UTRECHT UNIVERSITY

THESIS REPORT

DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

Embracing Core & Supervising the Optimisation Pipeline

Empowering Haskell developers with a looking glass into the core2core pipeline

---

Author:
H.A. Peters, B.Sc.
h.a.peters@uu.nl
5917727

Supervisors:
Dr. W. Swierstra
Dr. T. McDonell

DECEMBER 2022

Abstract

GHC – the Haskell compiler – uses a bespoke intermediate language upon which a number of separate optimisation transformations take place. From the compositional style of programming, that makes languages like Haskell so attractive, follows that optimisation is essential as to not produce unreasonably slow binaries. While generally successful, it is needed at times to inspect this intermediate representation throughout the transformations to understand why performance is unexpectedly disappointing or has regressed. This has historically been a task reserved for the more hardened and experienced Haskell developer, and is often done in a primitive manner.

Recent research has explored the ability to include assertions about optimisation that are expected to take place in traditional test suites. After all we generally want our programs to not only be correct, but also terminate in a reasonable amount of time. This is an exciting idea, but it does not address the need to inspect the intermediate representation itself and the skill required to do so.

We believe that Core inspection can be streamlined with an interactive tool that allows users to explore and comprehend such intermediate programs more pleasantly and efficiently. We describe what such a tool may look and how we implemented it. Then we empirically evaluate our tool by reproducing a real world performance regression in the popular `text` library and show how our tool could have been of assistance in that situation. Furthermore, we discuss how we used our tool to discover a performance bug in the fusion system of contemporary GHC itself.

Universiteit
Utrecht

Contents

Chapter 1

Introduction

1.1 GHC, an optimising compiler

Haskell is a high level language designed to write highly composable functions. This naturally encourages programmers to write code that is not particularly fast to evaluate. An extremely common example is the composition of list operations. Consider the function `halves` which divides each element in a list of `Int`s in halve, discarding those that are not a multiple of 2.

```haskell
halves :: [Int] -> [Int]
halves = map (`div` 2) . filter even
```

Both `map` and `filter` are defined as a loop over an input list, producing a new list as output. If code was generated for `halves` in its current form, it would produce two loops as well as allocate an intermediate list. This is unnecessary extra work and incurs more allocation costs because it would be possible to rewrite the function to circumvent this issue:

```haskell
halves_fast :: [Int] -> [Int]
halves_fast [] = []
halves_fast (x:xs) =
  let
    tl = halves_fast xs
  in if even x
     then (x `div` 2):tl
     else tl
```

However, requiring the programmer to do such rewrites manually tragically undermines the benefits of this compositional style of programming. Code simply would be harder to read, write and maintain and likely to be less correct as a result.

Luckily, GHC does address this issue - and many others - with an extensive set of optimisation transformations. This particular program will benefit greatly from the fusion system which specifically deals with removing the intermediate lists. This is a well-established optimisation that is also referred to as deforestation [25]. As a result, compiling with these optimisations enabled will result in a **syntactically** equivalent definition for `halves` and `halves_fast`!

But this poses a question of trust; No compiler is every perfect, so how can we be sure that our code is correctly optimised and will continue to be in the future? Furthermore, if optimisations are missed, how can we diagnose the root of the problem and explain what went wrong?

It should be noted that this problem is not unique to Haskell or even functional languages. Among `C` and `C++` programmers, it is common to refer to services like Godbolt [15] to inspect the generated assembly code for a given function. This is useful both when discussing performance implications of operations happening in the hot path of a program, and to confirm whether certain zero-cost abstractions are actually zero-cost.

1.2 The cascade effect

Optimisation transformations are applied in a certain order, giving rise to the *cascade effect* [21]. This effect refers to the dramatic consequences that the order of transformations can have. We will study an example that showcases a problematic tug-of-war between two optimisations: (1) inlining and (2) rewrite rules.

1.2.1 The inlining transformation

Inlining is arguably the most important optimisation for any functional language. Coincidentally, it is also one of the simplest transformations to implement and comprehend. To reveal why it is such a staple, we must consider it in conjunction with β-reduction. These transformations are fairly simply defined as:

```
-- inlining, knowing that x = a from some binding site
e                  -inline->      e[x := a]

-- beta reduction:
(\x -> e) a        -ß->           e[x := a]
```

We can apply this in a Haskell context to get a more familiar perspective. Consider the boolean negation function `not` and a basic usage thereof:

```
not :: Bool -> Bool
not = \x -> if x then False else True

t = True
f = not t
```

In the case of `f` function we can elect to inline both `not` and `t` in its body. This gives us

```
f = (\x -> if x then False else True) True
```

We can now perform β-reduction by taking the body of the lambda function and substituting `x` by `True`:

```
f = if True then False else True
```

Finally, we eliminate the `if` statement by evaluation. Knowing that it always takes the first branch, we get the final result of:

```
f = True
```

Thanks to this transformation duo, we have reduced an expression to a mere literal, eliminating any runtime cost. Because the inlining transformation so commonly goes hand in hand with β-reduction we will – for brevity's sake – from now on presume that β-reduction, wherever relevant, is also applied when we say that an inlining transformation has taken place.

In a sense we were lucky here that the body of the `not` function reduced so completely. But in reality it can be the case that function definitions are quite large. Inlining will then still eliminate the need for a function call, but it does come at the cost of larger code sizes through code duplication. GHC uses quite a few crude heuristics to help with weighing this trade-off [21]. Interestingly, these heuristics have not undergone much reconsideration until quite recently [13]. Suffice to say that it is not trivial to predict whether a function will be inlined and the performance implications thereof.

Laziness and thunks

Being a lazy language, Haskell can profit from inlining in a secondary way as well. A well-established language feature is the let expression, a syntactically nice way to bind extra definitions. Consider a function that decrypts the password of a user but only if it is the admin.

```haskell
getAdminPassword :: User -> Maybe String
getAdminPassword user =
  let decrypted :: String
      decrypted = decryptSHA1 (password user)

  in case (username user) of
    "admin" -> Just decrypted
    _       -> Nothing
```

Obviously the `decryptSHA1` is going to be an extremely costly function that under no circumstances we wish to be evaluated unless absolutely necessary. Luckily, because Haskell is a lazy language the let assignment does not actually evaluate anything; it only allocates a so-called *thunk*. Such a thunk represents an expression that is yet to be evaluated. Evaluation is in fact deferred to the point where the thunk is actually needed (which is never if the username is not `"admin"`). So, because of this property `getAdminPassword` is actually quite effective at avoiding unnecessary work.

Yet it can still be improved upon. While the let expression might not do any extra work it does require this thunk allocation, which is not free. But because our let bound variable `decrypted` is used exactly once, we can inline it at the only call-site without running the risk of duplicating the expensive operation. This yields the ever so slightly more performant definition:

```haskell
getAdminPassword :: User -> Maybe String
getAdminPassword user = case (username user) of
    "admin" -> Just (decryptSHA1 (password user))
    _       -> Nothing
```

1.2.2 Rewrite rules

We've seen how generic program transformation may improve performance. However, GHC can only use relatively shallow reasoning as to not jeopardize the correctness of transformations or explode compile times. The programmer on the other hand, may have much more in-depth knowledge about the domain of the program and its intended behavior. [22]. Programmers can leverage this knowledge by defining so-called *rewrite rules*. They inform the compiler of substitutions that are not obvious or strictly speaking even correct.

Consider the binary tree datatype `Tree` along with a the higher order `mapTree` functions that facilities transforming the values contained in the `Leafs`. We then use `mapTree` to compose two traversals with an addition in the function `addFive`. Obviously that function is rather contrived as an example of something that someone would write, but the pattern may very well show up during the transformation pipeline.

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node lhs rhs) = Node (mapTree f lhs) (mapTree f rhs)

addFive :: Tree Int -> Tree Int
addFive = mapTree (+1) . mapTree (+4)
```

By now it should be clear why `addFive` is non-optimal; it has a superfluous traversal and allocates an intermediate structure. `mapTree (+5)` is the far superior equivalent. However, for GHC to infer this fact it has to perform a too complicated and generic analysis. But we can add the rewrite rule *mapTree/mapTree* to convince GHC that consecutive applications of `mapTree` are allowed to fuse:

```
{-# Rules
   "mapTree/mapTree"   forall f g.   mapTree f . mapTree g    = mapTree (f. g)   ;
#-}
```

But as it turns out, *mapTree/mapTree* never fired for a reason that is not immediately obvious.

A note on common abstraction

A clever reader might have noticed that `mapTree` is a perfect candidate for an implementation of `fmap` as part of the `Functor` typeclass:

```
instance Functor Tree where
  fmap = mapTree
```

Applying this same reasoning to the rewrite rule, one might feel enticed to write:

```
{-# Rules
   "fmap/fmap"   forall f g.   fmap f . fmap g    = fmap (f. g)   ;
#-}
```

This is not entirely controversial as the soundness of this rule is verifiable under the Functor laws. However, GHC has decided against enforcing these laws nor performing transformations that require these laws to hold. This means that defining custom rules per datatype is required to yield maximum performance. Similar situations arise with the `Applicative` and `Monad` typeclasses.

1.2.3 Tug-of-war

A common manifestation of the cascade effect is the tug-of-war between inlining and the application of rewrite rules. Continuing with the `Tree` example from the previous section, we find that losing this tug-of-war is the exact reason that the *mapTree/mapTree* rule never fired. Because the inlining operation was performed first, the left-hand-side of *mapTree/mapTree* no longer occurs in the program and the rule is rendered non-active. The final optimised function has regrettably converged to the following form:

```
addFive :: Tree Int -> Tree Int
addFive = \tree -> mapTree (+1) (mapTree (+4) tree)
```

The important point is here is not that rewrite rules are inherently flawed (after all, we could easily drum up a secondary rule that does fire here), but that one optimisation may open or close the door to many other optimisations down the road. From this cascade effect follows that the interaction of a Haskell program with the optimiser may be quite unstable and consequently sensitive to small changes. Changes not only in the source but also in the build environment (a minor release of the compiler comes to mind). Thus, we cannot trust that our successfully optimised program will remain optimised in the future. We observe that each program we write may require specific, manual, effort to be made more efficient.

1.2.4 Non-functional requirements: `inspection-testing`

So if we want our programs to not only be correct but also terminate in reasonable amount of time while not consuming an overly large chunk of resources, we have to identify an extra set of contraints. These constraints do not deal with the functionality of the program but rather its compiled form. This collection of additional constraints are examples of *Non-functional* requirements.

To illustrate with a real world example, the very popular `text` library makes the following promise: '*Most of the functions in this module are subject to fusion, meaning that a pipeline of such functions will usually allocate at most one Text value.*' [3]. Like with `addFive` in Section 1.2.3, such promises cannot be checked with traditional tests as they do not concern the functionality of the code.

And as identified by Breitner [3], the aforementioned promise by the `text` library had in fact been broken in version `1.2.3.2`, shown by the following counter example:

```haskell
import qualified Data.Text as T
import qualified Data.Text.Encoding as TE
import Data.ByteString

countChars :: ByteString -> Int
countChars = T.length . T.toUpper . TE.decodeUtf8
```

Although `countChars` uses a value of type `Text` during the computations, it does not need to be actually constructed in the final composite definition. As we learn from the definition:

```haskell
data Text = Text ByteString Int Int
```

`Text` text is merely a *view* into a `ByteString` by virtue of an offset and length parameter. This means that length reducing operations can cleverly avoid the costly task of modifying the underlying `ByteString` and instead just change the offset and length parameters. Now because UTF-16 contains surrogate pairs, the character length of a `Text` value cannot directly be determined from the byte-length its `ByteString` and still requires $O(n)$ time. However, this does not justify constructing a concrete `Text` value as opposed to just using the `ByteString` and the offset and length parameters as separate bindings.

An analogous situation would be a factory with production line workers that pack and unpack their intermediate results between every exchange in the assembly line. While they may conveniently communicate about receiving a 'car door, a bolt, and a nut' unified as their 'input', they never actually mean to suggest that they wish to receive them packaged together. So too with our `Text` values, it is mighty handy to communicate about a package of a `ByteString` and two `Int`s, but we never intend to actually package them at every turn.

But as mentioned, despite the extensive set of rewrite rules that the `text` library has, the ideal compilation result was not achieved. In itself this example formed the main motivation to develop a method to tests these non-functional requirements. The result is the `inspection-testing` package [3]. It provides the machinery necessary to add the following statement directly to the source file, preventing the same regression from occurring in the future.

```haskell
inspect $ 'countChars `hasNoType` ''T.Text
```

Despite preemptively saving us from future regressions, `inspection-testing` does not help to identity and path underlying cause. Consider when the test fails at some point, and you are tasked with finding the root of the problem. In this very example the final optimised definition of `countChar` will have undergone many expanding transformations producing over 100 lines to painfully sift through without little more ergonomics than a string search.

It would be possible to get the output at various intermediate moments to gauge where the offending `Text` should have been erased, resulting in an extra compilation artifact per transformation. Usually this means you have to analyze around 20 different versions of the code. This is a tedious and error-prone process, not to mention requires relatively highly skilled programmers with an in-depth understanding of the GHC and its optimiser.

Moreover, we risk having to repeat this work in the future when any small number of changes could, through the cascade effect, trigger this test as failing again.

1.3 Introducing `hs-sleuth`

We intend to address the tediousness and skill required for exploring interactions of specific pieces of Haskell source code with GHC's optimisation pipeline.

We believe that there is an opportunity to improve the way that Haskell programmers reason about the performance characteristics of their code while simultaneously appealing to a larger audience of less experienced programmers.

This belief stems first from the results of the yearly Haskell survey where in 2021 only 16% of respondents either agreed or strongly agreed with the statement '*I can easily reason about the performance of my Haskell programs*' [7]. We are not the only one seeing this statistic as a call to action. As recently as the current year, Young. J. announced work on a complete Haskell optimisation handbook [27]. A preliminary version of the book already shows that a section on analysing optimisation transformations is planned.

Although, it is our opinion that a guiding resource – while certainly extremely helpful – is not a substitute for better tooling.

Secondly, through conversations with Haskell programmers at Well-Typed – who are also major contributors and maintainers of GHC itself – we learned that it was common practice to create personal tools that assisted them in analysing intermediate results of the optimisation transformations during the fairly frequent task of trying to make performance critical sections of programs more efficient. This proves that there is a demand for such tools and that a unified solution does not yet exist.

1.4 Research Questions

Main Question How can GHC's core2core passes be captured and presented in such a way that users productively gain insight into how their code is transformed?

Sub-Question 1 How does one efficiently identify where small changes occur in two or more captures?

Sub-Question 2 How to make viewing core more manageable using various display options?

Sub-Question 3 How could performance regressions that have occurred in the past in popular Haskell projects, have been resolved faster?

Chapter 2

Background

2.1 A Core Language

GHC recognises three distinct stages of the compilation process [21].

1. **Frontend**, parsing and type checking, enables to give clear errors that reference the unaltered source code.
2. **Middle**, a number of optimisation transformations.
3. **Backend**, generates and compiles to machine code, either via `C--` or `LLVM`.

The middle section is tasked with doing all the optimisation work it can, leaving only those optimisations to the backend it can not otherwise perform. Within the middle section, the work is further split into substages, where each transformation is a separate, composable pass. An obvious benefit to this approach is that each pass can be tested independently. Moreover, because the types are preserved throughout the middle section, it can be verified that each transformation preserves type correctness; Providing some evidence towards the correctness of the transformation.

Regardless, the Haskell source language itself is not a good target for optimisation. The source has to be rich and expressive, requiring an AST datatype with of over 100 constructors. Any transformation over such a datatype has far too many cases to be considered practical to implement; Not to mention the myriad of bug-inducing edge cases. To overcome this issue, the middle section first performs a desugaring pass, translating the source language into a far simpler – but expressively equivalent – intermediate language called *Core*.

Core – which is how we will refer to GHC intermediate representation going forward – is designed to be as simple as possible without being impractical. A testament to that property is the fact that we can fit the entire definition on one page:

```haskell
data Expr
  | Var Id
  | Lit Literal
  | App Expr Expr
  | Lam Bndr Expr
  | Let Bind Expr
  | Case Expr Bndr Type [Alt]
  | Cast Expr CoercionR -- safe coercions are required for GADTs
  | Tick Tickish Expr   -- annotation that can contain some meta data
  | Type Type
  | Coercion Coercion

data Alt = Alt AltCon [Bndr] Expr

data Bind
  = NonRec Bndr Expr
  = Rec [(Bndr, Expr)]

-- the Var type contains information about
-- a variable, like it's name, a unique identifier
-- and analysis results. Binders, Variables, Ids are
-- all the same thing in different contexts
type Bndr = Id
type Id = Var

type Program = [Bind]
```

Code Snippet 2.1: An ever so slightly simplified version of the Core Language

Writing an optimisation transformation essentially of type `Program -> Program` does not now seem as daunting, given the drastically reduced number of cases to consider. Likewise, maintaining and debugging transformations is much less of a strenuous task.

2.2 The core2core transformations

Over its numerous decades of development, the core2core pipeline has been fitted with a myriad of transformations. It would be impractical to discuss all of them here. However, we will discuss in depth the role of the multiple simplifier passes, as well as the worker/wrapper transformation. The first because it gives context to the rewrite rules and the latter because it is a natural bridge to unboxed types; both concepts which will become relevant in discussing the results of this thesis. Lastly, we zoom in on the analysis results stored in the `Var` type.

2.2.1 The simplifier: its parts

The simplifier is quite literally an indispensable part of the transformation pipeline. Although the parts of the pipeline are meant to be separable entities, they heavily rely on the simplifier to deal with the shape the cleanup some of the mess beforehand as well as after. As such, it has earned itself the reputation of being the workhorse of the pipeline [1].

If you were looking for an atom, you have not found it. The simplifier is in itself again a collection of smaller transformations, albeit applied repeatedly and interleaved such that they cannot be easily untangled. Each simplifier subpart is a local transformation, that is, it only looks at the immediate surroundings of the expression. We give a near comprehensive list of each subpart along with an example: [1]

Constant Folding

Evaluate expressions that can be evaluated in compile time. This is a very logical transformation that does not require any further considerations.

