CSI: Haskell - Tracing Lazy Evaluations in a Functional Language

CSI: Haskell - Tracing Lazy Evaluations in a Functional Language

Matthías Páll Gissurarson
Chalmers University of Technology
Gothenburg, Sweden
pallm@chalmers.se

Leonhard Applis
TU Delft
Delft, Netherlands
L.H.Applis@Tudelft.nl

ABSTRACT

In non-strict languages such as Haskell the execution of individual expressions in a program significantly deviates from the order in which they appear in the source code. This can make it difficult to find bugs related to this deviation, since the evaluation of expressions does not occur in the same order as in the source code. At the moment, Haskell errors focus on values being *produced*, whereas it is often the case that faults are due to values being *consumed*. For non-strict languages, values involved in a bug are often generated immediately prior to the evaluation of the buggy code. This creates an opportunity for *evaluation traces*, tracking recently evaluated locations (which can deviate from call-order) to help establish the origin of values involved in faults. In this paper, we describe an extension of GHC's Haskell Program Coverage with evaluation traces, recording recent evaluations in the coverage file, and reporting an evaluation trace alongside the call stack on exception. This lets us reconstruct the chain of events and locate the origin of faults. As a case study, we applied our initial implementation to the `nofib-buggy` data set and found that some runtime errors greatly benefit from trace information.

CCS CONCEPTS

• **Software and its engineering** → *Functional languages*; *Software testing and debugging*.

KEYWORDS

Laziness, Fault-Localization, Errors, Tracing

ACM Reference Format:
Matthías Páll Gissurarson and Leonhard Applis. 2023. CSI: Haskell - Tracing Lazy Evaluations in a Functional Language. In *The 35th Symposium on Implementation and Application of Functional Languages (IFL 2023), August 29–31, 2023, Braga, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3652561.3652562

1 INTRODUCTION

Problem and Motivation. In crime scene investigation (CSI), establishing the sequence of events constituting a crime is a key technique in solving cases. While less dramatic, programs can still die: despite satisfying the compiler, even Haskell code can crash or have faulty output. When an error occurs at runtime, a common approach is investigating the reported call stack to determine where the error originated. As an example, the code in Figure 1 crashes with `*** Exception: Prelude.head: empty list`, and provides an error message containing the stack trace (seen in Figure 2). Despite the crash in `head`, the root cause of the error is based on `divs n` which results in an empty list when `n` is prime (there is an off-by-one error: `n` should be `n+1` in line 4). This is a motivating case of an error caused by the wrong data being *produced*, in contrast to errors caused by the right data being incorrectly *consumed* [1] (e.g. evaluating an `undefined` that should have been replaced). The stack trace in Figure 2 does not mention `divs`, and only indicates that the error stems from `head`. This lack of information makes it difficult for developers to reconstruct the events that led to the error and determine the root cause of the fault.

Without further hints, any function used (and its dependencies) is a potential suspect. This offset in the tempo of call and evaluation is not a novel discovery; in fact, a similar example to Figure 1 was presented by Marlow in 2007 [19].

```
1  module Main where
2  divs :: Int -> [Int]
3  divs n = go 2
4    where go i | i == n = []
5          go i = if d i
6                 then i:(go (i+1))
7                 else go (i+1)
8          d i = n `mod` i == 0
9
10 smallestDiv n = head (divs n)
11
12 main :: IO ()
13 main = print (smallestDiv 13)
```

Figure 1: Our running example, a generator for the divisors of a number, with an off-by-one error in the base case.

Approach. To address the issue, we implement an extension to Haskell Program Coverage (HPC) built into GHC: in addition to tracking expression evaluation with *ticks*, we also emit instructions in the intermediate language to track the order of *started evaluations* and *completed evaluations*. HPC is discussed in further detail in Section 2. We also track the current *evaluation depth*, the number of ongoing evaluations. This allows us to reconstruct a partial *evaluation tree* an overview of completed, partial, and uncompleted evaluations of expressions, when an exception occurs (see Section 3.2 for details). We also track a *global trace index* that allows us to reconstruct a trace across all modules from the trace

[1] This can be translated to *blame*: is it the producer or the consumer that is wrong? Call stacks help to spot bugs in consumers, while we focus on bugs in producers for this work.

