlags` and `HscEnv`, which is a good indication that they don't belong to the domain layer.

Componentization Domain-driven design is usually applied to Object-Oriented Programming. As such, it assumes that "objects" are used to provide encapsulation at a finer level than layers. Functional languages such as Haskell don't enforce this kind of fine encapsulation. Nevertheless we believe it is beneficial to reach for it, as a complement to layering, which is itself a coarser encapsulation tool.

Instead of using the term "object", which would be misleading, [3] we use the term *component* to refer to this level of encapsulation. A component is a conceptual group of functions, types, type-classes. Each component should probably be put into its own module to avoid accidental coupling.

3: Most of our components are singleton objects.

Components and their dependencies form a directed acyclic graph (DAG). Layers and their dependencies also form a DAG, but a much simpler one. As such, the layer DAG present a collapsed view of the component DAG. [4]

4: This collapsing is a graph homomorphism.

Discussing where to draw layer lines—does component A belong to layer X or Y?—isn't our main concern. Some layering must be observable, however, otherwise it means that all the components are entangled into the same layer. It happens to be the situation we started with in GHC (c.f. Section 3).

In the following subsections, we discuss each standard layer in turn from the bottom up, and for each layer we discuss more fine-grained componentization.

4.2.1 Infrastructure Layer

The Infrastructure Layer provides services to other layers. Services in GHC include message logging, unique number generation, management of temporary files, file finder, etc.

To avoid tight coupling between these services and code from other layers, a simple solution in Haskell is (1) to create datatypes representing service components when they don't already exist (2) to pass them as arguments.

In the following example, we can see that function `foo` requires (and probably uses) two services: `Logger` and `TmpFs`.

```
1 foo :: Logger -> TmpFs -> ... -> IO ...
```

In current GHC, many similar functions would have the following prototypes instead because services behavior may be configured via command-line flags and some of their state may be stored in the session environment:

```
1 foo :: DynFlags -> ... -> IO ...
2 -- or
3 foo :: HscEnv -> ... -> IO ...
```

A significant part of our work is to refactor functions that have the latter interfaces (`DynFlags` and `HscEnv` parameters) into functions with the former interface (explicit passing of each service).

This loose coupling buys us several things:

- **Modularity**: Different clients of the function may pass different implementations of the services. Clients of the services are oblivious of their implementation.
- **Documention**: Type signatures clearly indicate services a function uses. If a service isn't passed to a function, we can be certain it isn't used by it.
- **Exposition of suboptimal interfaces**: in Section 3.3.3 we've presented an example of a function using a component—the interpreter—unexpectedly. Suboptimal interfaces like this appear for what they are if services are passed explicitly. This becomes a nudge to refactor the code.

Caveat warning Passing several services to a function can look cumbersome: why not bundle some of them into a single record (say `XYZEnv`)? In our experience, there is no one-size-fits-all record like this and the pattern is only a local maxima, instead, there is a slightly different record best suited *for each* function. Thus, the drive to bundle these services into a single record is exactly how coupling begins. The sequence of events is:

- Services are bundled into records such as `XYZEnv`.
- Most of the functions which input `XYZEnv` use most of its services or heavily related and thus have high coherence.
- The code base evolves by adding new functions or features; some functions require more than what is in `XYZEnv`, but still require some services that exist in `XYZEnv`.
- This produces an incentive to create new slightly altered `XYZEnv`, or expand the existing `XYZEnv`. Creating more `XYZEnv`s is cumbersome, and appears redundant, so the "path of least resistance" is taken and an existing `XYZEnv` is extended with whatever new fields are required for just the new functions.
- Now that `XYZEnv` has grown, there are two secondary effects: First, coherence is reduced because the number of functions which use all of the `XYZEnv` fields is lower. Second, there is more incentive to pass `XYZEnv` around because its functionality has expanded.
- And now we arrrive at a vicious cycle; more and more services are added into `XYZEnv` because it is conveniently threaded through many functions. But it is threaded through many functions because of the many services it provides.

This is how we ended up threading `DynFlags` and `HscEnv` everywhere and storing arbitrary values into these records!

To avoid this slippery slope, our recommendation is to pass services explicitly as much as possible in the library. This approach is the best we are aware of because it doesn't impede the implementation of any other interface on top of it *in the client code*:

- bundled services:
  ```
  foo :: XYZEnv -> ... -> IO ...
  ```
- tagless final style:
  ```
  foo :: (HasLogger m, HasTmpFs m) => ... -> m ...
  ```
- effect system:
  ```
  foo :: ... -> Eff '[Logger,TmpFs] ...
  ```

As will be discussed later in Section 5, we never want to be in a scenario where we need to rely on GHC developers suddenly changing their behavior after many decades. Rather, we want to shift the incentives they face every time they sit down to edit the code. If the "path of least resistance" is far less often the enemy of modularity, our reforms will be more durable to bit rot, and our developer colleagues ultimately less inconvenienced.

4.2.2 Domain Layer

The core business of an Haskell compiler is to compile Haskell code into machine code. All of this is essentially performed in the domain layer. Figure 1 depicts GHC's compilation pipeline used to achieve this.

As discussed in the introduction of Section 4.2, layers can themselves be subdivided into independent components to form a DAG. For modularity, it is particularly important to make each phase of the compilation pipeline an independent component that can be easily reused for other purposes, whatever these purposes may be (c.f. "Supple design" in Section 4.3), or easily replaced.

Doing this would be a boon for research and experimentation because changes could be precisely localized, with minimal impact on the rest of the compilation pipeline. It would also allow other compilers to reuse the frontend—Haskell parser down to Core or STG—and/or the backends—Core/STG/Cmm to machine code. GHC could become a compilation framework similar to LLVM but for functional languages.

To achieve this, there must be very low coupling between the phases of the pipeline. Ideally boxes in Figure 1 would represent independent components and some arrows would represent allowed coupling, depending on each box kind: IR components are standalone; optimiser components only depend on a single IR component; compilers depend on two IR components. The resulting DAG is shown in Figure 2.

It turns out that phases and representations were far from being independent in GHC. However a lot of work went into improving this (c.f. Section 5).

Extended Domain Layers The compilation pipeline is unarguably the main element of the domain layer. However it only deals with a single compilation unit (a module). To support separate compilation, we have to introduce more components: module names, module imports, module interfaces, module boot files, module signatures, etc.

All these new components don't necessarily belong to the same layer as the compilation pipeline for a single module. They could all be placed a layer above, in some "extended domain layer". Similarly, code handling packages/units with thinning, renaming, module hole instantiation, etc. could be placed in another "extended extended domain layer".

This kind of loose coupling would be useful to experiment with alternative separate compilation schemes. For example, decoupling the renamer from the extended layers by making the method to resolve external names pluggable could facilitate the implementation

of a Unison-like content-addressed model[5] for Haskell.

Currently, all these layers are unfortunately mixed into one. In addition some concepts aren't properly named and abstracted. For example, we don't have a name for the view that a unit has of other units, which may be different from one unit to another due to thinning and renaming of modules.

The recently introduced support for multiple home units is incompatible with thinning and renaming features probably because this lack of abstraction makes their interaction non trivial. Similarly, most interactions between components of these extended domain layers aren't well specified nor checked, making bugs mentioned in Section 3.3.2 very likely to occur.

5: Unison's "The big idea" page introduces this well.

4.2.3 Application Layer

The Application Layer uses the components of the Domain Layer to provides higher level services. In GHC the Application Layer is called the "Driver" and provides services such as: retrieving dependencies between a set of modules, running passes separately (parser, type-checker, code generation, etc.), performing interactive evaluation... Many of this layer's issues have been exposed in subsections of Section 3, especially 3.3 about `HscEnv` and 3.7 about the driver evolution over time.