```
-- before
3 + 4
-- after
7
```

Inlining

Inlining, replacing calls to functions with the body of that function, is a well known performance enhancing operation in most languages, but especially so in functional ones. The difference is staggering with a around 10-15% performance increase for conventional languages, as opposed to 30-40% for functional languages [21].

Unlike constant folding, it is not an ad-hoc no-brainer. An obvious good case would be to inline a function that is used exactly once. All that this does is remove the veil that hides the function body for further optimisations:

```
-- before
f x = x + 1
f 3
-- after
3 + 1
```

However, if the function is used in multiple locations with different arguments, you risk increasing the size of the program because the function body will be duplicated at each call-site. This is a trade-off that – although often worth it – must be considered on a case to case basis. GHC has a number of heuristics to determine whether inlining should occurs or not [20]. Obviously, this cannot be a perfect solution and one can imagine how a small change in the code can suddenly cause the inliner to reverse its decision; a testament to the non-continuous nature of the compiler with respect to the input program.

Besides inlining function calls, Haskell's let-bindings also form an opportunity for inlining. After all, let bindings are often described as syntax sugar for lambda abstractions, but there is an important difference to be considered. Because let bindings in Haskell are lazily evaluated, it may lead to explosion of work if the let bound variable is used in a lambda expression. For example:

```
-- before, the function 'expensive' is called at most once
let v = expensive 42
    l = \x -> ... v ...
in map l xs

-- after, 'expensive' is called for each element of 'xs'
let l = \x -> ... expensive 42 ...
in map l xs
```

In this case, inlining would be disastrous for performance and GHC will take great care to avoid it.

Case of known constructor

The case destruction is only the way to get to the Weak-Head-Normal-Form (WHNF) of an expression. This means that inside of a case expression we have learned information about the variable under scrutiny and can use it to infer the result of outer case expressions. Consider the following scenario:

```
safe_tail :: [a] -> [a]
safe_tail ls = case ls of
    [] -> []
    x:xs -> tail ls
```

Which inlining will transform into:

```haskell
safe_tail :: [a] -> [a]
safe_tail ls = case ls of
    [] -> []
    x:xs -> case ls of
        [] -> error "tail of empty list"
        (x:xs) -> xs
```

Since we scrutinize `ls` again in the inner case, we actually already know ls is not the empty list, so we can safely replace the inner case with the body of the case expression:

```haskell
safe_tail :: [a] -> [a]
safe_tail ls = case ls of
    []   -> []
    x:xs -> xs
```

The removal of the call `error` during this process is a testament to this function being deserving of the `safe` prefix.

Case of case

There are cases (no pun intended) where the case-of-known-constructor cannot quite do its job, although it is obvious that benefits are yet to be gained. Consider for example what happens when instead of scrutinizing the same variable, the outer case scrutinizes the result of the innner case:

```haskell
-- before (possibly desugared from 'if (not x) then E1 else E2'
-- after also inlining 'not')
case (case x of {True -> False; False -> True}) of
    True -> E1
    False -> E2
```

We might hope to gain something from the information that the inner case has produced by moving the outer case expression to each branch of the inner one:

```haskell
case x of
    True -> case False of {True -> E1; False -> E2}
    False -> case True of {True -> E1; False -> E2}
```

Now we can rely on constant folding to simplify all the way down to:

```haskell
case x of
    True -> E2
    False -> E1
```

Note that we do risk duplicating `E1` and `E2`, which could have been problematic if the expression which contained them did not reduce so completely. A solution to this problem are *join points* [16], which we will not go into here for the sake of brevity.

Rewrite rules

We already discussed in Section 1.2.2 how rewrite rules are a way to express domain specific knowledge to the compiler that it otherwise can not practically infer. Applying given rewrite rules is a task also reserved for the simplifier. It should now be clear that this process can get a little messy since the simplifier is under no obligation to apply the rules, nor its other tasks, in any particular order. We will get into this issue more in the next section where we discuss the simplifier at large.

It should be noted that the use of rewrite rules are very common during the compilation of most any Haskell program. This is because the internal fusion system on list operations are implemented as built in rewrite rules. We discuss this system more in depth in Section 2.3.

2.2.2 The simplifier: its sum

An attentive reader may have noticed that when explaining one part of the simplifier, we often relied on another to do its job. This begs the question: how does the simplifier determine in which order to run its subparts? The answer to that is that it does not. The analogy here is that a compiler is a gun and a program is a target. Every program is very different, and we cannot know in advance how to best hit it. Therefore, we must ensure that the compiler – or this case the simplifier stage – has a lot of bullets in its gun to ensure being able to effectively hit a lot of targets [21]. Running the simplifier once is therefore not satisfactory, and we must run it until it reaches a sort of *fixed point*; At least up until some arbitrary limit since there is no guarantee such a fixed point exists and even if it does that we find it and not get stuck in a loop.

Aforementioned looping behavior can actually occur quite easily due to certain rewrite rules. It is not always the case that the RHS of a rewrite rules objectively better than the right. It might be that the rewrite is benificial because it **may** enable other optimisation to take place afterwards. However, if for some reason that did not turn out to be possible, we may want to apply the reverse rewrite rule later. This is obviously problematic as we can be ping-pong between rewrite rules forever. To overcome that issue one can enable/disable rewrites rules in certain phases of simplifier. To understand this we must first understand when the simplifier is run.

In order, the simplifier is – rather unintuitively – interspersed between other transformations as follows:

1. **Gentle** (disables case-of-case transformations)
2. **Phase 2**
3. **Phase 1**
4. **Phase 0**
5. **Final** (is run multiple times throughout the transformations after phase 0)

By default, rewrite rules as well as inlinings can occur in each of these phases, but the programmer does have the ability to specify deviations from this behavior. For example, in the `text` library, we find in the `Data.Text` module the following snippet:

```haskell
-- This function gives the same answer as comparing against the result
-- of 'length', but can short circuit if the count of characters is
-- greater than the number, and hence be more efficient.
compareLength :: Text -> Int -> Ordering
compareLength t c = S.compareLengthI (stream t) c
{-# INLINE [1] compareLength #-}

{-# RULES
"TEXT compareN/length -> compareLength" [~1] forall t n.
    compare (length t) n = compareLength t n
#-}
```

Here phase control is used to indicate that `compareLength` should **only** be inlined from phase 1 onward and conversely that the rewrite rule **compareN/length** may be applied **except** in phase 1. What this ensures is that the inliner will not operate on the result of the rewrite rule directly in the same phase. This is supposedly because we expect `compareLength` to occur in the LHS of a different rewrite rule which is to be desired over inlining at first.

2.2.3 The worker/wrapper transformation

The *worker/wrapper* transformation is able to change the type of a computation (typically a loop) into a *worker* function along with a *wrapper* that can convert back to the original type. The example listed on the GHC wiki is that of the reverse function on lists [2]. One might concisely define `reverse` with direct recursion and the `++` operator:

```haskell
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Of course this is not very efficient as the `++` operator itself is already linear, making the reverse function quadratic; If we create an auxiliary **worker** function `reverse'` that takes an extra accumulator argument, we can create a linear version:

```haskell
reverse :: [a] -> [a]
reverse = reverse' []
  where reverse' acc [] = acc
        reverse' acc (x:xs) = reverse' (x:acc) xs
```

Here the **wrapper** is simply applying the empty list as a starting argument to the function. Thus, by introducing some state that exists only during the lifetime of the computation, the function has become asymptotically more efficient.

Impressively, the transformation may also greatly improve the constant factor of the runtime by leveraging unboxed types. Unlike strict languages, the `Int` type in Haskell – despite always evaluating to a 64 bit integer – is itself not statically sized. After all, any value is lazy and may therefore still be an unevaluated thunk whose size is unknown at compile time. As a result, `Int`s are always stored on the heap and thus require no runtime allocation. However, `Int` has a strict counterpart `Int#` (unboxed integer) in which it is defined: `data Int = I# Int#`. Computations on `Int#`s can be evaluated much more cheaply since such values can be stored on the stack.

Knowing this, we can opt to create a worker that does add state to the computation, but changes the types to their unboxed counterparts. Naturally the wrapper operation is then simply the constructor of the lazy type `I#`. Let us consider the example of the recursive `triangular` function which given a number `n` returns the sum of all numbers from 1 to `n`:

```haskell
triangular :: Int -> Int
triangular 0 = 0
triangular n = n + triangular (n-1)
```

Although an all-knowing compiler could have determined that the result of `triangular` can simply be calculated numerically in constant time as $\frac{n^2+n}{2}$, we can still trust GHC to infer an important property about the function using strictness analysis. Namely: `triangular` does not produce any intermediate results that can be used without evaluating the whole of `triangular`. That is: if you are willing to spent any amount of time on `triangular`, you might as well calculate the whole thing. Thus, GHC decides to rewrite the function using a strict worker, removing the need for many dubious allocations:

```haskell
wtriangular :: Int# -> Int#
wtriangular 0 = 0
wtriangular n = GHC.Prim.+# n (wtriangular (GHC.Prim.-# n 1))

triangular :: Int -> Int
triangular (I# w) = I# (wtriangular w)
```

To get a feeling for the difference in performance, we can compare the runtime of the original function with the worker/wrapper version:

Snippet (compiled with -O0)	**Result of `time` with input** 10000000
`original`	0,10s user 0,03s system 99% cpu 0,132 total
`transformed`	0,01s user 0,01s system 98% cpu 0,023 total

Table 2.1: The runtime of both version of `triangular`. We can see that the worker/wrapper tranformation has increased runtime performance by a factor of $0.132/0.023 = 5.7$.

It might not be immediately intuitive why the performance differs so drastically and where exactly these seemingly evil allocation occur. The ridiculousness of the original function becomes apparent when we consider a C implementation with the same allocation behavior. Although lacking in laziness, we can consider an `Int` to map to a `long*` (pointing to heap allocated memory) and an `Int#` to map to a plain `long`.

```c
#include <stdio.h>
#include <stdlib.h>

// utility function for heap allocating a 64 bit integer
long* alloc_long() { return (long*)malloc(sizeof(long)); }

long* triangular(const long* n_ptr) {
  // allocate the return value
  long* ret = alloc_long();
  // derefence the pointer into a value
  long n = *n_ptr;
  if (n==0) { *ret = 0; } else {
    // allocate a new pointer with which to call the function recursively
    long* inp_ptr = alloc_long();
    *inp_ptr = n-1 ;
    *ret = n + *triangular(inp_ptr);
  }
  return ret;
}
```

Rest assured that any C programmer suggesting this implementation would get some very weird looks.

2.2.4 Analysis transformations

Consider again the Core ADT given in Section 2.1, it was mentioned that the `Var` type is used to represent variables and their various properties. We look into the one field of `Var` that is dynamic during the core2core pipeline: `IdInfo`.

This data type follows the concept of weakening of information, i.e. the information is never incorrect but may be less precise or even missing. Furthermore, the `IdInfo` may differ for different `Var` instances that refer to the same variable.

It is as of `ghc-9.4.2` defined as:

```haskell
data IdInfo
  = IdInfo {
        ruleInfo        :: RuleInfo,
        -- ^ Specialisations of the 'Id's function which exist.
        -- See Note [Specialisations and RULES in IdInfo]
        realUnfoldingInfo   :: Unfolding,
        -- ^ The 'Id's unfolding
        inlinePragInfo  :: InlinePragma,
        -- ^ Any inline pragma attached to the 'Id'
        occInfo         :: OccInfo,
        -- ^ How the 'Id' occurs in the program
        dmdSigInfo      :: DmdSig,
        -- ^ A strictness signature. Digests how a function uses its arguments
        -- if applied to at least 'arityInfo' arguments.
        cprSigInfo      :: CprSig,
        -- ^ Information on whether the function will ultimately return a
        -- freshly allocated constructor.
        demandInfo      :: Demand,
        -- ^ ID demand information
        bitfield        :: {-# UNPACK #-} !BitField,
        -- ^ Bitfield packs CafInfo, OneShotInfo, arity info, LevityInfo, and
        -- call arity info in one 64-bit word. Packing these fields reduces size
        -- of `IdInfo` from 12 words to 7 words and reduces residency by almost
        -- 4% in some programs. See #17497 and associated MR.
        --
        -- See documentation of the getters for what these packed fields mean.
        lfInfo          :: !(Maybe LambdaFormInfo),

        -- See documentation of the getters for what these packed fields mean.
        tagSig          :: !(Maybe TagSig)
    }
```

Through non-structure-altering transformations like *Demamd Analysis* and *Strictness Analysis* the `IdInfo` record is updated accordingly. This information may then be used by future transformations that can only optimise safely or productively under certain circumstances (remember the disastrous work duplication in Section 2.2.1?).

The information contained in `IdInfo` is a major source of complexity when it comes to comprehending Core. Consider for example this 'pretty'-printed `IdInfo` instance:

```
[GblId,
 Arity=4,
 Str=<L,U(U(U(U(C(C1(U)),A,C(C1(U)),C(U),A,A,C(U)),A,A),1*U(A,C(C1(U)),A,A),A,A,A,A,A),U(A,A,C(U),...etc.
 Unf=Unf{Src=<vanilla>, TopLvl=True, Value=True, ConLike=True,
         WorkFree=True, Expandable=True, Guidance=IF_ARGS [50 0 0 0] 632 0}]
```

We can quickly learn of few things about the variable it describes. First, it is apparently a function that has an arity of 4 (i.e. it takes 4 arguments). Secondly, we obtain some of the magic heuristic values involved with inlining (also known *unfolding* hence `Unf`). However, if we want to diagnose why for example this variable was evaluated lazily even though it is always used exactly once, we would have to decode the strictness signature under `Str`. Currently, the best resource for decoding this would be to read the comments in the GHC source code itself.

2.3 The fusion system

Fusion – the process of combining multiple operations over a structure into a single operation – is in GHC implemented using the *build/foldr* idiom, coined as *short-cut fusion*. This system was developed as an improvement on deforestation (Wadler [25]) which has shortcomings when it comes to recursive functions [12].

We borrow an example from Seo [28] to illustrate how the build/foldr system works. Consider the example – coincidentally very similar to `triangular` (Section 2.2.3) – of summing a list of numbers:

```haskell
foldr (+1) 0 [1..10]
```

Very concise indeed but its execution would be much slower than say, an imperative for loop, because we first create a list and then consume it right away. If we look inside the definition of `[1..10]` we find the function `from`:

```haskell
from :: (Ord a, Num a) => a -> a -> [a]
from a b = if a > b
           then []
           else a : from (a + 1) b
```

Which is itself a specialisation for lists. We can write a more generic catamorphism that takes any duo of constructors of type `a -> b -> b` and `b` respectively (previously `:` and `[]`):

```haskell
from' :: (Ord a, Num a) => a -> a -> (a -> b -> b) -> b -> b
from' a b = \c n -> if a > b
                    then n
                    else c a (from' (a + 1) b c n)
```

The glue between `from` and `from'` is the *build* function, which requires the `Rank2Types` language extension and re-specialises these generic functions back to lists.

```haskell
build :: forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []

from a b = build (from' a b )
```

Thus far it seems we have only introduced more complexity without any apparent benefit. However, this is the point that we run into the key idea: `build` is the antithesis of `foldr` such that the following holds:

```haskell
foldr k z (build g) = g k z
```

And this also gives us the main reductive rewrite rule to make this system work. So to summarize, we have shown that generalizing lists to list producing functions to catamorphisms over list like arguments, produces a common interface exposing the elimination of code. More concretely, because we keep `g` as a generic function, we can choose to give it the arguments of the consequent `foldr` directly and thus eliminate the intermediate list. This is again a situation like the assembly line workers analogy from Section 1.2.4; It is very productive to communicate about functions taking and producing lists, but actually doing so in a chained context is like wrapping and unwrapping boxes at every step.