IFL 2023, August 29–31, 2023, Braga, Portugal

ACM ISBN 979-8-4007-1631-7/23/08.
https://doi.org/10.1145/3652561.3652562

```
divs: Prelude.head: empty list
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/List.hs:1643:3
    in base:GHC.List
  errorEmptyList, called at
    libraries/base/GHC/List.hs:82:11 in base:GHC.List
  badHead, called at libraries/base/GHC/List.hs:78:28
    in base:GHC.List
  head, called at Div.hs:10:17 in main:Main
CallStack (from -prof):
  Main.smallestDiv (Divs.hs:10:17-29)
  Main.main (Divs.hs:13:15-28)
  Main.main (Divs.hs:13:8-29)
  Main.CAF (<entire-module>)
```

Figure 2: Error message generated by the program in Figure 1.

of each individual module. These *recent evaluations* are kept in a circular buffer alongside the HPC ticks and can both be inspected directly at runtime or summarized and reported on an exception alongside the call stack.

In particular, adding an evaluation trace for users is as easy as passing an additional flag during the compilation phase. It constitutes a noninvasive addition to debugging, does not require any changes to the developer's code (such as call stack annotations) and allows a better understanding of what is going on at runtime even when external libraries are being used.

This extension for tracking evaluation traces constitutes the main contribution of this work. Improved runtime errors are one low-level domain that motivates the extension and is easy to understand for broad audiences. In the future, these evaluation traces could be used for more sophisticated use cases, such as program repair or visualization (see Section 5).

Experimental Evaluation. We apply our prototype to a subset of the `nofib-buggy` data set [32]. The data consist of a selection of `nofib` programs which GHC uses for internal validation with artificially introduced bugs (see Section 3.7). These bugs result in either a runtime exception (e.g. index-out-of-bounds or division by zero) or incorrect output. In our pre-processing of the data set, we

- remove all non-terminating programs, and
- add `assert` statements to those data points that return incorrect output to force a runtime exception.

This accounts for a total of 21 investigated data points. From the initial findings, we see a trend that certain exceptions benefit from trace information, depending on the exception type. The data points using `assert` usually cover the fault, but the quality of the trace is dependent on the scope of the test — unit tests are more precise, while system tests produce crowded traces with many locations irrelevant to the introduced bug. We analyze the performance overhead introduced by collecting traces, which seems stable: most data points require between 100% and 300% more compute time, depending on the length of the collected trace. Maximum memory usage increases from 20% to 120%, and the additional binary size is negligible. There is a general trend that the additional memory allocation is related to the number of modules, while the additional compute time seems to depend primarily on the total number of evaluations the program makes. As the `nofib` data set is used in the current test suite and the benchmarking of GHC, we consider it representative of performance estimates. We thus suggest collecting and reporting the evaluation on a per-exception basis. In the long-term view, we hope to support debugging for new and seasoned Haskell programmers alike, but we also see the potential for classroom use: using the data collected by HPC at runtime the evaluation tree can be partially reconstructed (up to the length of the trace) and a clearer view of non-strict evaluation presented to students. *Understanding* laziness is a big challenge for students from other programming paradigms, and visualizing (both buggy and working) program evaluation traces can be a great aid. Our experiments are shared in a replication package[2].

We utilize `nix` and shell scripts for easy replication, but we also provide the output (enhanced error messages) for lightweight investigation without additional dependencies. The contributions presented in this work can be summarized as follows:

(1) a prototype implementation of a non-invasive, optional, coverage-based tracing of evaluations,
(2) example tooling-improvement by reporting of evaluation traces alongside call stacks,
(3) an initial investigation on the `nofib-buggy` data set, and
(4) estimates of performance overhead

2 BACKGROUND AND RELATED WORK

Thunks. In non-strict languages, values are not evaluated until needed in the computation. In Haskell, this is implemented through *thunks*: instead of directly producing a value, expressions produce a *thunk* that represents that unevaluated expression. This behavior is similar to asynchronous concepts in other languages like `Promises` (JavaScript) or `Task` (C#), which are often used for side-effectful computations (e.g. network requests), while in Haskell they are used for all computation. When the value of that expression is required, the thunk is evaluated and resolved to a value. This value might be fully-evaluated if it is, e.g., an integer. But it might also be just the head of a list, with the rest of the list being another thunk. Thunks are in most cases memoized, meaning that the value is evaluated only once, and the result saved. This is then *shared* if the value is needed again at a later time, without requiring re-computation.

Program Coverage and Ticks. Haskell Program Coverage (HPC) is a tool that is part of GHC and allows developers to track which expressions were evaluated during the execution of the program: whenever an expression is evaluated, it bumps a number in an array (a "tick") [12, 14]. These numbers are unique identifiers specified in a per-module `mix`-file, which are on tick registered in a companion per-program `tix`-file. For this work, we reuse the `mix`-files and identifiers unchanged, and extend the `tix`-files with extra arrays using the identifiers. Note that HPC does not require changes to the source code, but instead operates with compilation flags that emit additional instructions to the intermediate language. As we consider this an elegant interaction, we also strive for a flag-based change to the intermediate language.