The most important issue to fix is the lack of layering and componentization. For example, it should be possible to use some components of the application layer without ever using `HscEnv` or `GhcMonad`. Proper componentization would make obvious the services, the caches, and the settings each component requires. It would also allow a client of these components that don't need some features (e.g. the interactive context) to totally ignore them.

Note that all this requires the Domain Layer to be free of `HscEnv` first, c.f. Section 4.2.2. Otherwise components of the application layer would inherit wrong dependencies from the components of the domain layer they use which still have them. These dependencies explain why we follow a bottom-up approach to fix these layering issues (c.f. Section 5).

Examples of components of the application layer are:

- "downsweep": a component that uses the header parser of module files to generate a module dependency graph. It should also return header pragmas for each module.
- "Haskell pipeline manager": a component to manage Haskell compilation pipelines. The domain layer provides components performing the real work (compilation phases), but the choice of the components to use and their composition can be left to the application layer to do.
- "Task graph": a component that manages a meta compi-

lation pipeline that encompasses: Haskell compilation, C compilation, linking, module instantiation, etc.

- "Interactive context": everything needed to manage an interactive session similar to GHCi's one.

The task graph is currently named "module graph". It's another example of an inherited non longer precisely relevant terminology. C.f. Section 4.1

Further steps include deduplicating code between features, and separating "planning" from "executing".

The driver is full of overlapping functionality. At the end of section 3.2.6, we mentioned how the newly implemented "multiple home units" overlaps with other features, but could subsume them. Indeed, various backpack and GHCi functionality can and should be subsumed, and multiple home units support itself can be refactored to have a much stabler foundation, once componentization is complete. This will ensure we have not only more functionality with less code, but better tested higher quality code.

Separating "planning" vs "execution" is best approached from two different perspectives.

The first perspective is that the driver as it exists today has too much infrastructure-layer concerns managing concurrent jobs mixed in and polluting the actual application-layer logic coordinating what needs to be done. The latter is planning, while the former is used to execute those plans. As always, we want to properly layer and keep the core logic together rather than lost in a say of miscellaneous details.

The second perspective is different clients of the GHC library need the same plans, but will execute them in markedly different ways. HIE will have all sorts of data in memory and wish to leverage that as much as possible. GHCi will be in a similar boat. Even within "tradition" GHC-the-program, "one-shot mode" and "batch mode" work in profoundly different ways. The best way to think about the former is that rather than scheduling tasks within GHC itself, the plan is returned to an external build system[6] which then invokes GHC per task. Thus, plan execution is managed by a completely separate process, an extreme case of planning vs execution separation.

6: today via Makefiles, but perhaps tomorrow via something like GHC Proposal 245 — Extended Dependency Generation.

With more execution work at least optionally left to the clients, GHC should double down on the granularity of planning that it currently manages. HIE implements tons of its own task planning logic using (in-memory) Shake as an execution framework. Even if Shake isn't used by GHC itself, as much as possible of that planning logic (and indeed likewise for any domain-layer logic) should be deduplicated with GHC wherever possible. Planning vs execution separation facilitates this.

Finally, one additional recommendation is to document the API of the Application Layer and to write test programs using it. This will unvariably make flaws obvious and will hopefully be an incentive to fix them. That's how one of us got into doing this work

in the first place, so we can testify that this method was successful at least once.

4.2.4 Presentation Layer / User Interface Layer

The Presentation Layer is the interface between an end-user and the rest of the Application Layer. Therefore, GHC has several components in its presentation layer, including: GHC the command-line program, GHCi, Haddock, HLS, HLint, and any other ghc-lib consuming program. These components most often don't belong to the same codebase but they are still conceptually in the same layer.

Most of these UI components have to find workarounds to exploit the Application Layer that was designed for the GHC program first, and extended a little for GHCi. A good example is that all these components have to deal with `DynFlags` values that represent command-line flags for the GHC program as we have seen in Section 4.2.3.

For example, if for the GHC program it was decided that a verbosity level —stored in the `DynFlags`—greater than 2 would enable the display of A and B, there is no way for another client to display only A, except maybe by changing the verbosity level in the `DynFlags` at some specific points in the pipeline (e.g. if the display of A and B occurs in different phases). Our recommendation once again to fix this is to implement proper layering and componentization in the Application Layer, to avoid the need for these kinds of workarounds.

A welcome componentization in GHC's UI layer would be to split GHC-the-program from GHCi. Their needs are very different and it would make for two "internal" clients more representative of what other external clients may require from the other layers.

We now present two examples of refactoring of GHC's UI layer that would accomodate other clients' needs: `OPTIONS_GHC` pragma handling and error messages.

`OPTIONS_GHC` pragma This pragma can be used to pass command-line flags for the GHC program that are only applied to the module containing the pragma. As we have seen in Section 3, this mechanism is fragile and it's easy to find flags that are not compatible with the global ones passed on the command-line, even more so in GHCi.

In the current implementation a single `DynFlags` is obtained by merging the global `DynFlags` with local `DynFlags` from parsing `OPTIONS_GHC`. Note that the local `DynFlags` are a *by-product* of the pre-processing phase of each module, and are threaded through the rest of the pipeline alongside the code itself.

Conforming to our recommendation above, we propose not to do this. Instead, components of the UI Layer such as the GHC program and GHCi would be responsible for interpreting pragmas—that are returned by the preprocessing phase and that they support—in order to setup the rest of the build plan for each module.

As such, a concrete first step is to provide an abstract syntax and a parser for GHC's command-line interface (CLI) parameters.[7] After "downsweep" and after parsing of the `OPTIONS_GHC` pragmas, each module can be paired with some CLI syntax value. Finally, we need two CLI syntax interpreters—one for GHC and another for GHCi—that check that these module specific CLI parameters make sense and can be combined with the other ones (global and/or other module specific ones). Similarly in GHCi to support dynamically modified CLI parameters: parse, validate with the current state, act.

7: #20730 — Refactor command-line options handling (DynFlags)

Generalizing this approach to other pragmas (`OPTIONS_XYZ`), other programs could support GHC's pragmas (reusing GHC's CLI parser and abstract syntax), their own pragmas, or no pragma at all.

Currently GHC's header parser—used by "downsweep"—treats `LANGUAGE xyz` pragmas as syntactic sugar for `OPTIONS_GHC "-Xxyz"` pragmas. As most clients will want to deal with these pragmas, they should probably be returned independently as proper ADT values instead. These values could in turn be converted into the "-X..." stringy form, but only by clients that require it.

Error messages Until recently, warning and error *messages* were produced below the Presentation Layer. This design meant that the Presentation Layer couldn't decide what message to show for a given error/message, except by matching on string messages themselves (another workaround). A typical example is that the Presentation Layer can't easily adapt the language of a message, which is a useful feature for responding to the user locale.

We kicked-off the work that consists in making aformentioned layers return warnings, errors, and hints as algebraic datatype values (a5aaceec). This work was then continued by Alfredo Di Napoli[8].

8: Status is tracked on GHC's wiki

Note that it is still possible to convert these values into default string messages. Separation of concerns—here reporting an error to client code *vs* reporting an error to a human user—once again adds possibilities without impeding existing use cases.