Analogies aside, observe how our original expression can now completely be reduced to a single integer value by rewriting `from` with this common interface.

```haskell
foldr (+) 0 (from 1 10)
-- { inline from }
foldr (+) 0 (build (from' 1 10))
-- { apply rewrite rule }
from' 1 10 (+) 0
-- { inline from' }
\c n -> (if 1 > 10
         then n
         else c 1 (from' 2 10 c n)) (+) 0
-- { beta reduce }
if 1 > 10
   then 0
   else 1 + (from' 2 10 (+) 0)
-- { repeat until base case }
1 + 2 + ... + 9 + 10 + 0
-- { constant fold }
55
```

But what if there is no `foldr` in the expression? What if we use simpler functions like `map` or `filter`? Well, as most introductory Haskell courses are likely to tell you, most list functions can be defined in terms of `foldr`. Thus, we can similarly use rewrite rules to map all these common list functions to a combination `build` and `foldr`. This idea also relieves us from the burden of having to define rewrite rules for all the million possible combination of lists operations:

```haskell
map f xs     = build (\ c n -> foldr (\ a b -> c (f a) b) n xs)
filter f xs = build (\ c n -> foldr (\ a b -> if f a then c a b else b) n xs)
xs ++ ys     = build (\ c n -> foldr c (foldr c n ys) xs)
concat xs    = build (\ c n -> foldr (\ x y -> foldr c y x) n xs)
repeat x     = build (\ c n -> let r = c x r in r)
zip xs ys    = build (\ c n ->
    let zip' (x:xs) (y:ys) = c (x,y) (zip' xs ys)
        zip' _ _ = n
    in  zip' xs ys)
[]           = build (\ c n -> n)
x:xs         = build (\ c n -> c x (foldr c n xs))
```

We can see some these functions at work when looking at the definition of `unlines` and its subsequent reductions through the build/foldr system:

```haskell
unlines :: [String] -> String
unlines ls = concat (map (\l -> l ++ ['\n']) ls)
-- { rewrite concat and map }
unlines ls = build
  (\c0 n0 -> foldr (\xs b -> foldr c0 b xs) n0 ( build
    (\c1 n1 -> foldr (\l t -> c1 (build
      (\c2 n2 -> foldr c2 ( foldr c2 n2 ( build
        (\c3 n3 -> c3 '\n' n3))) l )) t) n1 ls )))
-- { apply rewrite rule foldr/build }
unlines ls = build
  (\c0 n0 ->
    (\c1 n1 -> foldr (\l t -> c1 (build
      (\c2 n2 -> foldr c2 (
        (\c3 n3 -> c3 '\n' n3) c2 n2 ) l)) t) n1 ls)
          (\xs b -> foldr c0 b xs) n0)
-- { beta reduce }
unlines ls = build
  (\ c0 n0 -> foldr (\l t -> foldr c0 t( build
    (\c2 n2 -> foldr c2 (c2 '\n' n2) l))) n0 ls)
-- { apply rewrite rule foldr/build }
unlines ls = build
  (\c0 n0 -> foldr (\l b -> foldr c0 (c0 '\n' b) l) n ls)
-- { inline build }
unlines ls = foldr (\l b -> foldr (:) ('\n' : b) l) [ ] ls
-- { inline foldr }
unlines ls = h ls
    where h [] = []
          h (l:ls) = g l
              where g [] = '\n' : h ls
                    g (x:xs) = x : g xs
```

What we end up with is the most efficient implementation of `unlines` that we could possibly write by hand [12]. Because the list ls is the input of the function we cannot remove it all together, but keep in mind that when inlining call to this function at the use site, foldr/build fusion may apply again.

2.4 Contemporary Haskell comprehension

2.4.1 Communicating in Core

We are not pioneers discovering the land of Core inspection. Since its inception, Core has been used to communicate about programs and compiler interactions. Sifting through open issues on the GHC compiler itself, one quickly comes across discussions elaborated by Core snippets. Consider issue #22207 titled '*bytestring Builder performance regressions after 9.2.3*' for example.

While testing the performance characteristics of a bytestring patch meant to mitigate withForeignPtr-related performance regressions, it was noticed that several of our Builder-related benchmarks have also regressed seriously for unrelated reasons. The worst offender is byteStringHex, which on my machine runs about 10 times slower and allocates 21 times as much when using ghc-9.2.4 or ghc-9.4.2 as it did when using ghc-9.2.3. Here's a small program that can demonstrate this slowdown:

The author then provides two snippets containing the final Core representation of `byteStringHex` as produced by the two different GHC version. Each of these documents contain around 400 lines of unhighlighted code annotated with all available information. And while having all available information sounds like a good thing (and it is in a way) it poses a serious practicality issue. Namely: it is exceedingly difficult for a human to read and comprehend a certain aspect of the AST while having to filter out another. Not to mention, it solidifies reading Core as an activity reserved for only the most expert Haskell developers by scaring others away with a steep barrier to entry.

2.4.2 Current tooling

GHC itself

Core snippets of your program can easily be coerced out of GHC. The most information you can get is the Core AST at each pass of the optimisation pipeline by using `-ddump-core2core`. To reduce the signal-to-noise ratio of Core snippets, one can use any number of suppression options. It is common to suppress type arguments and type applications for example. These are commonly uninteresting to explicitly display because they are easily inferred by arguments to applications. Although types do sometimes influence the optimisation transformations – making them interesting for display – they generate such a degree of syntactical noise that suppression is often desirable.

As can be read in the GHC documentation, the following suppression flags are available to help to tame the beast.

GHC flag	Effect on Core printing
`-dsuppress-all`	In dumps, suppress everything (except for uniques) that is suppressible.
`-dsuppress-coercions`	Suppress the printing of coercions in Core dumps to make them shorter
`-dsuppress-core-sizes`	Suppress the printing of core size stats per binding (since 9.4)
`-dsuppress-idinfo`	Suppress extended information about identifiers where they are bound
`-dsuppress-module-prefixes`	Suppress the printing of module qualification prefixes
`-dsuppress-ticks`	Suppress "ticks" in the pretty-printer output.
`-dsuppress-timestamps`	Suppress timestamps in dumps
`-dsuppress-type-applications`	Suppress type applications
`-dsuppress-type-signatures`	Suppress type signatures
`-dsuppress-unfoldings`	Suppress the printing of the stable unfolding of a variable at its binding site
`-dsuppress-uniques`	Suppress the printing of uniques in debug output (easier to use diff)
`-dsuppress-var-kinds`	Suppress the printing of variable kinds

We can show how these suppression options greatly improve the readability of Core snippets by comparing the desugared (unoptimized) Core of `quicksort` with and without the `-dsuppress-all` flags.

First the source:

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort (filter (< x) xs) ++ [x] ++ quicksort (filter (>= x) xs)
```

The desugared Core without suppression:

```
-- RHS size: {terms: 55, types: 47, coercions: 0, joins: 0/10}
quicksort :: forall a. Ord a => [a] -> [a]
[LclIdX]
quicksort
  = \ (@ a_a1pH) ($dOrd_a1pJ :: Ord a_a1pH) ->
      let {
        $dOrd_a1w8 :: Ord a_a1pH
        [LclId]
        $dOrd_a1w8 = $dOrd_a1pJ } in
      let {
        $dOrd_a1w5 :: Ord a_a1pH
        [LclId]
        $dOrd_a1w5 = $dOrd_a1pJ } in
      \ (ds_d1wq :: [a_a1pH]) ->
        case ds_d1wq of wild_00 {
          [] -> ghc-prim-0.6.1:GHC.Types.[] @ a_a1pH;
          : x_a1hP xs_a1hQ ->
            letrec {
              greater_a1hS :: [a_a1pH]
              [LclId]
              greater_a1hS
                = let {
                    $dOrd_a1pQ :: Ord a_a1pH
                    [LclId]
                    $dOrd_a1pQ = $dOrd_a1pJ } in
                  letrec {
                    greater_a1pT :: [a_a1pH]
                    [LclId]
                    greater_a1pT
                      = filter
                          @ a_a1pH
                          (let {
                             ds_d1wF :: a_a1pH
                             [LclId]
                             ds_d1wF = x_a1hP } in
                           \ (ds_d1wE :: a_a1pH) -> > @ a_a1pH $dOrd_a1pQ ds_d1wE ds_d1wF)
                          xs_a1hQ; } in
                  greater_a1pT; } in
            letrec {
              lesser_a1hR :: [a_a1pH]
              [LclId]
              lesser_a1hR
                = let {
                    $dOrd_a1vV :: Ord a_a1pH
                    [LclId]
                    $dOrd_a1vV = $dOrd_a1pJ } in
                  letrec {
                    lesser_a1vY :: [a_a1pH]
                    [LclId]
                    lesser_a1vY
                      = filter
                          @ a_a1pH
                          (let {
                             ds_d1wD :: a_a1pH
                             [LclId]
                             ds_d1wD = x_a1hP } in
                           \ (ds_d1wC :: a_a1pH) -> < @ a_a1pH $dOrd_a1vV ds_d1wC ds_d1wD)
                          xs_a1hQ; } in
                  lesser_a1vY; } in
            ++
              @ a_a1pH
              (quicksort @ a_a1pH $dOrd_a1w5 lesser_a1hR)
              (++
                 @ a_a1pH
                 (GHC.Base.build
                    @ a_a1pH
                    (\ (@ a_d1wx)
                       (c_d1wy :: a_a1pH -> a_d1wx -> a_d1wx)
                       (n_d1wz :: a_d1wx) ->
                       c_d1wy x_a1hP n_d1wz))
                 (quicksort @ a_a1pH $dOrd_a1w8 greater_a1hS))
        }
```

The same desugared Core representation with the `-dsuppress-all` flag:

```
-- RHS size: {terms: 55, types: 47, coercions: 0, joins: 0/10}
quicksort
  = \ @ a_a1pH $dOrd_a1pJ ->
      let { $dOrd_a1w8 = $dOrd_a1pJ } in
      let { $dOrd_a1w5 = $dOrd_a1pJ } in
      \ ds_d1wq ->
        case ds_d1wq of wild_00 {
          [] -> [];
          : x_a1hP xs_a1hQ ->
            letrec {
              greater_a1hS
                = let { $dOrd_a1pQ = $dOrd_a1pJ } in
                  letrec {
                    greater_a1pT
                      = filter
                          (let { ds_d1wF = x_a1hP } in
                           \ ds_d1wE -> > $dOrd_a1pQ ds_d1wE ds_d1wF)
                          xs_a1hQ; } in
                  greater_a1pT; } in
            letrec {
              lesser_a1hR
                = let { $dOrd_a1vV = $dOrd_a1pJ } in
                  letrec {
                    lesser_a1vY
                      = filter
                          (let { ds_d1wD = x_a1hP } in
                           \ ds_d1wC -> < $dOrd_a1vV ds_d1wC ds_d1wD)
                          xs_a1hQ; } in
                  lesser_a1vY; } in
            ++
              (quicksort $dOrd_a1w5 lesser_a1hR)
              (++
                 (build (\ @ a_d1wx c_d1wy n_d1wz -> c_d1wy x_a1hP n_d1wz))
                 (quicksort $dOrd_a1w8 greater_a1hS))
        }

```

A drastic improvement for sure, but there are still some things to be left desired. A simpler language like Core generally needs more code to express the same thing, we can thus expect to generate more data than the original Haskell code. Moreover, should you be interested the state off the program at each of the intermediate steps, you can expect to see about 20x more data still. Unless you know exactly what to search for, this begs for a more ergonomic, filtered view of the data.

GHC Plugins

GHC – by nature a playground for academics and enthusiasts alike – is extremely flexible when it comes to altering its functionality. Using the now well established plugin interface, one is able to hook into almost any operation of the front- and midsection of the compiler. One such place is managing the core2core passes that will be run on the current module. This point of entry can be used to intersperse each core2core pass with an identity transformation that smuggles away a copy of the AST in its full form as a side effect.

One such existing plugin is `ghc-dump` [10]. Besides extracting intermediate ASTs, it defines an auxiliary Core definition to which it provides a GHC version agnostic conversion. This has the increased benefit of being able to directly compare snapshots from different GHC versions; A not uncommon task as discussed in Section 2.4.1. And while certainly being an improvement over plain text representation, we believe exploring and comparing such dumps requires a more rich interface.

Chapter 3

Methods

We describe how we made `hs-sleuth`, the tool proposed in this thesis. We start with a general overview of the architecture and then zoom in on all the technical design decisions made during the implementation.

3.1 Requirements

The following prime requirements are identified to guide the implementation process.

- **GHC $\geq$ 8.4 cross compatible**, Important to facilitate inspecting the effects of changes in the compiler on the same source.
- **Simple and non invase steps to create dumps**, Large and established code bases should be able to use `hs-sleuth`.
- **Cross-platform ability to explore dumps**, `hs-sleuth` should be able to run on all major platforms, preferably without any additional dependencies.
- **Extendable**, Not everything needs to be supported (think unfoldings) but should be extendable in the future.

3.2 Architecture

We envisioned a high degree of interactability with the snapshots of the intermediate ASTs. To realise this in a cross-platform, dependency-free fashion, we decided to use a browser based frontend application. Because the concept of mutually recursive algebraic datatypes are very pervasive in the Core AST, we felt it would be extremely helpful if the frontend language had first class support for this. This quite naturally led to us to Elm, a functional language that compiles to Javascript [6].

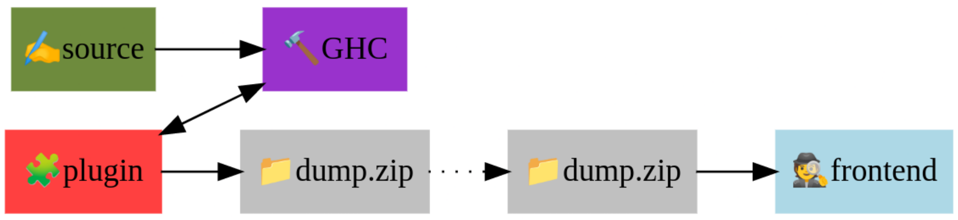


Figure 3.1: Dataflow diagram of `hs-sleuth`

3.3 Creating the GHC plugin

The least invasive option would be to parse the output of GHC given a number of `-ddump` flags. However, for the sake of convenience and robustness, we instead decided to create a plugin that hooks directly into the core2core pipeline and captures the ASTs completely. By interspersing each transformation with a snapshot action, we extract as much information from GHC as we could reasonably hope for.

3.3.1 Capturing the information

If we wish to support multiple recent versions of GHC we need to deal with the fact that the Core ADT has undergone a few minor changes and additions. We believe that the solution is to create some auxiliary definition to which we can map various versions of the Core ADT. This was done very efficiently by building upon the existing `ghc-dump` package, which already implemented such a representation as well as a version agnostic conversion module with the help of `min_version_GHC` macro statements [10].

What `ghc-dump` also intelligently addresses, is the issue of possible infinite recursion. This problem arises through the unfolding inside the `IdInfo` struct attached to each variable. This effectively makes any Core expression closed as the binding information is stored in the variable itself, to great utility of the inliner. However, when a variable represents a recursive – or even mutually recursive – value, the inlining will contain itself. This fact implies that we can never serialise the AST to a finite value. Therefore, `ghc-dump` demotes each usage site of a variable to an identifier referencing its binding site. This allows us to obtaining a finite representation that we can later reconstruct by traversing the AST with an environment.

3.3.2 Globally unique variables

It is not strictly necessary for variable names in a Core program to be unique. A variable name always references the nearest binding site. However, is not very convenient when we want to analyze a certain variable in isolation. After all, we cannot know if two separate free variables with the same name are actually the same variable or live in a different scope. Consider the definition of tail:

```haskell
tail xs = case xs of
  x:xs -> xs
  _    -> error "tail of empty list"
```

We cannot simply refer to the variable `xs` as that name has two different binding sites. We solve this by running a *uniquefication pass* as part of each snapshot that freshens all duplicate names in the entire module after each core2core transformation. After this operations every variable is globally unique. This gives us the ability to refer to a binding site and its usages unambiguously using simply an identifying integer. The big idea here is that any viewing logic is completely decoupled from binding semantics:

```haskell
tail xs_0 = case xs_0 of
  x_1:xs_2 -> xs_2
  _        -> error "tail of empty list"
```

It is possible to omit the numbered suffixes when displaying the AST, but internally it is very useful to be able to make this distinction without any further effort.

3.3.3 Detecting changes

If a module is of a slightly larger size, it becomes difficult to spot the changes made by a certain transformation, if there even are any. To address this, we decided to develop a feature that allows for the filtering of code that remains unchanged. Let us define what unchanged means in this context. It is important to make the subtle distinction between syntactic equivalence and α-equivalence. The difference is that the latter is agnostic to the names of variables, as long as they refer to the same binding site.

We can quickly solve the decision problem of syntactic equality by calculating a hash of an expression beforehand and simply checking for equality of this hash value. We considered using recent improvements of full sub expression matching [17], but decided against it as it was not clear how to effectively present the results nor did it rarely prove useful to isolate changes in the AST as they were rarely local to begin with. Instead, we opted for a far simpler approach where we only hash the top-level definitions, and provide a more crude option to hide any top-level definitions that have not changed at all.

All in all, we still recommend that issues are attempted to be reproduced in small modules as the amount of noise can quickly become overwhelming despite change detection.

3.4 Creating the frontend application

We begin with a brief introduction to the Elm language and its concepts. Elm is very much a domain specific language; It is similar enough to Haskell to be familiar yet sufficiently simplified to be a frontend only language. It constrains the architecture to a trinity of concepts:

- **Model** - The state of the application.
- **Message** - Typically abbreviated to `Msg`, this describes all the events that can occur and are processed by the `update : Msg -> Model -> Model` function.
- **View** - The way the state is rendered: `view: Model -> Html Msg`. The `Html` type is parameterized over the `Msg` such that it event emitters like `onClick` can only produce `Msgs` that are handled by `update`.

The big idea is to have exclusively *pure* and *complete* functions to handle viewing and updates. These updates are triggered by emitted `Msgs` that are the result of user interaction like hovering, clicking, etc. The increment/decrement example is a testament to the simplicity focused design of the language [6]:

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main = Browser.sandbox { init = 0, update = update, view = view }

type Model = Int
type Msg = Increment | Decrement

update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment -> model + 1
    Decrement -> model - 1

view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Unlike Haskell, there is no explicit `IO` in user code. All side effects are encapsulated by the framework, typically in the form of a function that takes a `Msg` constructor and populates it with a `Result x a` value for failure handling. Given this situation it feels justified to disallow any form of errors and by extent incomplete functions. This powerful property gives us great confidence in the robustness of our application.

3.4.1 Reproducing the AST

It would have been extremely tedious to have to constantly maintain a Core ADT in Elm along with a JSON parser that is compatible with the JSON output of the Haskell plugin. Luckily, we were able to use the `haskell-to-elm` [9] package to automatically generate all the required boilerplate code.

For example the `Alt` datatype – representing an arm of a case expression – is defined as follows in our AST:

```haskell
data Alt = Alt
    { altCon :: AltCon          -- The constructor being matched on
    , altBinders :: [Binder]    -- The variables being bound during the deconstruction
    , altRHS :: Expr            -- The right-hand side of the case arm
    }
    deriving (Generic)

deriving instance SOP.Generic Alt
deriving instance SOP.HasDatatypeInfo Alt
type AltElm = ElmType "Generated.Types.Alt" Alt
deriving via AltElm instance Aeson.ToJSON Alt
deriving via AltElm instance HasElmType Alt
deriving via AltElm instance HasElmDecoder Aeson.Value Alt
```

Note how we derive the `Aeson.ToJSON` instance via the `ElmType` machinery. This control allows us to generate a compatible and robust JSON parser for Elm. The auto-generated Elm datatype and parser (`Decoder` in Elm speak) look like this:

```elm
type alias Alt =
    { altCon : AltCon
    , altBinders : List Binder
    , altRHS : Expr
    }

altDecoder : Json.Decode.Decoder Alt
altDecoder =
    Json.Decode.succeed Alt |>
    Json.Decode.Pipeline.required "altCon" altConDecoder |>
    Json.Decode.Pipeline.required "altBinders" (Json.Decode.list binderDecoder) |>
    Json.Decode.Pipeline.required "altRHS" exprDecoder
```

Conveniently, it would be very easy in the future to extend the Haskell ADT with additional information because it is the single source of truth; The Elm type and JSON machinery can then be regenerated with a single command.

Additionally, we use this pipeline embellish the AST to allow for reconstruction of the demoted call sites (see Section 3.3.1). Specifically, we add a field to each `BinderId` of the type:

```elm
type BinderThunk = Found Binder | NotFound | Untouched
```

Initially this field is set to the `Untouched` variant. Once the finitely encoded AST is decoded into the Elm universe, a reconstruction traversal takes place that, with the help of an environment, strengthens the `BinderThunk` with a reference to its binder. The `NotFound` variant exits purely for verification purposes and should never occur with sound inputs. At this time, the unfoldings of binders are not yet considered, breaking the infinite recursion problem. This important because unlike Haskell Elm is strict, the reconstruction would thus never terminate on an infinite datatype. In the future, it is conceivable to run an on demand reconstruction as Elm. Furthermore, the reconstruction pass does not significantly increase memory usage as Elm's single static assignment semantics dictate that the fields contained in a `BinderThunk` are references not copies.

After this initial reconstruction we again no longer need to keep binding semantics in mind, and we can isolate subexpressions without losing information about the binding site of now seemingly free variables.

3.4.2 Pretty printing

It certainly should be considered useful to display the Core in exactly the same way that GHC does. After all, this is what programmers are currently already used to and its design has been given a lot of thought.

However, we felt a need to first create a separate representation that is tailored to those who have not seen Core before. We discuss the pros and cons of this decision more extensively in Section 5.3.

Haskell programmers that care enough to inspect the interaction with the compiler are likely to be avid readers of at least basic Haskell syntax. We believe that such a representation is a suitable way to minimize the shock reactions in newcomers, as well as provide a more comfortable viewing experience for those who are primarily interested in the structure of their program throughout optimisation. Therefore, we decided to create a pretty printer that attempts to be as similar to Haskell source as possible. Just like GHC, we used the pretty printing method developed by Wadler [26], implemented in Elm using `elm-pretty-printer` [23].

To compare, consider again the Core representation presented in Section 2.4.2 with the quicksort representation produced by our pretty printer:

```
quicksort :: forall a. Ord a -> [a] -> [a]
quicksort a $dOrd ds = case ds of
  { : x xs -> GHC.Base.++ @a
                (quicksort @a $dOrd
                   (GHC.List.filter @a
                      (\ds -> GHC.Classes.< @a $dOrd ds x) xs))
                (GHC.Base.++ @a
                   (GHC.Base.build @a
                      (\a c n -> c x n))
                   (quicksort @a $dOrd
                      (GHC.List.filter @a
                         (\ds -> GHC.Classes.> @a $dOrd ds x) xs)))
    [] -> GHC.Types.[] @a
  }
```

Notice that although this might look like normal Haskell, it contains explicit type variables like `a`, hinting at the SystemFC nature of Core. Furthermore, operators application are written in Polish notation, that is the operator comes before its arguments. We decided this to be more sensical because the type values muddy the waters when it comes to the clarity of infix notation, note this alternative form:

```
-- Polish notation
\ds -> GHC.Classes.> @a $dOrd ds x

-- Infix notation
\ds -> ds (GHC.Classes.> @a $dOrd) x
```

To facilitate syntax highlighting, the pretty printer adds the appropriate token identifiers such that `pygments` [11] can be used to colorize the output.

3.4.3 Including the source

It goes without saying that the source code of a module is an essential part of any analysis. Therefore, the plugin copies the source code and runs it through the `pygmentize` [11] tool to obtain an HTML representation of the source code that is highlighted exactly the same way as the pretty printed Core. This HTML source code is embedded in the output of the dump.

The artifacts produced by a single module now look like this:

```
Quicksort_0.json
Quicksort_10.json
Quicksort_11.json
Quicksort_12.json
Quicksort_13.json
Quicksort_14.json
Quicksort_15.json
Quicksort_16.json
Quicksort_17.json
Quicksort_18.json
Quicksort_1.json
Quicksort_2.json
Quicksort_3.json
Quicksort_4.json
Quicksort_5.json
Quicksort_6.json
Quicksort_7.json
Quicksort_8.json
Quicksort_9.json
Quicksort.html
Quicksort_meta.json
```

All these files will be compressed in a zip archive to keep the size of the output smaller. It also makes for a convenient atomic entity to load up in the frontend application where it is unzipped on the fly.

3.4.4 Unfoldings and capture sizes

As mentioned in Section 3.3.1, the entire body of a variable is sometimes saved as part of its `IdInfo` which is exported to interface files such that inlining can take place across modules. This part of the `IdInfo` struct is called the *unfolding* of a variable. By nature, this can be a very large expression. At this time, we have no need for the exact unfolding in the frontend application, other than knowing it exists. Therefore, we decided to replace any unfolding with a string literal to indicate that the unfolding was removed:

```haskell
removeUnfolding :: Unfolding -> Unfolding
removeUnfolding u@CoreUnfolding {..} =
  u { unfTemplate = ELit (MachStr (T.pack "unfolding removed by plugin")) }
removeUnfolding x = x
```

This reduces the size of captures in some cases by a factor of 2.

3.4.5 Interactions

Aside from a more human-readable representation and syntax highlighting, the user is further supported by a number of interactive features. These include:

- Highlighting binding and call sites of a variable on hover
- Renaming variables
- Toggling various Core display options such as hiding typelevel terms and type applications, desugaring leading lambda abstractions, etc.
- Multiple approaches to hiding toplevel bindings such as hiding all but the selected binding, which can conveniently be followed up by un-hiding its referenced bindings transitively.
- A variable detail popup that shows all the available information of a variable.
- Querying variables and their types on Hoogle.

3.4.6 Improving performance with cache semantics

The way that Elm operates – like many other interactive browser based applications – is by rendering the HTML on every update and then diffing the result with the current state of the DOM to create a minimal change list. This is generally a very good strategy because mutating the DOM is by far the most dominant cost contributor. However, when it comes to pretty printing a large Core expression, the `view` function starts to incur a significant cost. This is wasteful because many updates do not actually affect the output of the pretty printer (events like `onHover` and `onClick` for example). Initially this led to some serious usability issues where the application would stutter and freeze.

We were able to overcome this problem using the `Html.Lazy` module. which takes some HTML producing closure and its arguments separately. If on some update the arguments have not changed compared to the last update, the closure is not evaluated and a cached result is returned. Regardless, the aforementioned diffing still takes place, reducing the cost of the update to something negligible. Of course, we cannot get around the fact that any updates that do affect the output of the pretty printer might still cause some stutters, but the frequency of such updates is generally far lower. A dataflow diagram is shown in Fig. 3.2.

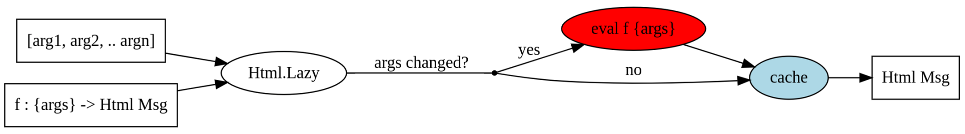


Figure 3.2: A graph showing the logic behind the caching mechanism.

Elegantly, because `f` is guaranteed to be pure, the cache is guaranteed to always return the same value as the evaluation would have.

3.4.7 Note on deployment

By virtue of being compiled to solely a stateless Html/JS application, the frontend can easily and cheaply be deployed to any static file hosting service. Because the dump files are never send to the server, we can discard any privacy concerns while still providing a no effort method to analyze the dumps. Anyone is still free of course, to build and host their own build of the frontend which is similarly open sourced.

Currently some CSS files are served from a CDN but it would be trivial to bundle them with the application, making even an internet connection no longer a requirement.

Chapter 4

Results

We evaluate our tool by applying it to a number of real world cases. Firstly we reproduce the issue underlying the work of `inspection-testing` [3], which also serves as a comprehensive, didactic example of how to use our tool. The second case study is about stream fusion and focuses on the way that the theory differs from the true implementation. In doing so, it covers a large spectrum of GHC's optimisation pipeline. Thirdly, we present how we were able to use our tool to find a performance bug in GHC itself and how we were furthermore able to use it to verify a possible solution. Finally, we compare our experience with `hs-sleuth` with that of using the existing output that can be obtained from GHC today.

4.1 Diagnosing a failing inspection test in `Text`

Hearkening back to `inspection-testing` discussed in Section 1.2.4, we put ourselves in the shoes of a programmer who gets surprised by a failing inspection while using `text-1.2.3.2` test and reason how `hs-sleuth` may be employed to diagnose the problem.

To summarize, we expected that the function `countChars` – which counts the number of characters in a `ByteString` by composing three functions in the `Text` library – will in its final form not actually construct a `Text` value.

1. Isolate the problem Modules typically contain more than 1 function and during the core2core transformations many more auxiliary functions are added. Furthermore, many functions are inlined to produce ever more code. Despite `hs-sleuth` being designed with features to comprehend medium-sized modules, it is still most helpful to temporarily isolate the failing test case into a separate module:

```haskell
{-# LANGUAGE TemplateHaskell #-}

module InspectionTests where

import Test.Inspection
import qualified Data.Text as T
import qualified Data.Text.Encoding as TE
import Data.ByteString

countChars :: ByteString -> Int
countChars = T.length . T.toUpper . TE.decodeUtf8

-- the failing test case
inspect $ 'countChars `hasNoType` ''T.Text
```

The following error is produced by the build:

```
app/InspectionTests.hs:21:1: countChars `hasNoType` Data.Text.Internal.Text failed:
# ...
# 700 lines of Core as seen at the end of the core2core pipeline
# ...
```

700 lines of textual data is generated by `inspection-testing` from just this one function! It is also incomplete in the sense that it does not show the process that produced this final artifact.

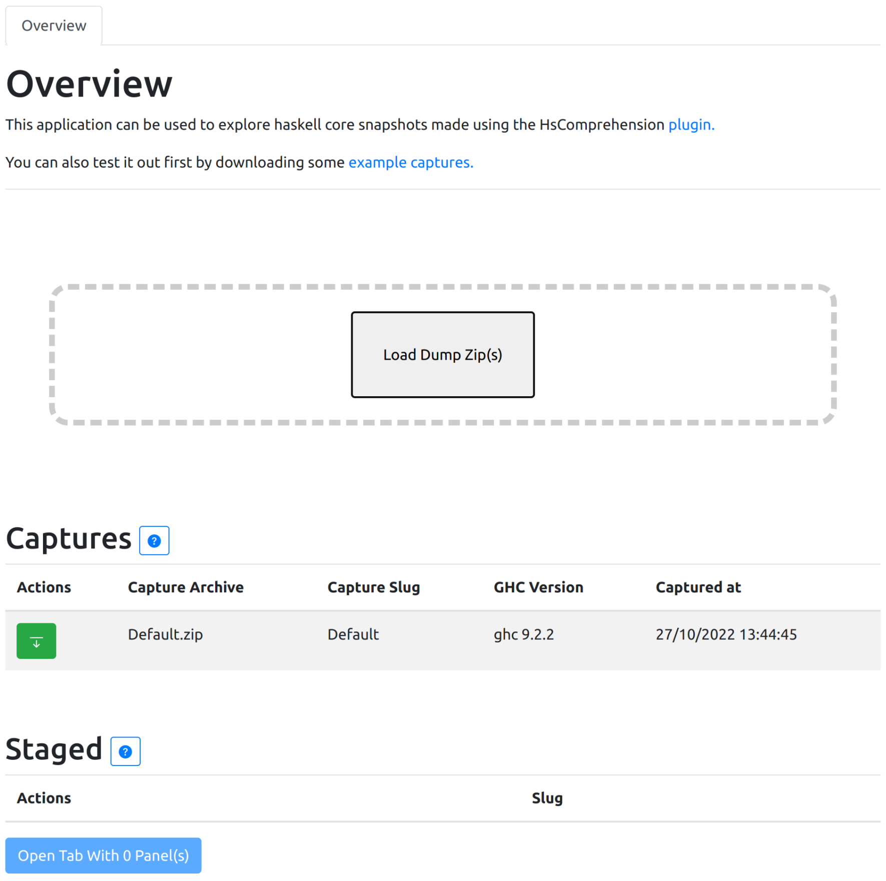
2. Creating a capture Because we only want to create a capture of this module, we can use an exported TemplateHaskell primitive that registers the plugin for the current module only by simply adding the `dumpThisModule` slice anywhere at the top level:

```
{-# LANGUAGE TemplateHaskell #-}
import HsSleuth.Plugin (dumpThisModule)

...

dumpThisModule
```

Following a successful build, we can bundle the generated artifacts to a zip archive by running:

```
$ cabal run hs-sleuth-zip
Attempting to archive dump files in ./dist-newstyle/coredump-Default
Archiving 23 files
Created /home/hugo/repos/hs-sleuth/test-project/dist-newstyle/Default.zip
```

3. Finding the root cause We navigate to core.hugopeters.me and upload the zip archive we just produced.



Overview

This application can be used to explore haskell core snapshots made using the HsComprehension plugin.

You can also test it out first by downloading some example captures.

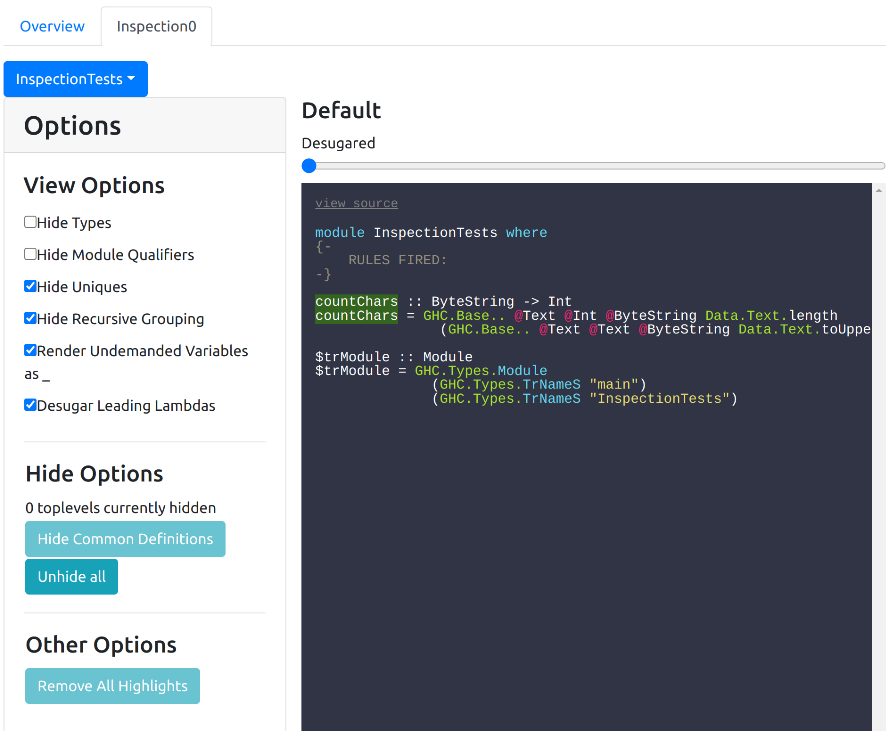
Captures

Actions	Capture Archive	Capture Slug	GHC Version	Captured at
↓	Default.zip	Default	ghc 9.2.2	27/10/2022 13:44:45

Staged

Actions	Slug

We then click the green arrow to reference the capture in the staging area. Here we could elect to stage more than one capture if we want to compare them. In this case we are only interested in the current situation, and so we just open a single panel tab with this single capture.



Options

View Options

- [ ] Hide Types
- [ ] Hide Module Qualifiers
- [x] Hide Uniques
- [x] Hide Recursive Grouping
- [x] Render Undemanded Variables as _
- [x] Desugar Leading Lambdas

Hide Options

0 toplevels currently hidden

Other Options

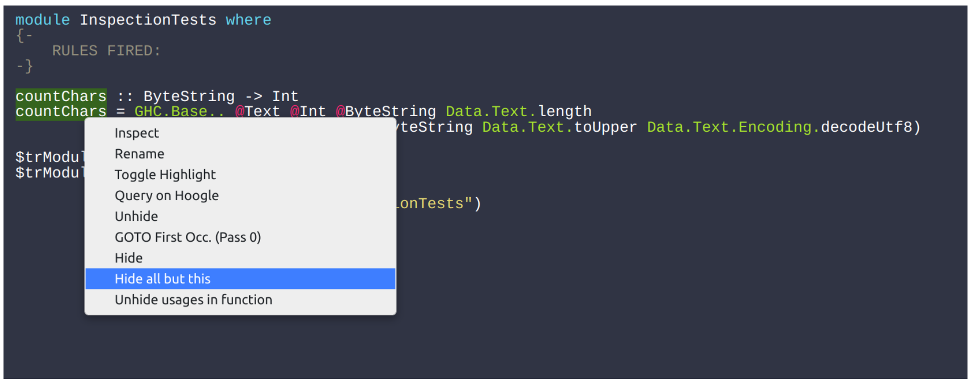
Default

Desugared