[2] https://doi.org/10.5281/zenodo.10090375

The data collected by HPC can be used to generate a report on how many times each expression was evaluated and used, for example, to check the test coverage or identify unused code.

Stack Traces & Error Messages. While research on the use of stack traces is a popular topic, e.g. when applied for program slicing or crash reproduction, their dedicated value for manual debugging is not thoroughly investigated. Bettenberg et al. [7] investigate differences originating from different perspectives of bug reports. One of their findings is that developers need information that users rarely provide, of which stack traces are particularly useful.

We also recommend the work of Becker et al. [6] as a general overview of research on error messages. Their extensive meta-study covers many findings and trends from the fields of technical implementation, pedagogic use, and improvement of error messages. Among their primary findings relevant to this work are: students and programmers alike actually read error messages and stack traces [5], students are discouraged when error messages do not point toward the faults [28, 33, 38] and that cognitive load should be considered in the design and presentation of errors [24, 25, 31] (i.e., do not overwhelm the user with information). Lastly, motivating this work, they identify *localization* as one of the defining technical attributes of error messages that constitute their quality, and we aim to enable better localization.

We argue that our work contributes towards the usefulness of traces and forms a starting point for similar research on Haskell. In the absence of detailed analysis in Haskell, our objective is to provide a similar investigation to that of Schroeter et al. [29] in the coverage of bug locations through stack traces. In their work, Schroeter et al. run buggy programs with known faults and investigate the produced stack traces to determine whether and where they contain the fault. We reproduce this for the `nofib-buggy` stack traces and apply the same approach for the evaluation traces.

Stack Traces for Haskell. We often take stack traces for granted, but they have only been available in a limited form until recently for Haskell. As late as 2009, Allwood et al. [2] and Marlow [19] tackled the first issues that appeared due to a mismatch between the source code and the optimized executed code. Their central contribution is to address the differences between the behavior of the stack (and stack traces) and the original program syntax, by introducing a transformation of GHC core programs into ones that simulate passing a stack around to preserve the stack trace of the executed program [2]. This was further improved by introducing the `HasCallStack` constraint that does not need to be simulated by the runtime system, but while this constraint can sometimes be inferred, our experiments with the `nofib-buggy` data set show that this is not often the case. Similarly, the simulated call stack adding `-prof` in conjunction with the `-fprof-auto` and `-fprof-auto-calls` flags is either

- not printed for the exceptions in `nofib-buggy`, with the output being `Main: divide by zero` or similar,
- or in the form of

```
CallStack (from HasCallStack):
  assert, called at Main.lhs:78:5 in main:Main
CallStack (from -prof):
  Main.main (Main.lhs:(75,3)-(78,59))
  Main.CAF (<entire-module>)
```

indicating the `assert` and the `main`-function, without giving further clue to the location of the bug.

In the output of our running example `div` from Figure 2, adding the `-prof -fprof-auto -fprof-auto-calls` improves the situation slightly, indicating the `smallestDiv` function, but this improvement does not extend to the `nofib-buggy` data points. Using GHC's profiling also requires annotating the `Prelude.head` function with a `HasCallStack` constraint, but it is still not enough to locate the fault. Manual annotations with `HasCallStack` are non-optimal for programmers and in our running example extend the information on the crash, but not on the source of it.

Based on the existing research, the current state of Haskell stack traces faces two main challenges: higher-order functions and lazy evaluation. Especially when combined, these tend to disturb stack traces or produce errors that are not aligned with the reported traces. We hope that our work improves the quality of errors for lazy evaluation and enables later researchers to improve errors for higher-order functions.

Tracing Evaluations for Haskell. There have been some approaches to tracing Haskell evaluations, which differ from the coverage-based technique presented in this work. Chitil et al. [36] compared three systems available in 2000: HAT [9], HOOD [13], and Freja [21]. Another system mentioned is Buddha [20].

Some approaches are conceptually different, namely Buddha and Hood require changes on a source code level. This limits their application, e.g. is it hard to extend tracing to external libraries.

A part of Freja consists of a custom Haskell compiler that covers a subset of the Haskell98 standard (e.g. excluding `IO`). The code that is compiled is altered in an intermediate language, and the redexes are recorded. Frejas concept is the closest to the one presented in this work. Our approach differs by ① extending existing GHC modules instead of requiring an extra compiler ② covering a trace of the last evaluations instead of *all* evaluations and ③ tracking whether an evaluation was started and or finished separately.