4.3 Supple Design

To avoid paraphrasing, here is the description of *Supple Design* extracted from [3] (pp. 243–245):

> "As a program evolves, developers will rearrange and rewrite every part. They will integrate the domain objects into the application and with new domain objects. Even years later, maintenance programmers will be changing and extending the code. People have to work with this stuff. But will they want to?
>
> When software with complex behavior lacks a good design, it becomes hard to refactor or combine elements. Duplication starts to appear as soon as a developer isn't confident of predicting the full implications of a computation. Duplication is forced when design elements are monolithic, so that the parts cannot be recombined.
>
> [...] When software doesn't have a clean design, developers dread even looking at the existing mess, much less making a change that could aggravate the tangle or break something through an unforeseen dependency. In any but the smallest systems, this fragility places a ceiling on the richness of behavior it is feasible to build. It stops refactoring and iterative refinement.
>
> To have a project accelerate as development proceeds—rather than get weighed down by its own legacy—demands a design that is a pleasure to work with, inviting to change. A supple design.
>
> [...] When complexity is holding back progress, honing the most crucial, intricate parts to a supple design makes the difference between getting sucked down into legacy maintenance and punching through the complexity ceiling."

We claim that GHC lacks a supple design. As an evidence of this, many interesting developments require the use of GHC forks, which are the paramount of code duplication:

- Cross-compilers: Eta (JVM), GHCJS (JavaScript), Asterius (WebAssembly)
- Alternative backends: Intel Labs Haskell Research Compiler [10], External STG interpreter [11]

The lack of modularity is presumably one of the main reasons why these forks exist. It matches our direct experience with GHCJS, which requires changes to unabstracted parts of GHC (e.g. linker handling), which are—sadly—easier to implement by slightly modifying duplicated GHC code.

The lack of modularity isn't the only reason why working with GHC doesn't provide a supple design experience.

- Building GHC takes a very long time: bootstrapping needs to build GHC twice; by default other packages such as Cabal and Haddock are built too

- Constraints on boot libraries: the set of libraries that can be used into GHC's codebase is restricted.
- GHC's build system is complex, slow, and poorly documented. It isn't well-integrated with usual Haskell build tools (stack, cabal-install, Nix), requires specific learning, and has its own set of bugs. It is difficult to modify to adapt to new needs.
- GHC's CI is flaky: CI for merge requests take a long time and may fail arbitrarily on some platforms

As a consequence, it is often much more productive and enjoyable to work outside of GHC. Some of the forks above are only used to make the extraction of intermediate forms (external Core, external STG) easier. All the interesting work then happens in a separate Haskell project that doesn't face GHC's issues.

GHC may not be the best vehicle for research that it used to be. It is difficult, however, to quantify the amount of research that could have relied on GHC but that didn't because of its issues. Taking for example the Unison language [12] whose main contribution is that functions are content-addressed (hash of the RHS), we believe that Haskell could have been used as a source language to experiment with this idea, but we also believe that it would be utterly difficult to reuse GHC's components to achieve this currently.

We hope that GHC's modularity will be the first step towards a more supple design. Improvement of GHC's build system and build process is out of the scope of this paper. However interested readers can read #20748, #17191, #19209, and issues marked with the Hadrian label.

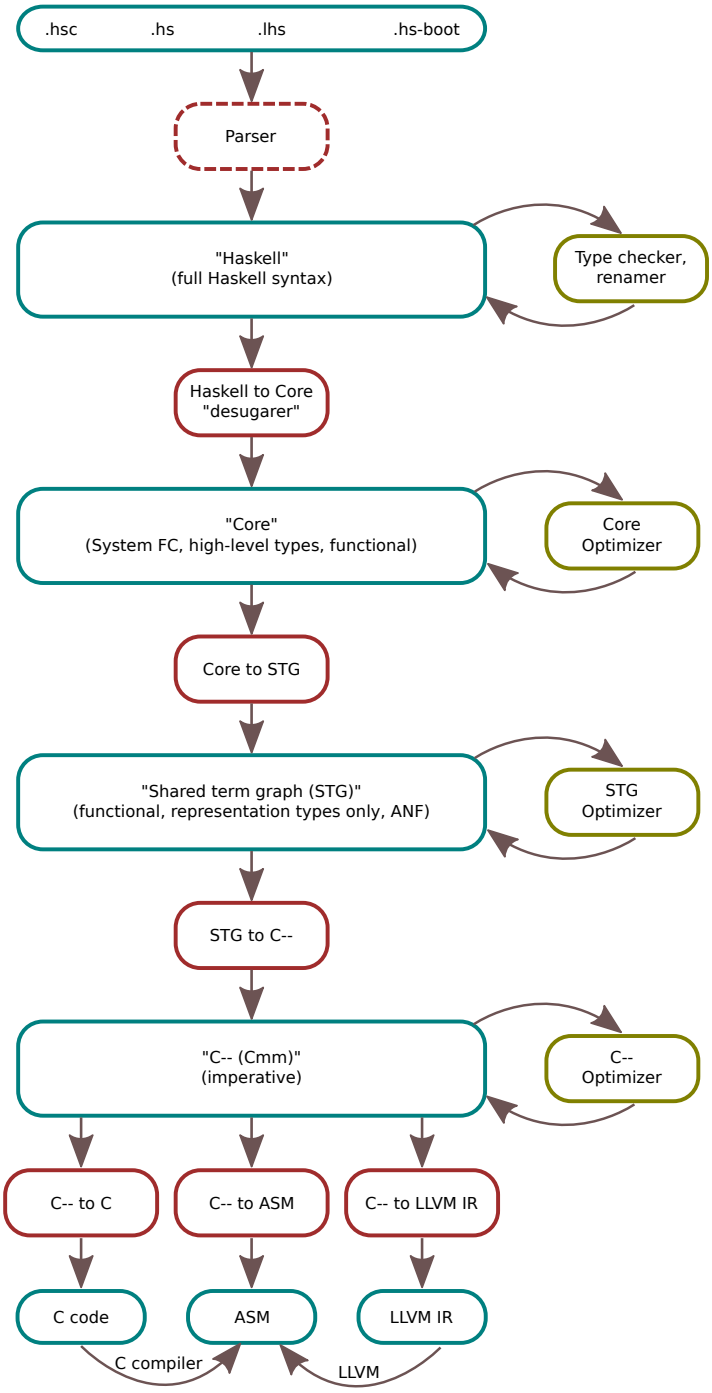


Figure 1: GHC's compilation pipeline.
Blue boxes: code representations ("syntaxes").
Red boxes: compilers from one representation to another.
Green boxes: code transformers/checkers that work on a single representation.