```
view source

module InspectionTests where
{-
    RULES FIRED:
-}

countChars :: ByteString -> Int
countChars = GHC.Base.. @Text @Int @ByteString Data.Text.length
                (GHC.Base.. @Text @Text @ByteString Data.Text.toUppe

$trModule :: Module
$trModule = GHC.Types.Module
               (GHC.Types.TrNameS "main")
               (GHC.Types.TrNameS "InspectionTests")
```

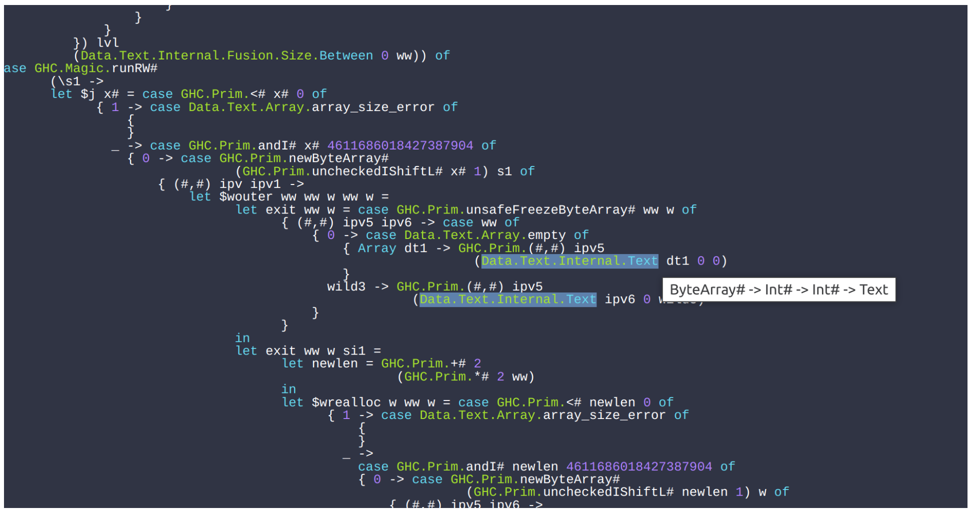
On the left, we are immediately presented with a number of viewing options. To the right we can see the rendered Core, under influence of the view options. Above it, there is a slider indicating that we are looking at the Core in the desugared stage (so without any transformations yet applied). Scrolling this slider will reveal the intermediate Core ASTs that were produced by the compiler. Whenever rewrite rules are fired, they are included as comments at the top of the module.

As you can see, the desugaring process has produced another top-level definition, namely `$trModule`. Since we do not care for anything but our `countChars` function at this time, we can elect to filter out all other definitions, including those that will be generated in the future:

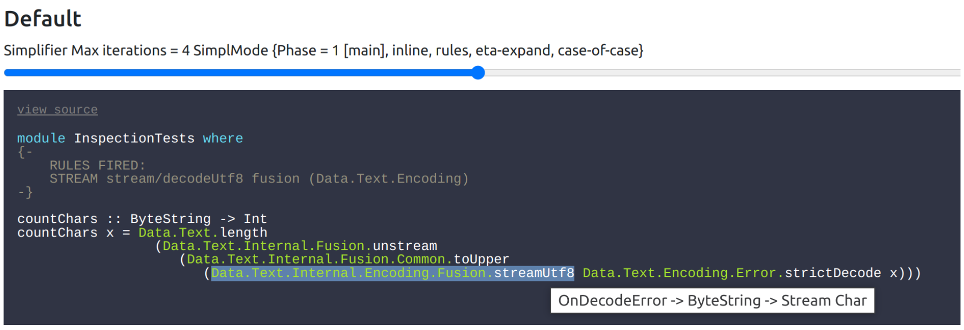


```
module InspectionTests where
{-
    RULES FIRED:
-}

countChars :: ByteString -> Int
countChars = GHC.Base.. @Text @Int @ByteString Data.Text.length
                                      teString Data.Text.toUpper Data.Text.Encoding.decodeUtf8)

$trModu
$trModu

                                      onTests")
```

If we then scroll all the way to the end, we get the same final Core AST as we saw in the error message. Granted, we now have syntax highlighting and a slightly more readable representation, but it is still unwieldy. Using a basic string search we can find the needle in the haystack:


```
                 }
             }
         }) lvl
         (Data.Text.Internal.Fusion.Size.Between 0 ww)) of
ase GHC.Magic.runRW#
      (\s1 ->
      let $j x# = case GHC.Prim.<# x# 0 of
            { 1 -> case Data.Text.Array.array_size_error of
                {
                }
              _ -> case GHC.Prim.andI# x# 4611686018427387904 of
                { 0 -> case GHC.Prim.newByteArray#
                              (GHC.Prim.uncheckedIShiftL# x# 1) s1 of
                    { (#,#) ipv ipv1 ->
                        let $wouter ww ww w ww w =
                              let exit ww w = case GHC.Prim.unsafeFreezeByteArray# ww w of
                                    { (#,#) ipv5 ipv6 -> case ww of
                                        { 0 -> case Data.Text.Array.empty of
                                            { Array dt1 -> GHC.Prim.(#,#) ipv5
                                                             (Data.Text.Internal.Text dt1 0 0)
                                            }
                                          wild3 -> GHC.Prim.(#,#) ipv5
                                                       (Data.Text.Internal.Text ipv6 0 w
                                        }
                                    }
                              in
                              let exit ww w si1 =
                                    let newlen = GHC.Prim.+# 2
                                                   (GHC.Prim.*# 2 ww)
                                    in
                                    let $wrealloc w ww w = case GHC.Prim.<# newlen 0 of
                                          { 1 -> case Data.Text.Array.array_size_error of
                                              {
                                              }
                                            _ ->
                                              case GHC.Prim.andI# newlen 4611686018427387904 of
                                              { 0 -> case GHC.Prim.newByteArray#
                                                            (GHC.Prim.uncheckedIShiftL# newlen 1) w of
                                                  { (#,#) ipv5 ipv6 ->
```

But we don't really care about finding the needle, but more so how it got it there. Using the scroll bar, we can go back in time to a moment before everything was inlined. Specifically, we can go back to the first moment where no Text constructor existed yet:


Default

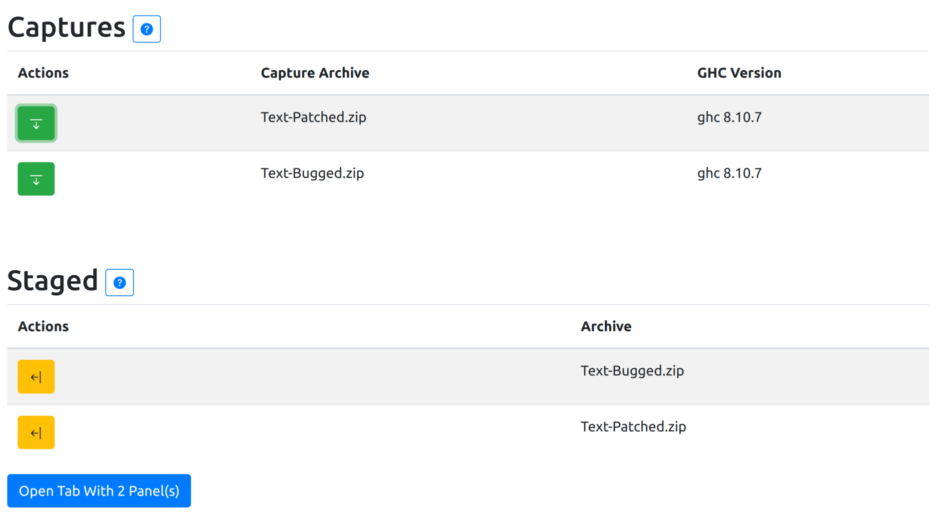
Simplifier Max iterations = 4 SimplMode {Phase = 1 [main], inline, rules, eta-expand, case-of-case}

```
view source

module InspectionTests where
{-
    RULES FIRED:
    STREAM stream/decodeUtf8 fusion (Data.Text.Encoding)
-}

countChars :: ByteString -> Int
countChars x = Data.Text.length
                 (Data.Text.Internal.Fusion.unstream
                    (Data.Text.Internal.Fusion.Common.toUpper
                       (Data.Text.Internal.Encoding.Fusion.streamUtf8 Data.Text.Encoding.Error.strictDecode x)))
```

We find a far more manageable definition of `countChars` that has partially been transformed to operate on streams (`Stream Char`). This is a concept to facilitate fusion based on the work of D. Couts et al. [5], we will discuss its theory more in depth in the next section. For now, it is only important to realise that instead of embedding the incoming `ByteString` in a `Text` value, we are converting to a `Stream Char` first before `unstream` converts to an actual `Text`. This final conversion is not necessary, because using `length` function for type `Stream Char` directly would suffice.

So we can conclude that the `text`'s fusion machinery did not produce the optimal result because it is conceivable to find the length of a stream directly using some alternative `length :: Stream Char -> Int` function.

4. Back to the future Luckily, we were reliving someone else's experience, and we have the luxury of seeing how the situation unfolded. So, what we can do is make another capture with the slightly more recent `1.2.4.0` version of the library. `inspection-testing` already told us that this newer version does not produce a `Text` constructor. We can compare the two captures to explore when they diverge.



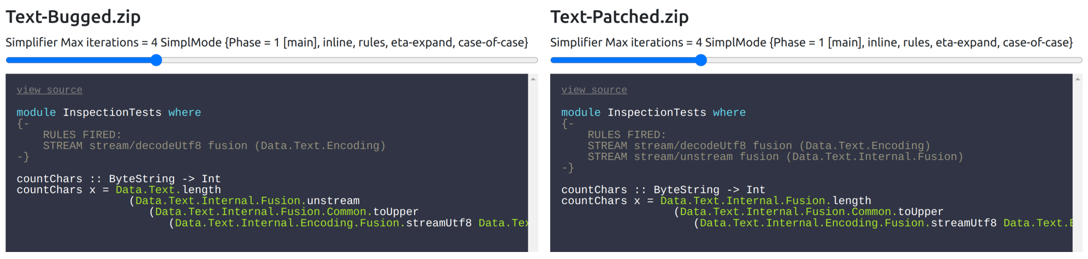
Captures

Actions	Capture Archive	GHC Version
	Text-Patched.zip	ghc 8.10.7
	Text-Bugged.zip	ghc 8.10.7

Staged

Actions	Archive
	Text-Bugged.zip
	Text-Patched.zip

Because we now have more than 1 capture open at the same time we can use the *Hide common definitions* feature to find the first moment where the two captures diverge. This happens to be at phase 1 of the simplifier pass:



Text-Bugged.zip

Simplifier Max iterations = 4 SimplMode {Phase = 1 [main], inline, rules, eta-expand, case-of-case}

```
view source

module InspectionTests where
{-
    RULES FIRED:
    STREAM stream/decodeUtf8 fusion (Data.Text.Encoding)
-}

countChars :: ByteString -> Int
countChars x = Data.Text.length
                 (Data.Text.Internal.Fusion.unstream
                    (Data.Text.Internal.Fusion.Common.toUpper
                       (Data.Text.Internal.Encoding.Fusion.streamUtf8 Data.Tex
```

Text-Patched.zip

Simplifier Max iterations = 4 SimplMode {Phase = 1 [main], inline, rules, eta-expand, case-of-case}

```
view source

module InspectionTests where
{-
    RULES FIRED:
    STREAM stream/decodeUtf8 fusion (Data.Text.Encoding)
    STREAM stream/unstream fusion (Data.Text.Internal.Fusion)
-}

countChars :: ByteString -> Int
countChars x = Data.Text.Internal.Fusion.length
                 (Data.Text.Internal.Fusion.Common.toUpper
                    (Data.Text.Internal.Encoding.Fusion.streamUtf8 Data.Text.E
```

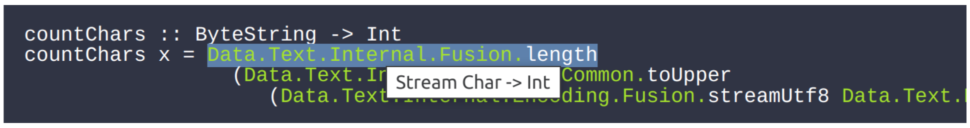
For clarity, let us extract the text from both panels and compare them:

```
1   {-
2       Text-Bugged.zip
3       RULES FIRED:
4       STREAM stream/decodeUtf8 fusion (Data.Text.Encoding)
5   -}
6
7   countChars :: ByteString -> Int
8   countChars x = Data.Text.length
9                    (Data.Text.Internal.Fusion.unstream
10                      (Data.Text.Internal.Fusion.Common.toUpper
11                         (Data.Text.Internal.Encoding.Fusion.streamUtf8 Data.Text.Encoding.Error.strictDecode x)))
12
13  -------------------------------------------------------------------
14  {-
15      Text-Patched.zip
16      RULES FIRED:
17      STREAM stream/decodeUtf8 fusion (Data.Text.Encoding)
18      STREAM stream/unstream fusion (Data.Text.Internal.Fusion)
19  -}
20
21  countChars :: ByteString -> Int
22  countChars x = Data.Text.Internal.Fusion.length
23                   (Data.Text.Internal.Fusion.Common.toUpper
24                      (Data.Text.Internal.Encoding.Fusion.streamUtf8 Data.Text.Encoding.Error.strictDecode x))
```

The most notable difference is the extra rewrite rule that was fired in the patched version (line 18). Unfortunately, we have yet to discover a way retrieve definition of fired rewrite rules from GHC. As such, they are not available in `hs-sleuth` itself. But given that we know its name and originating module, we can find it in the source of `text` without too much effort:

```
{-# RULES "STREAM stream/unstream fusion" forall s. stream (unstream s) = s #-}
```

From this we learn that at some point there was a stream/unstream pair to remove. Another difference is the module from which the `length` function is imported (`Data.Text.Internal.Fusion.length` over `Data.Text.length`). Like we predicted earlier, the patched version uses a variant that operates directly on streams:



```
countChars :: ByteString -> Int
countChars x = Data.Text.Internal.Fusion.length
                 (Data.Text.Ir                Common.toUpper
                    (Data.Text.Internal.Encoding.Fusion.streamUtf8 Data.Text.
```

Given that in the previous pass the captures were identical, and since no rewrite rule fired regarding `length`, we can speculate that the difference is caused by inlining `length`. If we collect the definition of `length` from both versions of the text library we get:

```
-- Text-Bugged.zip
length :: Text -> Int
length t = S.length (stream t)
{-# INLINE [0] length #-}

-------------------------------------------

-- Text-Patched.zip
length :: Text -> Int
length t = S.length (stream t)
{-# INLINE [1] length #-}
```

The only difference is the phase annotation of the `INLINE` pragma. The maintainers somehow decided that it was better to inline `length` one simplifier phase earlier (remember, phase 1 comes **before** phase 0). And they turned out to be right, because inlining earlier uncovered the opportunity for the *stream/unstream* rule to fire and remove the need to allocate an intermediate `Text` value; Another exemplary manifestation of the Cascade Effect.

5. Epilogue: Brittleness of implicit fusion At or around the same time as Breitner identified the failed fusion case [3], Andrew Lelechenko had discovered a problem involving the `tail` function [14] under fusion. `tail` just needs to drop the first character. Despite needing to check whether to skip 1 or 2 bytes because of the UTF-16 encoding, this can be done in $O(1)$ time and memory. Obviously this property should still hold when applying `tail` twice in row. As it turns out, it does not. The following steps occur:

```
tail . tail
-- { inline to fusion variant }
unstream . S.tail . stream . unstream . S.tail . stream
-- { apply 'stream . unstream = id' }
unstream . S.tail . S.tail . stream
```

By constructing a stream we have become committed to traversing the entire structure (in `unstream`) where it was not needed at first, yielding an $O(n)$ time and memory version after "optimisation". This is different from the situation in `countChars`, where UTF-16 already dictated $O(n)$ runtime.

The ending to this story is quite simply that implicit fusion was disabled entirely [14] for similar functions. Frequent `text` contributor, Oleg Grenrus, remarked the following on the proposal to remove them:

"I think this is the right thing to do. Implicit fusion is unpredictable, and you explain, doesn't even work in simple cases."

Instead, users can now opt in by using the stream variant of such functions explicitly. This is a tragic example of how optimisation can be unpredictable, and by extent, how people would favour predictability over automatic performance transformations that risk making the program slower in some cases.

4.2 Lists vs Streams

Besides short-cut fusion [12], there has been research into other fusion frameworks. One such framework is *stream fusion* [5]. Amusingly, the GHC wiki page about optimisation still mentions that stream fusion will be the default in the future [8]. It is our understanding that this is no longer the intention, as join points reveal fusion opportunities that are not possible with the stream fusion framework [16].

Regardless, stream fusion makes for an interesting case study to explore its effects throughout the transformation pipeline. The original implementation accompanied by the paper is still available as an archive on Hackage ([24]) and can reasonably easy be made to compile under modern GHC versions. Our goal here is not to give a detailed explanation of the framework based on the paper itself, but rather show how we can use `hs-sleuth` to gain an intuition about how code that we did not create ourselves interacts with the compiler and explain why.

The big idea underlying stream fusion is actually not that different from the *build/foldr* building blocks. Instead of creating an existentially typed `build` functions, an existentially typed datatype is used. Inside this datatype is a continuation function that produces the next element of the stream. This next element can also be `Skip` to indicate that the element was dropped, or `Done` to indicate that the stream is finished:

```haskell
data Stream a =
  forall s. Stream (s -> Step a s) s

data Step s a
  = Yield a s
  | Skip s
  | Done
```

To gain a better intuition about how this type operates, it helps to consider the `stream` and `unstream` functions which facilitate back and forth conversion to canonical lists:

```haskell
stream :: [a] -> Stream a
stream xs = Stream next xs
  -- the stream keeps track of the remaining list and peels of the
  -- head at each step
  where next []     = Done
        next (x:xs) = Yield x xs

unstream :: Stream a -> [a]
unstream (Stream next s) = go s
  -- effectively 'go' chains 'next' unto itself until a 'Done' is reached
  where go s = case next s of
          Done       -> []
          Skip s     -> go s
          Yield x s -> x : go s
```

One could describe a stream as a stateful generator function. The type of this state is hidden behind the existentially quantified type `s`. [5] This is actually quite similar to the concept of an iterator in imperative languages.

Why would we want to introduce this rather complex type? The answer is that it allows us to write functions over streams that are non-recursive and therefore be fused by already existing optimisations like inlining and the worker/wrapper transformation. But before we get to that point, let us first redefine some common functions like `map` and `filter` in terms of streams:

```haskell
-- In both these functions the state is itself a stream
map_s :: (a -> b) -> Stream a -> Stream b
map_s f = Stream next
  -- replaces the 'next' function with one that applies 'f' to any 'Yield' and propagates itself
  where next (Stream next s) = case next s of
          Done      -> Done
          Skip s    -> Skip (Stream next s)
          Yield x s  -> Yield (f x) (Stream next s)

filter_s :: (a -> Bool) -> Stream a -> Stream a
filter_s p = Stream next'
  -- replaces the 'next' function witih one that maps 'Yield' to 'Skip' if 'p' holds and propagates
  where next' (Stream next s) = case next s of
          Done      -> Done
          Skip s    -> Skip (Stream next s)
          Yield x s  -> if p x then Yield x (Stream next s) else Skip (Stream next s)
```

These variants can be used to create drop-in replacements for the canonical built-in functions.