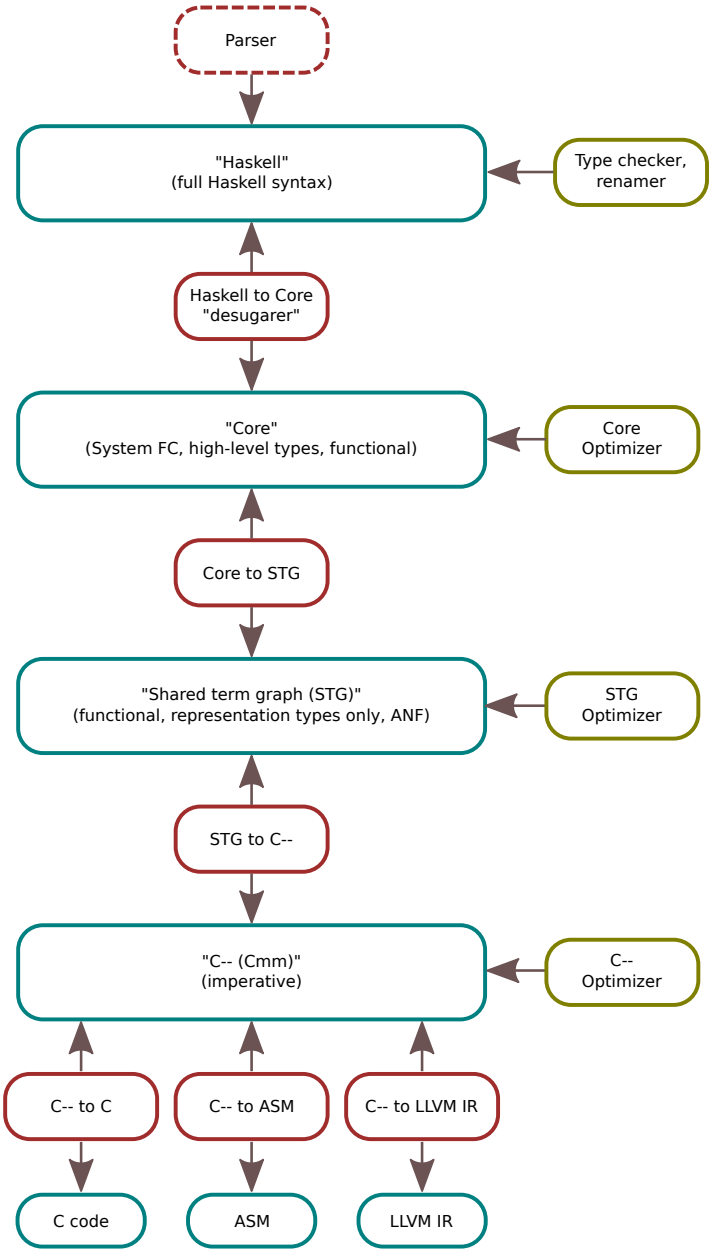


Figure 2: Dependencies between components composing GHC's compilation pipeline.

5 Method

Having discussed what the problems are, and what we hoped to do about them, it serves to discuss about exactly *how* we envision that work will actually happen. The main constraint to consider was that these challenges were always too difficult to be addressed in a single massive patch. Thus, most of our methodological decisions relate in some fashion to *time*. We need an approach that allows us to work little by little, not relying on other GHC developers to closely follow our efforts.

Layering and componentization are at the core of our modularity effort (c.f. Section 4.2). They can be achieved in three steps:

1. Introducing a module hierarchy to represent layers and components (Section 5.1)
2. Removing *accidental* coupling between components (Section 5.2)
3. Removing *undesirable* coupling between components (Section 5.3)

We also need good reasons to believe these improvements will be to some extent self-sustaining, so our work won't immediately bit-rot once the initial cleanup is finished (Section 5.4).

5.1 Introducing a module hierarchy

Task #13009 — Hierarchical Module Structure for GHC
Status Done

"Modules give people two views of the model: They can look at detail within a module without being overwhelmed by the whole, or they can look at relationships between modules in views that exclude interior detail.
The modules in the domain layer should emerge as meaningful part of the model, telling the story of the domain on a larger scale." [3] p. 109

Hierarchical modules have been supported in GHC Haskell since 2001 (6dad6315) and are a useful domain modeling tool, especially for complex Haskell projects. See for example the module hierarchy of the Agda compiler.

However, probably by inertia, almost all modules of the GHC library were top-level modules. Prefixes were used in some cases to distinguish modules belonging to different components of the model, but it wasn't done consistently.

As such it was almost impossible to enforce layering and componentization at the module level because we couldn't even state a rule such as "modules of the Domain Layer mustn't import modules of the UI or Application Layers" because we couldn't identify which layer a module belonged to.

Our first work was to introduce a module hierarchy to make the layering and componentization apparent in the module structure of GHC. This module refactoring started in GHC 8.10 and culminated in GHC 9.0.

From the 9.0 release, we have made most components on Figure 1 have their own module hierarchy. For example `GHC.Stg.*` modules

Component	Nickname	Module	Directory
HsToCore	Desugarer	Desugar	deSugar
CoreToStg	?	CoreToStg	stgSyn
StgToCmm	codegen	StgCmm	codeGen
CmmToAsm	native codegen	AsmCodeGen	nativeGen
CmmToC	ViaC	PprC	cmm
StgToByteCode	?	ByteCodeGen	ghci

Table 3: Terminology for compilation phases in GHC 8.8

implement the STG syntax and its related code; modules in `GHC.StgToCmm.*` implement the STG to C-- pass; etc. Similarly, modules composing the Application Layer have been confined in `GHC.Driver.*` hierarchy.

This hierarchical approach makes clear which modules belong to each component/layer. It was also an opportunity to fix up the terminology used to refer to some components (c.f. ubiquitous language, Section 4.1). As an example, Table 3 shows the terminology used in GHC 8.8 for compiler phases before these changes. In comparison the new terminology ("Component" column) is consistent and follows the principle of least surprise.

Fixing Layering Violations After we introduced a module hierarchy the codebase was still left with a lot of layering violations: modules importing other modules from an upper layer or from a sibling component that they should be independent of.

We fixed the easiest layering violations that only required splitting some modules and/or moving statements from one module to another. Some of the remaining ones are described in Section 3.2 and 3.3 (`DynFlags` and `HscEnv` leaking from UI and Application Layers into the Domain Layer) and were more difficult to fix. Some of them are still not fixed at the time of writing!

Most of these layering violations have not been introduced by design but accidentally. We now explain the approach we follow to remove them.

"It is a truism that there should be low coupling between modules and high cohesion within them. [..] It isn't just code being divided into modules, but concepts. There is a limit to how many things a person can think about at once (hence low coupling). Incoherent fragments of ideas are as hard to understand as an undifferentiated soup of ideas (hence high cohesion)." [3] p. 109

5.2 Removing accidental coupling

A lot of accidental coupling and layering violations occur because of the use of a shared data structure between unrelated components of the compiler. In Section 3 we presented several of these datatypes: `HscEnv`, `DynFlags`, `Hooks`, etc. A method to avoid this coupling is to use component-wise configuration datatypes.

5.2.1 Component-wise configuration

Task #17957 — Avoid direct access to compiler state
Status In Progress

```haskell
data CmmConfig = CmmConfig
  { cmmProfile             :: !Profile -- ^ Target Profile
  , cmmOptControlFlow      :: !Bool    -- ^ ...
  , cmmDoLinting           :: !Bool
  , cmmOptElimCommonBlks   :: !Bool
  , cmmOptSink             :: !Bool
  , cmmGenStackUnwindInstr :: !Bool
  , cmmExternalDynamicRefs :: !Bool
  , cmmDoCmmSwitchPlans    :: !Bool
  , cmmSplitProcPoints     :: !Bool
  }
```

Listing 10: Cmm configuration as part of DynFlags refactoring. Note that we have removed field comments for formatting. See !7199.

The basic idea of component-wise configuration is that we should treat each component (or phase) of the model as if it was an external program: its interface must be well-defined with a component-specific set of options (configuration) and no access to some global shared mutable state. The handshake between some upper level configuration (e.g. `DynFlags`) and the component-specific configuration may now happen in some upper layer: the component itself is only aware of its own configuration.

No rocket science here: We don't expect C compilers to support being passed all of GHC's options and to make sense of them to know how to compile C files produced by GHC with its C backend. Similarly we shouldn't expect its "internal compilers" (e.g. `CmmToAsm`) to support this either.

As an example, let's consider the `Cmm` phase. In the usual pipeline it takes its input from the `StgToCmm` phase and its output feeds one of the `CmmToAsm`, `CmmToLlvm`, `CmmToC`, or the envisioned `CmmToWasm` phase. But to be truly reusable, the `Cmm` phase mustn't know anything about the provenance and the expected usage of the `Cmm` it manipulates!

As such we've introduced a specific configuration datatype for the `Cmm` phase (Listing 10). To keep the existing behavior we've also introduced a `Cmm` configuration initialization function (Listing 11) in the Application Layer (`GHC.Driver.Config.Cmm`).

We can see that this new function sets some options depending on the selected downstream phase (`backend dflags`), some user specified options (`gopt Opt_*`), and some global GHC settings (target platform). These information were directly fetched by the `Cmm` component before, making `DynFlags` the only interface to configure it (cf Section 3.2).

To recap, our process is (1) to isolate components by creating a configuration record for each of them and (2) to create functions to initialize them for GHC's use case, typically a projection from other `DynFlags` or `HscEnv`. Note that other clients of the components don't have to use these functions and can initialize the configuration records as they see fit.

For components of the Domain Layer, we have picked the convention of putting configuration initialization functions in the Application Layer (`GHC.Driver.Config.*` modules). As such, it makes some layering violations blatant, which is good. For example, at the time of writing we can notice that `GHC.Cmm.Parser` imports

```
initCmmConfig :: DynFlags -> CmmConfig
initCmmConfig dflags = CmmConfig
  { cmmProfile             = targetProfile                dflags
  , cmmOptControlFlow      = gopt Opt_CmmControlFlow      dflags
  , cmmDoLinting           = gopt Opt_DoCmmLinting        dflags
  , cmmOptElimCommonBlks   = gopt Opt_CmmElimCommonBlocks dflags
  , cmmOptSink             = gopt Opt_CmmSink             dflags
  , cmmGenStackUnwindInstr = debugLevel dflags > 0
  , cmmExternalDynamicRefs = gopt Opt_ExternalDynamicRefs dflags
  , cmmDoCmmSwitchPlans    = not . backendSupportsSwitch . backend
                             $ dflags
  , cmmSplitProcPoints     = (backend dflags /= NCG)
                             || not (platformTablesNextToCode ...)
                             || usingInconsistentPicReg
  } where ...
```

Listing 11: Cmm phase configuration init function in the Application Layer as part of the DynFlags refactoring. See !7199.

`GHC.Driver.Config.StgToCmm` which is clearly a layering violation. In comparison, a wrong dependency to a sibling component of the same layer (e.g. `GHC.Cmm.Parser` importing `GHC.StgToCmm.*`) is more difficult to detect without proper tooling.

5.2.2 How to proceed: harder then easier

Writing these refactors feels like *untangling* GHC. And like the real-life counterpart of untangling a knot, progress often starts slow until various milestones are reached and the work gets easier.

At the beginning, it was often very daunting to even decide where to begin to start untangling, with few `DynFlags` parameters being obviously and simply more information than was needed. The biggest obstacle was probably pretty-printing. As discussed in section 3.2.4, `DynFlags` was used for pretty printing. And because the unsafe global `DynFlags` was so clearly bad, it really was better to thread otherwise-"overkill" `DynFlags` than use it. Between actual user-facing diagnostics (errors and warnings) and more internal assertions, pretty-printing was ubiquitous, and so this was the most frequent source of `DynFlags` that looked like they could be removed, but couldn't in fact be so easily.

Thankfully, Sylvain has already slayed that monster of pretty-printing, removing `DynFlags` from the pretty-printing internals altogether in addition to reducing the global state to the three booleans as already mentioned. With that, we are now ready to move to a new state of more rapid progress.

Jeff did some dependency analysis, and #17957 now contains a detailed road map of which components need to be converted to `CompConfig` *in dependency order*. That last bit is crucial, because it means someone can grab a check-boxed task, and work on just that component with reasonable confidence they will not be forced to go on a tangent, unraveling a much longer "thread" of separate

components whose lack of their own `CompConfig`s is blocking the fixing of the original component in question.

With this new road map, we are excited to be able to onboard more volunteers to join us in our efforts, and thus increase progress. We are excited to do that! On that front, we've been eagerly planning a 2022 Summer of Haskell project [‡‡] after a perspective mentee expressed interest [§§] in our work.

5.3 Removing undesirable coupling

Tasks Varied and not led by us, see Section 5.3.1 below
Status In Progress

The third step consists in removing coupling between components that we deem undesirable. Compared to Section 5.2, this coupling isn't present by chance but because a component really uses a feature of another. Still in some cases we want to decouple them to allow for greater modularity and code reuse.

As we mentioned above, most of our components are similar to "singleton objects": there exists only one of each. However in some cases we would like them to behave more like standard objects so that the client code can pass another one as long as it has the appropriate interface. In Haskell, this translates to using type-classes or datatypes representing component interfaces.

As an example, if the Renamer handles the execution of Template Haskell splices, we don't want it to depend on the whole compilation pipeline plus the interpreter. Instead we want the Renamer to take as argument a function whose type is something like `Splice -> TcRn HsSyn`. Clients of the Renamer are responsible for plugging-in an appropriate splice interpreter. The benefit is effective loose coupling: one could easily reuse the Renamer with a totally different splice interpreter.

The same kind of thinking is needed for every component: does it really need to know about that other component, or should it be programmed against an abstract interface.

5.3.1 Related GHC projects

We remain focused on the initial steps — pushing through the "grunt work" of decoupling where the path ahead is most clear. There is a lot inertia to be overcome, along with pessimism that GHC or "production tools" in general could ever be implemented in an elegant way, and so we think it is better for *us* to focus on

[‡‡] https://github.com/haskell-org/summer-of-haskell/issues/158
[§§] https://github.com/haskell-org/summer-of-haskell/issues/143#issuecomment-1089402865

initial milestones and getting them done rather than planning far out past any frontier of certainty.

Nonetheless, there are other ongoing GHC projects that could indeed be classified as "removing undesirable coupling". We're very happy they exist, and we look forward to our efforts "merging" with theirs down the road.

Structured errors As mentioned in Section 4.2.4, structured are a laudable and classic attempt to remove presentation layer concerns from the domain layer where they don't belong. Recently, the Haskell Foundation has taken on an official capacity to steer this work, largely due to the needs of HLS, which as a separate application has its own presentation layer, and thus wants to present errors differently. We fully support this project and are very happy the Haskell Foundation has agreed to push it forward.

Haskell Foundation Tech Proposal 24

The diagnostics[1] users most care about are in the early pipeline stages, especially the renamer and type checker. Per our dependency analysis, these will be some of the last components to be purged of `DynFlags` and `HscEnv`. That means the structured diagnostics work will be converting code that could easy use unstructured diagnostics again if code reviewers didn't care.

1: a word for warnings and errors, both are being so-converted

However, once our modularization work reaches those components, it could well be that the structured diagnostics are the *only* sort that are possible! This would especially true if we factored out the entire `SDoc` / pretty-printing infrastructure into a separately library to be only used by text-based presentation layers. This would preclude code reviewers from having to be vigilant as the path of least resistance[2] for contributors now strongly favors structuring any new diagnostic that is needed.

2: a topic we will return to in Section 5.4

Trees That Grow Trees that grow aims to make the abstract syntax / intermediate representation data types more faithful to the compiler pipeline stages that operate on them. Replacing a single AST that struggles to accommodate multiple pipeline stages with multiple variants is very analogous to replacing `DynFlags` with component-wise `CompConfig` records, and thus we support it for all the same reasons.

The GHC Wiki has a page on Trees That Grow, but it needs to be brought up to date. The "TTG" label in the GHC Issue tracker does not offer a good overview, but at least has the advantage of showing information that is up to date.