```haskell
map :: (a -> b) -> [a] -> [b]
map f = unstream . map_s f . stream

filter :: (a -> Bool) -> [a] -> [a]
filter p = unstream . filter_s p . stream
```

It is important to realise that these definitions are not subject to existing rewrite rules for the eponymous functions from `Data.List`. With our framework in place, we can now involve a recurring example, `halves`:

```haskell
halves :: [Int] -> [Int]
halves = map (*2) . filter even
```

We already know how GHC's short-cut fusion will treat this function, so let us assume we are now using the stream fusion variants instead and that we have convinced GHC to inline these definitions:

```haskell
halves :: [Int] -> [Int]
halves = unstream . map_s (`div` 2) . stream . unstream . filter_s even . stream
```

Thus far, this doesn't appear to be a good idea at all: we are wasting a lot of work converting between streams and lists. However, there is a rather obvious avenue for elimination given the equivalence `stream . unstream = id`:

```haskell
{-# RULES "stream/unstream" forall s. stream (unstream s) = s #-}

--after firing
halves :: [Int] -> [Int]
halves = unstream . map_s (`div` 2) . filter_s even . stream
```

Of course this only solved a problem we introduced ourselves in the first place, but this is the moment the big idea kicks in: because `map_s` and `filter_s` are not defined recursively, they are subject to inlining. Using our tool we can observe what happens in practice during the transformation of `halves` without any further assumptions. From the very start:

```
--[0] Desugared
halves :: [Int] -> [Int]
halves xs = Data.List.Stream.map
              (
              let div_int = GHC.Real.div GHC.Real.$fIntegralInt
              in
              let two = GHC.Types.I# 2
              in \v -> div_int v two)
              (Data.List.Stream.filter
                 (GHC.Real.even GHC.Real.$fIntegralInt) xs)
```

We have renamed the let bound variables within `hs-sleuth` to something descriptive, making the code more readible for the passes to come. We continue this practice for the upcoming changes where warranted. Then the `InitialPhase` of the simplifier is invoked:

```
--[1] Simplifier [InitialPhase]
{-
    RULES FIRED:
    Class op div (BUILTIN)
    filter -> fusible (Data.List.Stream)
    map -> fusible (Data.List.Stream)
    STREAM stream/unstream fusion (Data.Stream)
-}

halves :: [Int] -> [Int]
halves xs = Data.Stream.unstream
              (Data.Stream.map
                 (
                 let two = GHC.Types.I# 2
                 in \v -> GHC.Real.$fIntegralInt_$cdiv v two)
                 (Data.Stream.filter
                    (GHC.Real.even GHC.Real.$fIntegralInt)
                    (Data.Stream.stream xs)))

```

`map` and `filter` have now been rewritten to their stream variants (by the two *fusible* rules) and right after that the complementing `stream/unstream` pair has been removed by the *stream/unstream* rule. The version we have now is effectively the same as our original hypothesis before we started exploring the real world situation. Moving on:

```
--[2] Specialise
$dReal :: Real Int
$dReal = GHC.Real.$p1Integral GHC.Real.$fIntegralInt

$dEq :: Ord Int
$dEq = GHC.Real.$p2Real $dReal

$dEq1 :: Eq Int
$dEq1 = GHC.Classes.$p1Ord $dEq

$dNum :: Num Int
$dNum = GHC.Real.$p1Real $dReal

$seven :: Int -> Bool
$seven n = GHC.Classes.== $dEq1
             (GHC.Real.rem GHC.Real.$fIntegralInt n
                (GHC.Num.fromInteger $dNum 2))
             (GHC.Num.fromInteger $dNum 0)

halves :: [Int] -> [Int]
halves xs = Data.Stream.unstream
              (Data.Stream.map
                 (
                 let two = GHC.Types.I# 2
                 in \v -> GHC.Real.$fIntegralInt_$cdiv v two)
                 (Data.Stream.filter
                    (GHC.Real.even GHC.Real.$fIntegralInt)
                    (Data.Stream.stream xs)))
```

The specialise phase has generated a couple – currently unused – auxiliary functions like `seven` ('s' as in 'specialised even'). This definition will become relevant later as the resolver of finding a specific `Num` instance. We can expect to see this functions being used in the near future. Coming up is the transformation that floats definitions up.

```
-- [3] Float out

-- ... (omitted)

two :: Int
two = GHC.Types.I# 2

div2 :: Int -> Int
div2 v = GHC.Real.$fIntegralInt_$cdiv v two

even :: Int -> Bool
even = GHC.Real.even GHC.Real.$fIntegralInt

halves :: [Int] -> [Int]
halves xs = Data.Stream.unstream
              (Data.Stream.map div2
                 (Data.Stream.filter even
                    (Data.Stream.stream xs)))
```

The float out phase has, unsurprisingly, floated out some expressions to fresh top-level binds. In itself this has not achieved much, but it is generally a curveball to the simplifier:

```
-- [4] Simplifier [Phase=2]
{-
    RULES FIRED:
    Class op £p1Integral (BUILTIN)
    Class op £p2Real (BUILTIN)
    Class op £p1Ord (BUILTIN)
    Class op £p1Real (BUILTIN)
    Class op fromInteger (BUILTIN)
    Integer -> Int# (wrap) (BUILTIN)
    Class op fromInteger (BUILTIN)
    Integer -> Int# (wrap) (BUILTIN)
    Class op == (BUILTIN)
    Class op rem (BUILTIN)
    divInt# (BUILTIN)
    SPEC/Streaming even @Int (Streaming)
-}

zero :: Int
zero = GHC.Types.I# 0

$seven :: Int -> Bool
$seven n = case n of
  { I# ipv -> GHC.Classes.eqInt
                (GHC.Types.I#
                   (GHC.Prim.remInt# ipv 2)) zero
  }

div2 :: Int -> Int
div2 v = case v of
  { I# ww1 -> GHC.Types.I#
                (GHC.Prim.uncheckedIShiftRA# ww1 1)
  }

halves :: [Int] -> [Int]
halves xs = Data.Stream.unstream
              (Data.Stream.map div2
                 (Data.Stream.filter $seven
                    (Data.Stream.stream xs)))
```

Simplified indeed, the specialised functions have been adopted and sometimes inlined. However, any signs of fusion can not be found yet however. Let us wind the clock forward another simplifier invocation.

```
--[5] Simplifier [Phase=1]
{-
    RULES FIRED:
    ==# (BUILTIN)
    tagToEnum# (BUILTIN)
    tagToEnum# (BUILTIN)
-}

$seven :: Int -> Bool
$seven n = case n of
  { I# ipv -> case GHC.Prim.remInt# ipv 2 of
      { 0 -> GHC.Types.True
        _ -> GHC.Types.False
      }
  }

div2 :: Int -> Int
div2 v = case v of
  { I# ww1 -> GHC.Types.I#
                (GHC.Prim.uncheckedIShiftRA# ww1 1)
  }

halves :: [Int] -> [Int]
halves xs = Data.Stream.unstream
              (Data.Stream.map div2
                 (Data.Stream.filter $seven
                    (Data.Stream.stream xs)))
```

Again still nothing exciting, only the `zero` constant has been inlined. Upcoming is the last configurable phase of the simplifier:

```
--[6] Simplifier [Phase=0]
{-
    RULES FIRED:
    Class op expose (BUILTIN)
    Class op expose (BUILTIN)
-}

$seven :: Int -> Bool
$seven n = case n of
  { I# ipv -> case GHC.Prim.remInt# ipv 2 of
      { 0 -> GHC.Types.True
        _ -> GHC.Types.False
      }
  }

halves :: [Int] -> [Int]
halves xs =
  let unfold_unstream s1 = case s1 of
        { L ipv -> case ipv of
            { : x xs -> case x of
                { I# ipv -> case GHC.Prim.remInt# ipv 2 of
                    { 0 -> GHC.Types.:
                             (GHC.Types.I#
                                (GHC.Prim.uncheckedIShiftRA# ipv 1))
                             (unfold_unstream
                                (Data.Stream.L xs))
                      _ -> unfold_unstream
                             (Data.Stream.L xs)
                    }
                }
              [] -> GHC.Types.[]
            }
        }
  in unfold_unstream
       (Data.Stream.L xs)
```

Now something quite drastic has changed, and it is not clear what exactly has happened. Let us first investigate what the rewrite rule that just fired twice contributed to these changes. `Class op {f}` implies that the function `f` as part of some class constraint was specialised. If we scour the source we find the following typeclass:

```haskell
class Unlifted a where

  -- | This expose function needs to be called in folds/loops that consume
  -- streams to expose the structure of the stream state to the simplifier
  -- In particular, to SpecConstr.
  --
  expose :: a -> b -> b
  expose = seq

  -- | This makes GHC's optimiser happier; it sometimes produces really bad
  -- code for single-method dictionaries
  --
  unlifted_dummy :: a
  unlifted_dummy = error "unlifted_dummy"
```

Supposedly this class is implemented for `Stream a`, as those are terms that are affected, and indeed we find:

```haskell
instance Unlifted (Stream a) where
  expose (Stream next s0) s = seq next (seq s0 s)
  {-# INLINE expose #-}
```

From this we can speculate that this typeclass exists to ensure that expressions are evaluated to WHNF (by definition of `seq`), which is a requirement to ensure that other optimisations fire.

But we have not seen any call-site for `expose` in the code, so apparently it appeared somehow **and** was specialised during this transformation. The most logical explanation for new function calls appearing is that they were the result of an inlining. Our likely perpetrator is one of the four stream functions:

- `stream`
- `map`
- `filter`
- `unstream`

By inspecting each definition it turns out that of those suspects only `unstream` contains calls to `expose`:

```haskell
unstream :: Stream a -> [a]
unstream (Stream next s0) = unfold_unstream s0
  where
    unfold_unstream !s = case next s of
      Done       -> []
      Skip    s' -> expose s' $     unfold_unstream s'
      Yield x s' -> expose s' $ x : unfold_unstream s'
{-# INLINE [0] unstream #-}
```

Our expectations are further confirmed by the inline pragma which says to inline only at phase 0, which is exactly what we think happened. The same inline pragma is present on all of our other suspects as well, so we are looking a quadruple inlining event. The calls to `seq` that we observed previously are desugared as case expressions, as they are the primitive operation in Core that evaluates to WHNF.

Not all questions are answered however, as we don't know what the `L` constructor does. Again, looking that up gives the following source:

```haskell
-- | Boxes for user's state. This is the gateway for user's types into unlifted
-- stream states. The L is always safe since it's lifted/lazy, exposing/seqing
-- it does nothing.
-- S is unlifted and so is only suitable for users states that we know we can
-- be strict in. This requires attention and auditing.
--
data    L a = L a  -- lazy / lifted
newtype S a = S a  -- strict / unlifted
```

It seems that `L` is just a box around a type that provides a barrier for WHNF evaluation. We can find it being used in the definition of `stream`:

```haskell
stream :: [a] -> Stream a
stream xs0 = Stream next (L xs0)
  where
    {-# INLINE next #-}
    next (L [])      = Done
    next (L (x:xs)) = Yield x (L xs)
    {-# INLINE [0] stream #-}
```

In essence, it just cancels the effect of `seq`. But if we are careful then in some situation we might improve performance by using the strict version `S` instead.

So that aside, have we achieved fusion? It takes some effort to realise that, (1) through the use of `expose` in the definition of `unstream`, and (2) by the direct use of the incoming `next` function in the definition of the following next function, our resulting list has a head element that is defined as the composition of `next` function of `map` and `filter`. This is a very contrived way to say that we have indeed achieved fusion. Another way to look at is that the `(:)` constructor is called once per element.

But we are still left with some noise, notably the wrapping/unwrapping of values in now redundant `L` constructors. That description should invoke a sense of familiarity, as the reader should know by know that an upcoming transformation is that of the *worker/wrapper*! If we follow the rest of the pipeline in chronological order:

```haskell
-- [7] Float inwards
-- no changes

-- [8] Called arity analysis
-- no structural changes (IdInfos might be updated)

-- [9] Simplifier [Phase = Final]
-- no changes

-- [10] Demand analysis
-- no structural changes (IdInfos might be updated)

-- [11] Constructed Product Result analysis
-- no structural changes (IdInfos might be updated)

-- [12] Worker/wrapper binds
halves :: [Int] -> [Int]
halves xs =
  let $wunfold_unstream ww =
        let w = Data.Stream.L ww
        in
        let s1 = w
        in case s1 of
        { L ipv -> case ipv of
            { : x xs -> case x of
                { I# ipv -> case GHC.Prim.remInt# ipv 2 of
                    { 0 -> GHC.Types.:
                             (GHC.Types.I#
                                (GHC.Prim.uncheckedIShiftRA# ipv 1))
                             (unfold_unstream
                                (Data.Stream.L xs))
                      _ -> unfold_unstream
                             (Data.Stream.L xs)
                    }
                }
              [] -> GHC.Types.[]
            }
        }
      unfold_unstream w = case w of
        { L ww -> $wunfold_unstream ww
        }
  in unfold_unstream
       (Data.Stream.L xs)
```

From the `w` prefix in the name of `wunfold_unstream` we can derive that this function was generated by the worker/wrapper transformation. This specific piece of knowledge is not strictly necessary however since our tool can tell you for any function in which pass it was generated regardless of name.

If you look through the let bindings, it becomes apparent that we wrap an element `ww` in an `L` constructor, and then always evaluate it to WHNF in the case expression. This is a classic wasteful pattern that the simplifier is able to deal with:

```
-- [13] Simplifier [Phase = Final]
halves :: [Int] -> [Int]
halves xs =
  let $wunfold_unstream ww = case ww of
        { : x xs -> case x of
            { I# ipv -> case GHC.Prim.remInt# ipv 2 of
                { 0 -> GHC.Types.:
                        (GHC.Types.I#
                           (GHC.Prim.uncheckedIShiftRA# ipv 1))
                        ($wunfold_unstream xs)
                  _ -> $wunfold_unstream xs
                }
            }
          [] -> GHC.Types.[]
        }
  in $wunfold_unstream xs
```

With that final elision we have obtained a fully fused version of our `map`/`filter` composition with any auxiliary machinery like `stream` and `unstream` being simplified away.

So what we have shown is that it is perfectly feasible to create and retrofit an alternative fusion system in vanilla Haskell. However, it is also clear that the process of implementating it goes beyond translating the theory verbatim. Namely, we have seen how `Unlifted` class was necessary to ensure that GHC correctly handles lazy and strict situation, including the need to put an extra unused function in the typeclass to avoid some unexpected GHC behavior. All this tells us that the developers of the library have most certainly spent a large chunk of their time looking at Core printouts to identify these issues before there were able to fix it like they did.

Furthermore, it was with the help of our tool that we were able to explain – without direct consultation and within a reasonable timeframe – how stream fusion truly operates in a real world scenario. This means that our tool may support those trying to reproduce and verify existing research involving Haskell and the GHC compiler.

4.3 Erroneous program structure recovery in GHC fusion

As part of our initial experiments to validate `hs-sleuth`, we decided to attempt to reproduce the shot-cut fusion of `unlines` as presented in its introductory the paper [12]. This serendipitously led to the discovery of an important performance bug in GHC.

4.3.1 The problem described

Consider the implementation of unlines given in the paper:

```
unlines :: [String] -> String
unlines ls = concat (map (\l -> l ++ ['\n']) ls)
```

Using `hs-sleuth`, we can observe the following Core at the end of the entire optimisation pipeline:

```
cr_chr :: Char
cr_chr = C# '\n'

cr :: [Char]
cr = : cr_chr []

go1 :: [[Char]] -> [Char]
go1 ds = case ds of
  { : y ys -> ++
                (++ y cr)
                (go1 ys)
    [] -> []
  }

unlines :: [String] -> String
unlines ls = go1 ls
```

This definition is problematic because every line is traversed once to append a newline character and then again to append it to the rest of the line, while we already know that a single traversal is possible from the original paper. [5] To rule out any other factors we ran a benchmark on both this version of `unlines` and `Prelude.unlines` with the first 73 lines of lorem ipsum. This confirmed our suspicions:

```
benchmarking our_unlines
time                 127.9 μs   (125.0 μs .. 130.7 μs)
                     0.997 R²   (0.996 R² .. 0.998 R²)
mean                 126.6 μs   (124.6 μs .. 128.4 μs)
std dev              6.532 μs   (5.742 μs .. 7.504 μs)
variance introduced by outliers: 53% (severely inflated)

benchmarking prelude_unlines
time                 80.23 μs   (79.87 μs .. 80.53 μs)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 79.70 μs   (79.05 μs .. 80.13 μs)
std dev              1.858 μs   (1.088 μs .. 2.962 μs)
variance introduced by outliers: 20% (moderately inflated)
```

So it is clear that `Prelude` has defined `unlines` in a more efficient manner, but that does not take away the fact that any such expression as our version of `unlines` should reasonably be expected to fuse to something with a single input traversal (it was the example given the original paper after all [12]).

4.3.2 Investigating the problem

Our approach to investigating this issue is to scroll all the way back to the desugared stage and see if we can find a distinct reason why the program was not transformed to something more optimal. Given that the culprit is a secondary list traversal, it is safe to assume that it is indeed the fusion system, or at least an iteraction with it, that is at the root of this problem. Therefore, we decided to analyze the problem by seeking meaningful differences between the results of the transformation and the steps given in the paper. The tool reports that the following Core is desugared from the source:

```
unlines :: [String] -> String
unlines ls = concat $fFoldable[]
               (map
                  (\l -> ++ l
                           (build
                              (\c n -> c
                                         (C# '\n') n))) ls)
```

Here we can observe that the list literal `['\n']` is already represented as a list producer function, paving the way for future short-cut fusion rules applications. After all we can hypothesize that in the near future `map` and `++` will be rewritten to their respective *build/foldr* representation, making the fusion rule applicable. After the first transformation, which is the also first pass of the simplifier (Gentle), we get the following:

```haskell
{-
    RULES FIRED:
    Class op foldr (BUILTIN)
    ++ (GHC.Base)
    augment/build (GHC.Base)
    map (GHC.Base)
    fold/build (GHC.Base)
-}

unlines :: [String] -> String
unlines ls = build
               (\c n -> foldr
                          (mapFB
                             (\x y -> foldr c y x)
                             (\l -> build
                                      (\c n -> foldr c
                                                 (c
                                                    (C# '\n') n) l))) n ls)
```

We can see how concat has been specialized and also rewritten, as well as `++`. The result of `map` however, is a bit more peculiar; contrary to what the paper suggested at the time, we instead observe a call to some function `mapFB`. We keep this observation in mind and continue on. The first pass that makes any significant change is the float out pass, which floats expressions up to the highest level to expose potential optimisations like common subexpression elimination. Again we employ `hs-sleuth` to give these binidings meaningful names:

```haskell
cr_chr :: Char
cr_chr = C# '\n'

append_cr :: [Char] -> [Char]
append_cr l = build
                (\c n -> foldr c
                           (c cr_chr n) l)

unlines :: [String] -> String
unlines ls = build
               (\c n -> foldr
                          (mapFB
                             (\x y -> foldr c y x) append_cr) n ls)
```

In essence, nothing has significantly changed, let us continue. Simplifier phase 2 comes and goes without modifying the code. Simplifier phase 1 reduces to:

```
{-
    RULES FIRED:
    foldr/app (GHC.Base)
    foldr/app (GHC.Base)
-}

cr_chr :: Char
cr_chr = C# '\n'

append_cr :: [Char] -> [Char]
append_cr l = ++ l
                (: cr_chr [])

unlines :: [String] -> String
unlines ls = foldr
               (mapFB ++ append_cr) [] ls
```

The only way to again find fusible pairs now is if by inlining mapFB. We can easily find its definition by right-clicking the term and selecting the *Query on Hoogle* option. From there we can quickly get the source code:

```
-- Note eta expanded
mapFB ::  (elt -> lst -> lst) -> (a -> elt) -> a -> lst -> lst
{-# INLINE [0] mapFB #-} -- See Note [Inline FB functions] in GHC.List
mapFB c f = \x ys -> c (f x) ys
```

It seems that we may yet have a chance, simplifier phase 0 is the next pass and the annotation on `mapFB` specifically requests that we only inline the function in phase 0. However, we find the following after phase 0:

```
cr_chr :: Char
cr_chr = GHC.Types.C# '\n'

unlines :: [String] -> String
unlines ls =
  let go1 ds = case ds of
        { : y ys -> GHC.Base.++
                      (GHC.Base.++ y
                         (GHC.Types.: cr_chr GHC.Types.[]))
                      (go1 ys)
          [] -> GHC.Types.[]
        }
  in go1 ls
```

The version is nearly identical to the final Core produced by the entire pipeline, minus the inconsequential top-level binding introduced for constructing the newline singleton string. So have discovered what went wrong? Well, not directly, but we have seen something that differs from the paper, namely the `mapFB` function. We might have already seen its definition, but it is not entirely clear what it does. Luckily, GHC has well documented source code, and so we return to Hoogle to find the following note near the definition of `mapFB`:

```
{- Note [The rules for map]
~~~~~~~~~~~~~~~~~~~~~~~~~~~
The rules for map work like this.

* Up to (but not including) phase 1, we use the "map" rule to
  rewrite all saturated applications of map with its build/fold
  form, hoping for fusion to happen.

  In phase 1 and 0, we switch off that rule, inline build, and
  switch on the "mapList" rule, which rewrites the foldr/mapFB
  thing back into plain map.

  It's important that these two rules aren't both active at once
  (along with build's unfolding) else we'd get an infinite loop
  in the rules.  Hence the activation control below.

* This same pattern is followed by many other functions:
  e.g. append, filter, iterate, repeat, etc. in GHC.List

  See also Note [Inline FB functions] in GHC.List

* The "mapFB" rule optimises compositions of map

* The "mapFB/id" rule gets rid of 'map id' calls.
  You might think that (mapFB c id) will turn into c simply
  when mapFB is inlined; but before that happens the "mapList"
  rule turns
     (foldr (mapFB (:) id) [] a
  back into
     map id
  Which is not very clever.

* Any similarity to the Functor laws for [] is expected.
-}

{-# RULES
"map"       [~1] forall f xs.   map f xs                = build (\c n -> foldr (mapFB c f) n xs)
"mapList"   [1]  forall f.      foldr (mapFB (:) f) []  = map f
"mapFB"     forall c f g.       mapFB (mapFB c f) g     = mapFB c (f.g)
"mapFB/id"  forall c.           mapFB c (\x -> x)       = c
  #-}
```

This documentation does not give us any insight yet into the role of the `mapFB` function. Perhaps the note `[Inline FB functions]` can provide some more insight:

```
Note [Inline FB functions]
~~~~~~~~~~~~~~~~~~~~~~~~~~

After fusion rules successfully fire, we are usually left with one or more calls
to list-producing functions abstracted over cons and nil. Here we call them
FB functions because their names usually end with 'FB'. It's a good idea to
inline FB functions because:

* They are higher-order functions and therefore benefits from inlining.

* When the final consumer is a left fold, inlining the FB functions is the only
  way to make arity expansion to happen. See Note [Left fold via right fold].

For this reason we mark all FB functions INLINE [0]. The [0] phase-specifier
ensures that calls to FB functions can be written back to the original form
when no fusion happens.

Without these inline pragmas, the loop in perf/should_run/T13001 won't be
allocation-free. Also see Trac #13001.
```

Here we find a plausible answer to why `mapFB` exists: "*calls to FB functions can be written back to the original form when no fusion happens*". It is desirable to retain the structure of the original map if it is not fused away (we will go more in depth as to why in Section 4.3.4). `mapFB` enables just that, it exposes an initial opportunity to fuse but is reversible using the *mapList* rule if no fusion happens. This rule also gives us a lot of intuition how `mapFB` operates. Namely, given the cons function and an empty generator is equivalent to canonical `map`. Similarly, the rule *map* gives us information about how `map` is more generally related to `mapFB`. At this point we should also compare how this `foldr` based implementation differs from the one given in the paper:

```haskell
-- The definition from the short-cut fusion paper
map f xs = build (\ c n -> foldr (\a b -> c (f a) b) n xs)

-- The definition from GHC.Base
map f xs = build (\c n -> foldr (mapFB c f) n xs)
```

It does not take much convincing to see that the two definitions are syntactically equivalent after inlining mapFB. Remember however the pragma instructing the compiler to inline mapFB **only** from phase 0 onwards. This turns out to be too late and it is in the previous iteration of the simplifier (phase 1) where we overlooked something. Namely, the *foldr/app* rule fired twice. A grep in the GHC code base reveals its definition:

```
"foldr/app"     [1] forall ys. foldr (:) ys = \xs -> xs ++ ys
        -- Only activate this from phase 1, because that's
        -- when we disable the rule that expands (++) into foldr
```

This rule appears to reverse the expansion of `++`. This is corroborated by the reasoning behind activiting it in phase 1, otherwise it would form a circular rule with the expansion the and the simplifier would exhaust its iterations without finding a fixed point. However, because the reconstruction fired **before** the inlining of `mapFB`, we missed out on an important fusion opportunity.

4.3.3 Fixing the problem

To confirm our theory about `mapFB` clouding the fusion process, we can patch GHC in the following crude manner:

- Redefine `map` directly as `build (\ c n -> foldr (\a b -> c (f a) b) n xs)` and mark it always as inlinable.
- Remove the *map* rule, to prevent it firing instead of inlining.

- Remove the rule *mapList*, *mapFB*, and *mapFB/id*, they should never be applicable anymore and there functionality now handled by the short-cut fusion rule.

For completeness:

```
{-# INLINE map #-}
-- GHC.Base

-- previously:
map :: (a -> b) -> [a] -> [b]
map [] = []
map (x:xs) = f x : map f xs

-- now:
{-# INLINE map #-}
map :: (a -> b) -> [a] -> [b]
map f xs = build (\c n -> foldr (\x r -> c (f x) r) n xs)
```

Note that because previously map was defined recursively it would never have been inlined. In the new situation we inform GHC that we are actually very keen to inline this function. If we feed the same source code to this patched version of the base library again we observe the following steps:

```
==================== Simplifier [Gentle] ====================
unlines
  = \ls ->
      build
        (\ c n ->
           foldr
             (\x r -> foldr c (c (C# '\n'#) r) x)
             n
             ls)
```

GHC took our wishes to heart and inlined `map` right away.

```
==================== Simplifier [Phase = 1] ====================
cr_char = C# '\n'#

unlines
  = \ls ->
      foldr (\x r -> ++ x (cr_char : r)) [] ls
```

Already we seem to be in a better place because we see only one call to `++`. But things become clearer when foldr is inlined in the next simplifier phase:

```
==================== Simplifier [Phase = 0] ====================
cr_char = C# '\n'

unlines
  = \ls ->
      let go ds =
          case ds of
            []    -> []
            y:ys -> ++ y (cr_char : (go ys))
      in
      go ls
```

Although we still have a call to `++` its left-hand side does not grow at each recursive iteration anymore. Furthermore, the other call to `++` is now correctly replaced by a much cheaper **prepend** action to the recursive call instead. If run the same benchmark we observe that indeed our program is assembled to something significantly more efficient:

```
benchmarking my_unlines
time                 64.80 µs   (62.43 µs .. 66.47 µs)
                     0.996 R²   (0.995 R² .. 0.997 R²)
mean                 62.43 µs   (61.68 µs .. 63.40 µs)
std dev              2.779 µs   (2.062 µs .. 3.346 µs)
variance introduced by outliers: 48% (moderately inflated)

benchmarking prelude_unlines
time                 75.36 µs   (74.74 µs .. 75.80 µs)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 73.98 µs   (73.69 µs .. 74.35 µs)
std dev              1.102 µs   (886.1 ns .. 1.300 µs)
```

In fact, it seems we have managed to beat the performance of the prelude implementation by a small but real margin.

4.3.4 It's more complicated

Seemingly we have solved the problem by returning to the original theory of the paper. But obviously the GHC developers did not simply incorrectly implement the material. Functions like `mapFB` do have a place. We already saw a hint of this in note from the source code:

```
...ensures that calls to FB functions can be written
 back to the original form when no fusion happens.
```

`mapFB` provides us a tag that there used to be a call to `map` here. If in the end we did not manage to fuse, or if a `mapFB` is left over after the *mapFB* rules has combined consecutive instances, we can reconstruct the original call to `map`. This has two important befinits: First of all, it allows a more recognizable form of compiled Core that more closely resembles the original source code. Secondly and arguably more importantly, we save on code size by reusing the map function by not rewriting each list operation to a local, recursive `go` function.

Let us find a concrete example by comparing our patched version of the base library with the original one when compiling the very simple function:

```haskell
addTwo :: [Int] -> [Int]
addTwo = map (+1) . map (+1)
```

The Core produced looks this:

```haskell
-- Core from canonical GHC (total of 11 terms)
addTwo1 :: Int -> Int
addTwo1 x -> x+2

addTwo :: [Int] -> [Int]
addTwo xs -> map addTwo1 xs

-- Core from our patch (total of 23 terms)
addTwo :: [Int] -> [Int]
addTwo ds = case ds of
              [] -> []
              y : ys -> let xs = addTwo ys
                            x  = y+2
                        in x:xs
```

This increased number of terms is a significant regression in compiled code size. However, we should not expect performance characteristics of the assembled code to differ much at all, as they essentially describe the same computation. We verify this by running a benchmark for `addTwo [0..10000]`:

```
benchmarking addTwo_baseline
time                 79.70 μs   (79.61 μs .. 79.80 μs)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 79.71 μs   (79.67 μs .. 79.78 μs)
std dev              181.6 ns   (100.2 ns .. 336.0 ns)


benchmarking addTwo_patched
time                 79.17 μs   (78.80 μs .. 79.52 μs)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 79.74 μs   (79.40 μs .. 80.10 μs)
std dev              1.181 μs   (981.2 ns .. 1.436 μs)
```

Despite the regression in compiled code size, the argument could be made this trade off is worth it if in fact we can optimize more programs. It should also be mentioned that our patch is very crude and does not take into account the subtle interations with other existing rewrite rules; It is merelyl a confirmation of the problem. It might be possible to find a more nuanced solution that still uses `mapFB` but ensures it inlines in time when it can be fused, but is still rewritten to `map` otherwise. Finding the ideal solution is not entirely trivial and an excellent topic for future work.

We have opened issue #22361 on the GHC bug tracker – as well as a merge request with the proposed changes therein – to spark a public conversation and collect feedback on the impact of the change. It should be noted that comments by both Sebastian Graf and Simon Peyton Jones were instrumental in helping us fully understand the problem and the potential solution.

4.4 Comparisons with existing approaches

As we will discuss in detail in Section 5.1, we were unable to use `hs-sleuth` when working with unreleased versions of GHC, something we had to do repeatedly during Section 4.3. As a silver lining, this presented an excellent opportunity to compare `hs-sleuth` with the canonical existing approach: GHC flags.

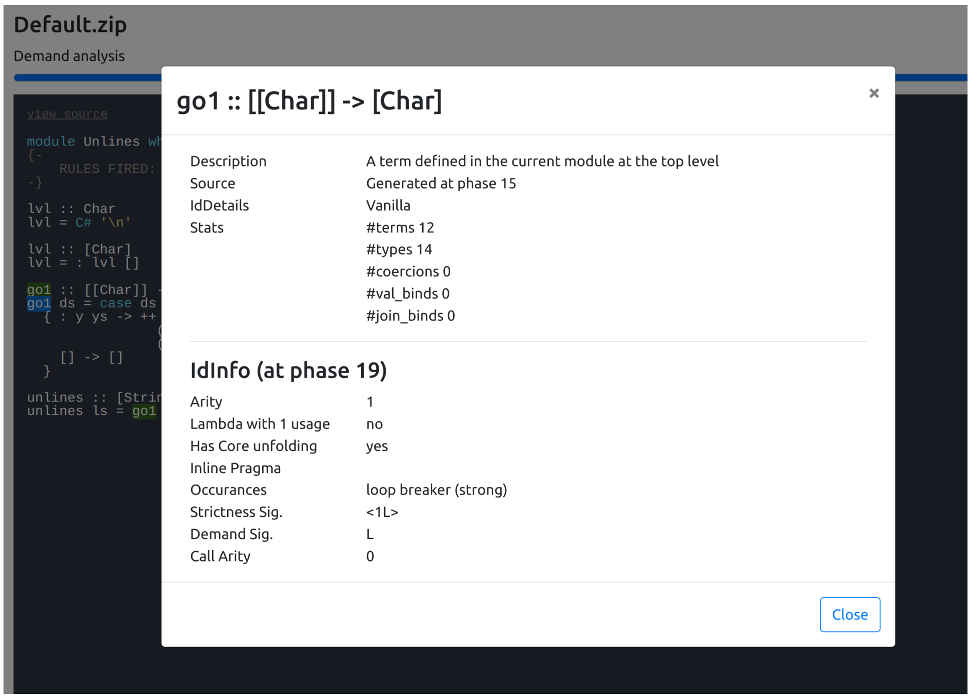
An important milestone for us was that when using `-dsuppress-all`, the readability of the Core expressions is still not on par with our tool. For clarity, we have manually edited the output of GHC to remove any irrelevant information. The original compiled Core definition of `unlines` looked like this:

```
-- RHS size: {terms: 2, types: 0, coercions: 0, joins: 0/0}
lvl_rVO :: GHC.Types.Char
[GblId, Unf=OtherCon []]
lvl_rVO = GHC.Types.C# '\n'#

-- RHS size: {terms: 3, types: 2, coercions: 0, joins: 0/0}
lvl1_rYZ :: [GHC.Types.Char]
[GblId, Unf=OtherCon []]
lvl1_rYZ
  = GHC.Types.:
      @GHC.Types.Char lvl_rVO (GHC.Types.[] @GHC.Types.Char)

Rec {
-- RHS size: {terms: 17, types: 18, coercions: 0, joins: 0/1}
Unlines.unlines_go1 [Occ=LoopBreaker]
  :: [[GHC.Types.Char]] -> [GHC.Types.Char]
[GblId, Arity=1, Str=<1L>, Unf=OtherCon []]
Unlines.unlines_go1
  = \ (ds_s11T [Occ=Once1!] :: [[GHC.Types.Char]]) ->
      case ds_s11T of {
        [] -> GHC.Types.[] @GHC.Types.Char;
        : y_s11V [Occ=Once1] ys_s11W [Occ=Once1] ->
          let {
            sat_s11Y [Occ=Once1, Dmd=ML] :: [GHC.Types.Char]
            [LclId]
            sat_s11Y = Unlines.unlines_go1 ys_s11W } in
          case GHC.Base.++ @GHC.Types.Char y_s11V lvl1_rYZ
          of sat_s11X [Occ=Once1]
          { __DEFAULT ->
          GHC.Base.++ @GHC.Types.Char sat_s11X sat_s11Y
          }
      }
end Rec }

-- RHS size: {terms: 3, types: 2, coercions: 0, joins: 0/0}
Unlines.unlines :: [GHC.Base.String] -> GHC.Base.String
[GblId, Arity=1, Str=<1L>, Unf=OtherCon []]
Unlines.unlines
  = \ (ls_s11Z [Occ=Once1] :: [GHC.Base.String]) ->
      Unlines.unlines_go1 ls_s11Z
```

As opposed to what `hs-sleuth` produces without any further manual effort such as renaming variables:

```
lvl :: Char
lvl = C# '\n'

lvl :: [Char]
lvl = : lvl []

go1 :: [[Char]] -> [Char]
go1 ds = case ds of
  { : y ys -> ++
                (++ y lvl)
                (go1 ys)
    [] -> []
  }

unlines :: [String] -> String
unlines ls = go1 ls
```

We believe it goes without saying that our representation is faster to read and comprehend. Of course this is not entirely fair because the GHC output contains for information that just happens to be uninteresting during many of our experiments. That said, this information is still available upon request by double-clicking on any term to reveal all the embedded information:



Default.zip

Demand analysis

```
view source

module Unlines wh
{-
    RULES FIRED:
-}

lvl :: Char
lvl = C# '\n'

lvl :: [Char]
lvl = : lvl []

go1 :: [[Char]] -
go1 ds = case ds
  { : y ys -> ++
                (
                (
    [] -> []
  }

unlines :: [Strin
unlines ls = go1
```

go1 :: [[Char]] -> [Char]

Description	A term defined in the current module at the top level
Source	Generated at phase 15
IdDetails	Vanilla
Stats	#terms 12
	#types 14
	#coercions 0
	#val_binds 0
	#join_binds 0

IdInfo (at phase 19)

Arity	1
Lambda with 1 usage	no
Has Core unfolding	yes
Inline Pragma	
Occurances	loop breaker (strong)
Strictness Sig.	<1L>
Demand Sig.	L
Call Arity	0

Figure 4.1: The popup that appears when double-clicking on a variable. It displays all available information embedded in the Core AST for that term.

This means that our tool is does not necessarily compromise on available information, but merely utilizes an interactive environment to present the information less crudely. Clearly we were able to retrieve all the information needed from GHC without our tool, but the experience was somewhat painful.

We found the best way to analyse GHC's output was by writing it directly to a file so we could use features of most text editors like a string search to scroll through the output of each pass and quickly find points of interest. However, because of auxiliary definitions in the Core, the name of the current pass was usually not visible on screen without having to scroll up. This minor annoyance compounds to major cumbersomeness when having to repeatedly scroll back and forth in time to see how the structure changes. Similarly, it is tough to attribute fired rewrite rules to a specific pass without having to double-check regularly. In short, interpreting the textual output was more laborious and error-prone that our newly developed alternative.

In short, we find that working with our tool is a much more pleasurable and productive experience over using GHC's compiler output, but this is of course merely a subjective account. We hope that through our vision more people will agree that an improvement is possible and further debate ensues about how our existing solutions can be improved upon or inspire something better altogether.

Chapter 5

Discussion & Future work

5.1 Review of the tech stack

Through extensive deriving and code generation, we have bridged the gap between Elm and Haskell in our tech-stack rather seamlessly. Unfortunately, overcoming this issue has led us to merrily go down that path before realizing some important drawbacks.

Because we always envisioned a high degree of interactibility for exploring captures – as that is the lion's share of the value proposition – it made sense to extract the capture and make interpretable by a separate frontend solution. Furthermore, in order to work on the newest version of GHC, the plugin needs to have a minimal dependency footprint. Initially we thought that serialising to JSON would a natural fit, however with the added requirement of needing to match the generated decoder at the Elm site, the JSON deriving strategy was dependent on more than just `aeson`, namely `haskell-to-elm`. For all intents and purposes this is a great library but through no fault of its author it is not widely adopted and therefore not actively maintained. Concretely, this means when working on the latest commit of GHC the odds are very high that the plugin won't work because the underlying dependencies are not yet compatible. We ran into this issue when trying to patch `unlines` in Section 4.3 and were forced to analyze the effect of our patches the old-fashioned way by sifting through the textual output of GHC like described in Section 2.4.2.

Furthermore, the need to serialise the infinite Core AST datatype at all turns out to be problematic. As described in Section 3.3.1, to make the representation finite we need to address the issue of self-referential unfoldings by replacing call sites with only references to binding sites. This is fundamentally flawed because late in our research we stumbled upon an inconvenient truth in the documentation of `IdInfo`:

```
Two 'Id's may have different info even though they have the same
'Unique' (and are hence the same 'Id'); for example, one might lack
the properties attached to the other.
```

More research is needed whether this phenomenon actually occurs in real life and what shortcomings this brings, but needless to say it is a concerning flaw in the basis of our approach.

It is not immediately obvious what alternatives there are. Ideally there would be no need to serialise at all by running the frontend from the plugin directly via for example a terminal application itself written in Haskell (at the cost of some interactive features that the web offers). But is not at all obvious how to plug in this into the compiler pipeline directly. Besides, comparing two different compiler runs is completely of the table in this approach as the information only exists transiently during the compiler run. A separate terminal application could still relax some dependency pressure by using a far more stable and adopted serialisation method that is less likely to break with unreleased GHC versions. An important mention here is that `ghcjs` already provides a way to compile Haskell code to the browser. Even more promising is the rapid work that is currently being undertaken to add web assemply a compilation target to GHC.

Considering that human inspection of code aligns more with the responsibilities of the 'Haskell Language Server' (`hls`) [18] than the compiler itself, the argument could be made that `hls` is the right place to implement inspection tools. Regardless, integrating compilation inspection capabilities that are compatible with the standardized LSP protocol is an interesting avenue to explore in further research.

The tragic reality is that for this tool to work on bleeding edge GHC versions, it has itself to be part of the GHC build system, which is not realistic at this stage. In summary, we would recommend re-evaluating the tech stack and possible starting the development from scratch, inheriting mostly the lessons learned from our approach.

5.2 GHC knows more than it tells

A fundamental problem with the current snapshot approach are the multiple changes – even multiple kinds – that are applied in between each snapshot. It is largely impractical to programmatically try to recover the full story that connects one snapshot to the next. This leads one to wonder the possibilities that would be unlocked if GHC embedded information about the changes it has made directly in the AST. Consider for example a constructor that indicates that its contained expression was the result of an inlining:

```haskell
data Expr b =
  ...
  | Inlined Expr InlineDetails
  ...
```

Obviously these added nodes could be omitted during normal compilation to not affect memory and performance. Less obvious however is how these extra nodes would interfere with the transformations themselves that often rely on deep matching. Take for example detecting an opportunity for beta-reduction:

```haskell
betaReduce :: Expr b -> Expr b
betaReduce (App (Lam b e) arg) = substExpr b arg e
betaReduce e = e
```

The following invocation to `betaReduce` would not work as expected:

```haskell
-- unchanged
betaReduce (App (Inlined (Lam b e) details) arg) = App (Inlined (Lam b e) details) arg
```

A good robust solution to this is not obvious. Furthermore, we need to respect correctness invariant of these extra nodes. If we do for example make an extra case for `betaReduce`:

```haskell
betaReduce (App (Inlined (Lam b e) details) arg) = Inlined (substExpr b arg e) details
```

Then one should wonder if the `Inlined` tag is still entirely accurate. It might be more reasonable to have tags live for only 1 transformation step and be erased before the next one starts. This still allows for transient information to be captured and used for analysis without affecting the pipeline all that much. Relevant research into extendable ASTs that has already been integrated into the frontend Haskell language provide might provide a way to annotate nodes with extra information without affecting the pipeline at all [19].

Alternatively, the information could be written to a separate log that precisely describes events such as `Inlined [var] at location [loc]`. This solution has the added benefit of being able to peacefully coexist with the current Core AST. However, work is still necessary to explore how to best encode this information – especially precisely pointing to locations of events – as well as specifying the exact semantics necessary to replay the log in a third party application.

5.3 Printing Core like GHC

We have shown how an alternative Core printer is highly beneficial for both the inexperienced and those interested in the syntactical structure of a program. It does not take away from the reality that GHC's Core printer has its place. After all, reading Core as if it was Haskell is not entirely ingenuous. Take the complete ignorance towards showing which let bindings are join points for example, information that can be crucial to the transformation that a program undergoes.

Simply summarized, it cannot be a complete tool without allowing to display the Core in a way that is as close to the original GHC output as possible, but of course still with benefits of an interactive exploration frontend. A skeleton of this alternative pretty printer has been implemented, using GHC itself as a reference, but there still are some discrepancies not accounted for.

5.4 Incomplete transformation pipeline coverage

During our research we have mainly discovered and discussed Core debugging scenarios that involved the general syntactical structure of the code. For example, which functions are being called after rewrite rules and inlinings. Although program structure analysis might in fact be the main constituent of most Core debugging sessions, we have barely scratched the surface of other problems that need to be debugged in Core, and by extent the suitability of our tool to do so. Examples here would be inadequately strong analysis results, preventing what would have been a sound and desirable transformation. One such underdiscussed topic is the role that correctly identifying join points play. More research is needed into the full spectrum of Core related problems and the information required to deal with them.

5.5 Post Core factors

Despite the idealistic design principle of the 3 stage compiler (Section 2.1), where the Core transformation middle-section is responsible for all optimisations, reality throws a spanner in the works. A number of critical optimisations are only possible on a level closer to the hardware. Because of this, performance regressions cannot always be attributed to the Core pipeline. A real world exemplary encounter with this fact is a performance regression of `alfred-margaret` [4], a Haskell implement of the *Aho-Corasick* string search algorithm. After updating from GHC 8.8.4 to GHC 8.10.7 a 10% performance regression was observed, which is rather significant. After ruling out significant changes in the dependencies, it was noted that the used LLVM backend version 12 was not yet well-supported by the new GHC version. That is, LLVM 9 incidentally yielded better performance.

There is simply no way to account for this in the Core pipeline. However, our tool could be extended to capture more extensive build information that could then be diffed during inspection. Therefore, if a user establishes that the Core is not the culprit, he or she can still request to get a report of all differences in the build processes. This would include the LLVM version, the used C compiler, the linker, the assembler, etc. This way `hs-sleuth` can still provide some leads during performance regression analysis.

Lastly, information could be retrieved information from GHC's *Spineless, Tagless, G-machine* (STG), which converts the functional code to imperative form. This process similarly takes care of some optimisations that were otherwise not possible in the Core pipeline. Whether, and if so how, this telemetry can play a useful role in `hs-sleuth` is an avenue for future research.

5.6 Fusion by rewrite rules

As part of collecting Core related performance issues from the past, we made the observation that rewrite rules were the usual suspects, and often rightly so. Especially rewrite rule based fusion, either within GHC itself or as part of custom library, come with many brittle interactions that cause failures on specific cases are after small changes. This naturally begs the question if rewrite rules are a suitable method for implementing fusion. We believe research is needed to evaluate the feasibility of fusion specific transformations that rely on, and interfere less, with the simplifier.

Chapter 6

Conclusion

We have shown how GHC can have unpredictable results when it comes to its Core optimisation pipeline. We motivated why existing techniques to analyze the output of the core2core pipeline are unsatisfactory. We then proposed our custom tool `hs-sleuth` to aid in this discovery process. During the development of `hs-sleuth`, we set out to answer a number of questions (Section 1.4) which have since been answered. We answer them here explicitly for completeness's sake.

Main Question How can GHC's core2core passes be captured and presented in such a way that users productively gain insight into how their code is transformed?

A snapshot of the current state of the program can be captured by embellishing the core2core pipeline with snapshot passes via GHC's plugin interface. The captured ASTs can be pretty printed in a web application to provide interactive features such a hovering, clicking, dragging, and highlighting. By providing various suppression options, the Core snapshots can be presented such that they nearly resemble canonical Haskell source code. Finally, by allowing users to scroll through time, they can quickly observe if and when their expectations are not met.

Sub-Question 1 How does one efficiently identify where small changes occur in two or more captures?

While a novel tree diff approach might prove more usefull, matching toplevel definitions by a hash of their body is a productive approach to highlight deviations among captures.

Sub-Question 2 How to make viewing core more manageable using various display options?

Allowing programmers to hide as much irrelevant information as is sensible for the goal at hand is a key part to ensure manageability. Moreover, making information such as variable names manually more precise significantly reduces the mental strain of reading the Core.

Sub-Question 3 How could performance regressions that have occurred in the past in popular Haskell projects, have been resolved faster?

As `hs-sleuth` lowers the bar to entry for analysing Core, we think that it will similarly become a more approachable task for developers to confirm their optimisation approaches and subsequently guard against regressions using non-functional tests.

Bibliography

[1] Ghc wiki: Core to core pipeline, 2022.

[2] Ghc wiki: Worker wrapper, 2022.

[3] J. Breitner. A promise checked is a promise kept: Inspection Testing. *arXiv e-prints*, page arXiv:1803.07130, Mar. 2018.

[4] Channable. alfred–margaret. https://github.com/channable/alfred-margaret, 2022.

[5] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion. from lists to streams to nothing at all. volume 42, pages 315–326, 09 2007.

[6] E. Czaplicki. Elm: https://https://elm-lang.org/.

[7] T. Fausak. 2021 state of haskell survey results: https://taylor.fausak.me/2021/11/16/haskell-survey-results.

[8] G. foundation. Ghc optimisations. https://wiki.haskell.org/GHC_optimisations, 2022.

[9] O. Fredriksson. haskell-to-elm. https://hackage.haskell.org/package/haskell-to-elm, 2019.

[10] B. Gamari. ghc-dump: https://github.com/bgamari/ghc-dump, 2019.

[11] J. A.-S. Georg Brandl, Matthäus Chajdas. pygments-pretty-printer. https://pygments.org/, 2019.

[12] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. *[No source information available]*, 04 1995.

[13] C. Hollenbeck, M. F. P. O'Boyle, and M. Steuwer. Investigating magic numbers: Improving the inlining heuristic in the glasgow haskell compiler. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, Haskell 2022, page 81–94, New York, NY, USA, 2022. Association for Computing Machinery.

[14] A. Lelechenko. A tale of two tails. https://github.com/haskell/text/pull/348, 2021.

[15] P. Q. Matt Godbolt, Rubén Rincón and A. Morton. https://godbolt.org/, 2022.

[16] L. Maurer, P. Downen, Z. M. Ariola, and S. Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 482–494, New York, NY, USA, 2017. Association for Computing Machinery.

[17] K. Maziarz, T. Ellis, A. Lawrence, A. Fitzgibbon, and S. Peyton Jones. Hashing modulo alpha-equivalence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'21)*. ACM, ACM, June 2021.

[18] N. Mitchell, M. Kiefer, P. Iborra, L. Lau, Z. Duggal, H. Siebenhandl, J. N. Sanchez, M. Pickering, and A. Zimmerman. Building an Integrated Development Environment (IDE) on top of a Build System. *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, 9 2020.

[19] S. Najd and S. Peyton Jones. Trees that grow. *JOURNAL OF UNIVERSAL COMPUTER SCIENCE*, 23, 10 2016.

[20] S. Peyton Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, 12:393–434, July 2002.

[21] S. Peyton Jones and A. Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1), October 1997.

[22] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in ghc. *Haskell 2001*, 04 2001.

[23] pwentz. elm-pretty-printer. https://github.com/the-sett/elm-pretty-printer, 2019.

[24] D. Steward. Stream fusion on hackage https://hackage.haskell.org/package/stream-fusion, 2006.

[25] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[26] P. Wadler and J. Kilmer. A prettier printer. 2002.

[27] J. M. Young. *The Haskell Optimization Handbook*. Haskell Foundation, 2022.

[28] K. Yul Seo. Short cut fusion: https://kseo.github.io/posts/2016-12-18-short-cut-fusion.html, 2016.