Trees That Grow initially has initially been more about representing the invariants of each compiler phase, and thus avoiding partial code or invalid states. These are benefits our modularity work also strives to offer, but nonetheless is a distinct goal. After that, however, what remains is #19932 — Reduce AST & parser dependencies , which references the `CountDepsParser` test we mentioned in the introduction (Section 2) along with an analogous `CountDepsAst`. This, on the other hand, is squarely modularization work. The goal here is to allow factoring out key data structures

like ASTs (with any code operating them being purely optional to also factor out), and that is the purest example of making the domain layer reusable.

As with structured errors, once our efforts reach those key we should benefits in access of each project alone. With both the core data pipeline stages operate on, and configuration, services, and other "auxiliary" inputs all made modular, these core pipeline stage components would be well on their way to becoming just as nicely decoupled from their dependency as their inputs are.

Driver features and consolidation Matthew Pickering has initiated a bunch of work on overhauling the driver. We characterize this work as good initial steps in the process of separating planning and execution and also making the plans more fine-grained, two goals discussed in section 4.2.3. We discuss this work, and some antecedents, in grater detail in the appendix section 8.

5.4 Maintenance

It would be a real shame if, after all our hard work, GHC immediately started regressing and becoming more monolithic again. We are not so concerned about this, because the Haskell ecosystem writ large demonstrates that modularity is "sustainable", but it is still good to discuss the possible reasons why that is and make sure they also apply to our planned end state.

Make good outcomes easy Developers are humans, and humans are apt to take the "path of least resistance", all things equal. The heart of any approach to be a strategy of making modular and otherwise good code the natural and easy way to do things as much as possible, conserving the efforts of those authoring and reviewing changes alike. All the facets below of how we propose modularity will sustain itself are oriented around basic fact.

5.4.1 Better continuous integration

GHC's build system and continuous integration (CI) have long been a rather infamous, slowing down development by either being hard to understand, extremely fragile, or both. This frustrates developers and exhausts their patience. In particular, any patience for selfless morsels of effort to improve GHC in passing will be sucked away as getting anything merged at all becomes so much more arduous.

It's best to demonstrate this with an example, and one that implicates one of *us* lest anyone think we are trying to castigate the behavior of others obliquely. In the appendix section 8 we will

mention how in 2113a1d6 John didn't rename `ModuleGraph` when he ought to. In fact, the MR previously contained the rename but then it was removed. This is because due to various testing fiddly bits the MR took forever to get totally correct, and thus had to be rebased many, many times. By keeping the testing, John would have not only made much more work for himself, but also more work for various coworkers who took turns being assigned to the queue of open PRs to get them shepherded along. The extra churn just wasn't worth the rename.

This illustrates a funny dual hazard in the vein of the "Scylla and Charybdis" myth. On one hand, if CI is too lax, making large changes is off-putting because the risk of accidental mistakes slipping by, and, worse, the pain of encountering more such things if one tries to bisect the version control history later. On the other hand, if CI is slow, or stringent but with many false positives, making large changes is *again* off-putting because the risk of "tripping alarms" for silly reasons, and the relatively slow debug cycle of fixing them.

Both these excesses strongly steer developers away from "custodial" work. Given the general pain of development, the path of least resistance becomes strictly "touch as little code as possible", rather than a balance between keeping changes minimal but keeping the design simple.[3] Conversely, once CI is improved, we hope developers will welcome the ability to do more "custodial" work, or at least take more care to not "regress" in overall cleanliness. We hope this will lead to a GHC to a higher standard of quality. Furthermore, It is both easier and more rewarding to keep clean good clean than prevent messy code from merely getting worse. We hope that higher standard of quality will come to be be internalized as not something exceptional, but merely raised standards of the level of "good working order" GHC ought to be kept at without ongoing additional effort. Naturally, we expect such a higher standard to include preventing our work from bit-rotting.

3: While we and [3] especially emphasize tech debt *over time* making code bases hard to understand, there is still some cost imposed on the individual developer making a destabilizing change when they try to debug their work. When the accumulated costs dwarf the cost of another messy change, projects are in poor shape. When the initial state is cleanliness and beauty, this self-imposed cost should be more apparent.

The Haskell Foundation has recently committed to supporting GHC get better CI, and we are strongly supportive of this effort and excited about the benefits of reversing this long-time trend of a difficult developer experience.

Haskell Foundation Tech Proposal 18

5.4.2 Design that maintains itself

Communicating intent via "normal forms" So much of organizing large collaborative projects involves coordinating the design between many participants. GHC has many notes today, which are an excellent resource for readers of the code, placed or referenced right near the code which inspires questions. But they are better at explaining awkward or confusing things than laying forth how code *ought* to be.

Once code is modularized so components are only passed the resourcing and configuration we need, those components begin to take on a normal form of sorts. There is no more parameters to remove, or results to return. The code might also be slightly more general than its extant call sites need, but this is OK. We can think of this as casting off "artificial" limitations, rather than overgeneralizing past natural limitations.

We posit code like this is what most Haskellers are used to and what they like to see. For example `map` could be called "overkill" because it will never be called with arguments more than a small portion of the function space. But it is clearly more natural to write `map` than try to, e.g. defunctionalize for every function argument that happens to exist today.

Once a bit of code is reaches those ideals, it is closer to being "finished" and perhaps more stable. If new feature work creates new demands on the implementation, components might even already support those futures via the aforementioned excess generality.

We want it to be easier to take advantage of generality and preserve modularity than regress and introduce more couplings. Just as we've shown[4] with the evolution of pretty-printing in GHC in particular, halfway-to-omnipresent `DynFlags` and `HscEnv` are very tempting to start using in more places until they are back to being omnipresent once again. Conversely, a bunch of `CompConfig` records which are not used more than once do not create this perverse incentive.

4: our anchoring story throughout the first major section of this paper, section 3.

Overall, library code has a way of just "falling into place" via the separation concerns whereas executable code often evolves meandering ways, never quite settling down. We view our work as completing the job of making GHC a "real library", and not just a glorified executable with the "Main" module lopped off. We thus think this project will unlock those benefits.

Composition over configuration This slogan's origins are unclear but is as useful as catchy, in order to view the domain-driven design lessons from a perspective. Suppose one *not* following domain-driven design, but instead letting new UI/boundary requirements dive development. Suppose also that those boundary requirements are conditional, for example, like language extensions. It is likely that those configuration parameter will end up "worming" their way into domain concepts, obscuring their meaning in addition to entangling layer.

These two blog posts `https://clojurefun.wordpress.com/2012/08/17/composition-over-convention/` `https://johno.com/composition-over-configuration` seem to help create.

Conversely, imagine once the domain concepts have freed from the UI layer, and polished their own right. Instead of threading configuration parameters all the way down, they should increasingly instead just reflect how underlying less-configurable (or even configuration-free) domain objects are composed together. Indeed,

our trying to separate "planning vs execution" in the driver as discussed earlier in section 4.2.3 is but an example of this concept.

This is the key of composition over configuration — small, modular domain concepts that can be composed in many different ways can achieve as much flexibility of purpose as configurable large monoliths.

The existing literature doesn't mention this as much, but this also directly relates to *parametricity* and *free theorems*[13]. Large components with complex configuration have a huge space of inputs that makes reasoning about them very hard. Small components that prefer more abstract inputs (higher order functions, type variables) conversely can be well understood quite easily.

Earlier, we discussed the example of `map` versus a list-processing function that took an defunctionalized input. It is very easy to see that `map` is superior, and one shouldn't replace it with the second version that be aware of all the places or ways it applied, non-compositionally. The challenge is rather *starting* with the second version, and then trying imagine the `map` it wants to be. It is very hard to do this all at once, especially as one must remain vigilant against the temptation to have too much fun and design a bad abstraction. But by chipping away at non-modularity and waiting for "boring" abstractions to reveal themselves as obvious in hindsight, one can do this more safely.

One can view our efforts to make per-component configurations as just a first preparatory step in this longer process, which is merely getting rid of entirely unused parameters with per-component configuration records. With the "extraneous noise" cleared away, one can then try to further refine domain-layer concepts to reduce the remaining parameters. Library like bound show the ideal, where language implementation domain concepts are whittled down into reusable abstraction that are extremely composable with no need for any configuration. The challenge is to do the same with the components of GHC.

A GHC hollowed out into just a thin conglomeration of industrial-strength reusable libraries would be the ultimate testament to Haskell's philosophy, and a huge benefit to industrial and research applications alike.

6 Conclusion

With this paper we have tried to convey that some domain-driven design concepts are applicable and beneficial to functional languages, even strongly typed ones like Haskell. Types don't replace proper design or GHC would have been modular to begin with.

We expect this modularization effort to be beneficial to Haskell

tools developers using GHC. We also hope it will permit research to be conducted more easily by allowing effort concentration on the modification of a single component and reuse of all the other ones. Examples includes:

- new or finally well integrated backends: JavaScript, Wasm, JVM, Grin...
- different drivers: implement a Unison like driver for Haskell, explore automatically profiled guided optimization of Core...
- new or improved frontends: better debugging tools relating intermediate representations and source language (replacing the use of dump files)[1]...

We believe this experience report about GHC can also be useful for other projects so that they don't reproduce some design mistakes that were made. It's also an invitation to consider applying some domain-driven design concepts if your project also suffers from flaws similar to those exposed in this paper.

1: That's what one of us was working on five years ago, work that was put on hold to fix the GHC API first (demo link)

7 Appendix: Related work

We have strived to put facts before opinions regarding the specifics of GHC. Still, it can be useful to note that the broader themes and narratives of our analysis *do* have much precedent, both with other long-lived software in general, and other compilers in particular. Witnessing other projects struggle with the same challenges can help sharpen our diagnosis of what is wrong here with GHC. Witnessing other projects surmount those challenges can give us hope that we can do the same.

7.1 Related GHC projects

We are happy to say making GHC modular should complement other GHC projects currently in progress. These are discussed in the relevant sections, so we just include a glossary of references to those sections below.

Structured errors Discussed in section 5.3.1.
Trees That Grow Discussed in section 5.3.1.
Better Continuous Integration Discussed in section 5.4.1.
Improving the driver Alluded to in section 5.3.1, and then discussed more fully in section 8.

7.2 IDEs *vs* batch compilation

The "Language Server Protocol" that HLS uses to interface with many different editors is just 6 years old, but has taken our world of increasingly many languages and editors by storm. It has rightly been called a revolution [14], as good IDE support went from

being deemed a luxury to being expected by users. This has caught many traditional batch compilers off guard—especially those which didn't previously do something like GHCi. After all, for all the messiness of implementation of GHCi that we've described, at least its existence did mean that we were already aware that batch compiling wasn't the only use-case for our language implementations out there.

7.3 Rust

Perhaps the best story for us is the experience of Rust. Because Rustc was not written with non-batch usage in mind, the first prominent language server for Rust was RLS. RLS did not attempt to do anything fancy reusing the compiler code base, and instead relied on a combination of shelling out to Rustc with JSON standard streams, and using a heuristic implementation of code completion called "Racer". Later, however, rust-analyzer was created to do things the "right" way, always leveraging rustc as much as possible. Rust-analyzer, however couldn't come into prominence until rustc was sufficient re-architected to make this approach viable. And, inverting the usual relationship, sometimes *it* uses a heuristic approach where rustc is not yet well architected enough, whereas RLS can fall back on the correct but slow batch mode JSON output.

For a few years the two language servers competed, but finally in 2020 a Rust RFC [15] (analogous to our GHC proposals but broader in scope) was written to anoint rust-analyzer RLS's successor and the sole official language server. This RFC, and related documents it links, are an absolute gold mine of experience comparable to ours. It specifically mentions the both the need and difficulty of refactoring the older compiler, the inherent tension between the batch and IDE use-cases (especially with more advanced optimizations) and "library-ification" as the only way to support both use-cases without compromise.

7.4 HLS perspective

We've mentioned HLS multiple times, but what do the HLS authors actually have to say? In [16], prior to the GHCIDE–HIE merge creating today today's HLS, Neil Mitchell does have a few slides making note of annoyances caused by GHC. They are part of what we want to fix. The use of Shake to handle data dependencies and caching corresponds to the Infrastructure Layer in [3] p. 70's layer concept breakdown. In fact, the generalization of Shake to also work in memory for the IDE's sake could be conceived as generalizing Shake's *own* Infrastructure Layer.

7.5 Greenfield experiments

Frustrated with the difficulties in reforming existing legacy language implementations, some new projects have emphasized implementation architecture as priority / point of differentiation rather than just focusing on language design alone.

A first example is the "Sixten" compiler. It has a document describing its architecture [17], which nicely lays out the batch vs IDE problem, and a Domain Layer–Infrastructure Layer division of labor with a library called Rock. Rock is inspired in part by Shake and plays the same role as Shake in HIE-Core/GHCIDE/HLS.

A second example is "Unison" [12]. Unison aims to innovate on a number of fronts, but these include totally rethinking the Infrastructure Layer to dispense with the usual assumptions about source code being stored as text, in mutable files, etc. To handle richer and finer-grained infrastructure concerns while also tackling more domain-specific ones like improved incrementality, Unison must keep the infrastructure and Domain layers at least somewhat separate to reduce complexity.

7.6 Misc

GHC's Architecture One that wonders what other projects share GHC's challenges might also wonder when has GHC's architecture has been discussed before. The classic example, merely cited offhand in the main body of this paper, is the two Simons' chapter on GHC contributed to [9].

That document is broader in scope, but the sections "How to Keep On Refactoring", and "Crime Doesn't Pay" in particular relate strongly to what we are writing about here. Based on them, one can argue that even though we are critiquing the current state of GHC, we are also carrying on some older traditions of cleaning it up.

Tech Debt in General Edward Z. Yang, the onetime maintainer of Cabal and implementer of Backpack, has a humorous blog post [18] were he advocates stepping outside the confines of Conway's law to fix upstream problems rather than endlessly hack around them. We do relate, as the three of us authors didn't first approach GHC with our eyes on this work, but rather felt compelled to do so when the annoyances of working with the current interface become too much to ignore.

This blog post has sat in the back of at least one of our minds the past 6 year, and informed the ideas that have ended up in section 5.

8 Appendix: Case study of a consolidated feature in the Application Layer

As previously alluded to in 3.7, the driver is somewhat split and includes a newer "batch mode", used to implement `--make` and `--interactive`, and an older "one-shot mode". The tension between those two has long ben kept in an awkward compromised state, but we are happy to report this situation has dramatically improved in recent years relatively *independently* from our efforts.

Presaging this work is a bug fix (summarized in listing 12) John helped work on. The fix involved removing a restriction imposed on the driver to fix a bug in backpack. First of all, we do admit this is a blatant "ubiquitous language" violation, as the graph data type is still called `ModuleGraph` even though it is no longer just about modules.[1] That said, we don't regret the change itself.

The name "module graph" implies this is Domain Layer functionality, but, as we have asserted, the driver forms GHC's Application Layer. The key is an understanding that this was always a misnomer — while the module graph did previously just contain modules, this was never the *point*. Rather, the graph's purpose is to track the tasks the "batch mode" compilation manager is responsible for doing, whatever those tasks may be, and their dependencies.

That the work items all previously corresponded to modules merely indicated that we had not yet exposed the full range of tasks GHC performs to the compilation manager, still executing some imperatively as part of other tasks.

The name "batch mode" is rather confusing, since the traditional "one-shot mode" of GHC is the epitome of the traditional "batch compiler" model. The root confusion is that the word "batch" is being used in two senses. "Batch mode" comes from the original meaning of the world, as that mode exists to handle multiple Haskell modules together *in a batch*. "Batch compiler" presumably comes from the connotation of batch processing systems being long latency and non-interactive in comparison to client-server architectures.

1: Why John might have been so sloppy has been briefly discussed in section 5.4.1.

```
commit 2113a1d600e579bb0f54a0526a03626f105c0365
Date:   Thu Apr 30 11:09:24 2020 -0400

    Put hole instantiation typechecking in the module graph and
    fix driver batch mode backpack edges

    [...]

--- | A ModuleGraph contains all the nodes from the home package (
    only).
--- There will be a node for each source module, plus a node for
    each hi-boot
--- module.
+-- | A '@ModuleGraphNode@' is a node in the '@ModuleGraph@'.
+-- Edges between nodes mark dependencies arising from module
    imports
+-- and dependencies arising from backpack instantiations.
+data ModuleGraphNode
+  -- | Instantiation nodes track the instantiation of other units
+  -- (backpack dependencies) with the holes (signatures) of the
    current package.
+  = InstantiationNode InstantiatedUnit
+  -- | There is a module summary node for each module, signature,
     and boot module being built.
```

Listing 12: Commit broadening the scope of the module graph

```
+  | ModuleNode ExtendedModSummary
+
+-- | A '@ModuleGraph@' contains all the nodes from the home package (only). See
+-- '@ModuleGraphNode@' for information about the nodes.
+--
@@
 --
 -- The graph is not necessarily stored in topologically-sorted order.  Use
 -- 'GHC.topSortModuleGraph' and 'GHC.Data.Graph.Directed.flattenSCC' to achieve this.
 data ModuleGraph = ModuleGraph
-  { mg_mss :: [ModSummary]
+  { mg_mss :: [ModuleGraphNode]
   , mg_non_boot :: ModuleEnv ModSummary
     -- a map of all non-boot ModSummaries keyed by Modules
   , mg_boot :: ModuleSet
```

Matthew Pickering independently came to similar conclusions as us. His series of driver refactors including 25977ab5, 421beb3f, 5f0d2dab, through fd42ab5f, along with f243acf4 by Divam Narula but done in close collaboration with Matthew Pickering, have dramatically increased the progress of breaking up imperative spaghetti code and building more and finer-grained data structures to separate the work instead. This includes adding a third sort of task node to "module graph", and creating other data structures which are slated to eventually merge with it. All together, these are strong initial steps in separating "planning" and "execution" as discussed in section 4.2.3.

Matt's work on this front should end up merging with our work factoring out component-wise configuration records in an elegant way. Currently, planning data structures like the "module graph" sometimes contain `DynFlags`, e.g., as in Listing 1, and other times defer threading it (or `HscEnv`) until the execution time. But neither approach is fully satisfactory. Services are only needed for execution and so should be threaded, but configuration records are quite arguably part of the plan.

```
-- | Data for a module node in a 'ModuleGraph'. Module nodes of the module graph
-- are one of:
--
-- * A regular Haskell source module
-- * A hi-boot source module
--
data ModSummary
   = ModSummary {

        {- ... -}

        ms_hspp_opts    :: DynFlags,
          -- ^ Cached flags from @OPTIONS@, @INCLUDE@ and @LANGUAGE@
```

```
14          -- pragmas in the modules source code
15
16        {- ... -}
17
18    }
```

Let's unpack the planning and execution separation further. Some parts of the overall configuration are inspected when creating the plan, and other parts can be passed off as block boxes (from the perspective of the driver) to the domain layer components below. Those black boxes are precisely the component-specific configuration records! If we store them inside the plan data structures, then the plan execution can only take services, not further configuration as arguments, and then finally have completed a proper planning–executing separation.

Finally we note in passing that HLS's own task dependency structures and GHC's ought to be able to converge over time. The HLS Application Layer is inherently more complex, as it underpins a more complex Presentation Layer, but it should be able to reuse many of the same building blocks.

In section 4.2.1, we discussed some reasons why we thought it an `XYZEnv` combining the services and configuration for one component was ill-advised. The section above provides one more reason. Since the end goal is thus for the configuration to live in the plan, the services to only be passed in when executing the plan, those two sorts of parameters would be coming from different information flows anyways. Combining parameters together is most useful if their arguments are likely to come from the same place, but we see now that that is intentionally not the case in the end goal design.

References

[1] Simon Peyton Jones, ed. *Haskell 1998 Revised Report*. http://www.haskell.org/definition/. 2003 (cited on page 2).

[2] Simon Marlow, ed. *Haskell 2010 Report*. http://www.haskell.org/definition/haskell2010.pdf (cited on page 2).

[3] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 2003 (cited on pages 4, 5, 28–31, 38, 43, 44, 50, 54).

[4] Falcon Momot et al. "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them". In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016, pp. 45–52. DOI: 10.1109/SecDev.2016.019 (cited on page 5).

[5] Alexis King. *Parse, don't validate*. https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/. Nov. 2019 (cited on pages 6, 30).

[6] Mike Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE*. Sept. 1975 (cited on page 10).

[7] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004 (cited on page 17).

[8] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: *Proceedings of the 2002 Haskell Workshop, Pittsburgh*. Oct. 2002, pp. 1–16 (cited on page 26).

[9] Simon Marlow and Simon Peyton Jones. "The Glasgow Haskell Compiler". In: *The Architecture of Open Source Applications, Volume II*. 2011 (cited on pages 27, 55).

[10] Hai Liu et al. "The Intel Labs Haskell Research Compiler". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. Haskell '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 105–116 (cited on page 39).

[11] Csaba Hruska. *External STG Interpreter*. https://www.patreon.com/posts/external-stg-49857800. Apr. 10, 2021 (cited on page 39).

[12] Paul Chiusano et al. *Unison*. http://unisonweb.org. 2015 (cited on pages 40, 55).

[13] Philip Wadler. "Theorems for free!" In: *Functional Programming Languages and Computer Architecture*. ACM Press, 1989, pp. 347–359 (cited on page 52).

[14] Piotr Kaźmierczak. *The LSP Revolution*. https://piotr.is/2020/the-lsp-revolution/. Nov. 19, 2020 (cited on page 53).

[15] Niko Matsakis and Aleksey Kladov. *Transition to rust-analyzer as our official LSP (Language Server Protocol) implementation*. https://github.com/rust-lang/rfcs/blob/master/text/2912-rust-analyzer.md. 2020 (cited on page 54).

[16] Neil Mitchell. *Making a Haskell IDE*. https://ndmitchell.com/downloads/slides-making_a_haskell_ide-07_sep_2019.pdf. Sept. 7, 2019 (cited on page 54).

[17] Olle Fredriksson. *Sixten's query-based compiler architecture*. https://github.com/ollef/sixten/blob/master/docs/QueryCompilerDriver.md. 2019 (cited on page 55).

[18] Edward Z. Yang. *Seize the Means of Production (of APIs)*. http://blog.ezyang.com/2016/09/seize-the-means-of-production-of-apis/. Sept. 2016 (cited on page 55).