In a similar manner, HAT is tied to nhc98 and transforms the source code outside of the compilation process, which can cause performance issues and is harder to integrate with external tools.

With their dual systems of data creation and browsers, the existing tools went a step further than the contribution of CSI: Haskell. Concepts of how to use the evaluation data produced are presented in Section 5. We also hope that the separation of tracing and debugging helps to create additive tools in a modern Haskell landscape.

Static Methods. CSI: Haskell is a *dynamic* approach, based on running the code, in contrast to *static* methods, which analyze faults without running the code. In Haskell, there is already a rich type system that allows expressing a wide variety of behavior that is checked at compile time. However, these do not capture many attributes commonly expressed with properties. One approach to lift "property-style" testing and debugging to static checking is to use *refinement types* [35]. These types of checks are integrated through a GHC plugin [27], allowing properties such as `x > 0 ==> f x > 0` to be statically checked by an SMT solver. One extension to this is *lazy counterfactual symbolic execution* [15]: When paired with refinement types such as those in Liquid Haskell, lazy counterfactual execution allows the localization of refinement

type errors, revealing faults in the code to be found without need for tracing. This constitutes a heavier approach and raises the entry barrier for ecosystems (including libraries) that do not yet have a refinement type specification.

Algorithmic debugging. Algorithmic debugging methods are dynamic approaches based on recording information during program execution and then asking the developer whether the intermediate statements agree with their intention [10]. In most languages, this debugging suffers from side effects, which are no concern in pure functional languages, making Haskell a prime candidate for algorithmic debugging. One tool for Haskell is HOED [10], which extends the debugger HOOD [13] with GHC's profiling information. Like HOOD, it requires users to annotate the functions that they wish to "observe" and create profiling cost centers. Based on this combination, it is possible to construct a computation tree from the collected traces for the observed expressions [10]. This rich approach provides a lot of information but differs from CSI: Haskell in a few points. First, CSI: Haskell utilizes HPC and thus coverage and does not rely on tracing and cost centers. Second, our approach is capable of capturing evaluation trees, in a similar manner to computation trees, providing information on the *actual execution of an evaluation* (that is, the state of each value within the captured tree), but do not capture the values themselves. Lastly, CSI: Haskell gathers data on the entire project and does not require manual annotations on suspicious elements of the codebase. Thus, we start with an earlier stage of debugging, where suspicious elements still need to be identified.

We consider CSI: Haskell not as a debugger, but instead a source of trace information for follow-up tools. The example presentation as evaluation traces could greatly benefit from concepts of algorithmic debugging, but lies beyond the scope of this work.

3 APPROACH

3.1 Evaluation Trees

The approach taken by CSI: Haskell is aimed at the collection of just enough data at runtime to reconstruct the global *evaluation tree* of a program. Lazy functional program evaluation can be viewed in terms of an evaluation tree: the evaluation of each expression requires the evaluation of its subexpressions whenever those expressions are needed to produce output [18]. For Haskell, this evaluation has been linearized using implementations of machines such as Sestoft's lazy abstract machine [30], placing evaluation trees on sound theoretical foundations and (by now) a robust amount of experience. Re-using the theoretical structures lends itself for the application of debugging too: For debugging, a tree-like view of the expressions and the order of evaluations for each component is an important part of understanding the programs and how they behave. This is especially relevant when the programs fail and throw an exception at runtime, e.g. the evaluation tree in Figure 3. This tree shows how evaluation proceeds by resolving the functions to be used in the relevant context (using big-step semantics, denoted by "⇓" for readability), which are then applied to the fully resolved value of their arguments, resulting in their final value.

3.2 Trace Data

To collect the data used in constructing the trace, we extend HPC, the Haskell Program Coverage built into GHC by Gill et al. [14]. HPC divides the source code into *expression boxes*, which are extracted during compilation and stored in an associated `mix` file. The code is then instrumented with additional instructions in the intermediate language (C- -) to add a *bump* to the appropriate array value when the evaluation of an expression starts (i.e. its output is demanded). At runtime, HPC maintains a module-per-module in-memory array, with one entry per expression in the original program. Whenever an expression starts to be evaluated, the corresponding array entry is incremented with the bump instruction, allowing HPC to track the coverage of programs [14]. CSI: Haskell adds three additional in-memory arrays to the runtime system, along with additional bookkeeping, the *trace array*, *evaluation depth array*, and *global index array*. An example of these for the program in Figure 1 is provided in Section 3.3.

The Trace Array. The first additional array holds the trace itself, a log of values corresponding to the expression boxes defined by HPC. This array contains an entry whenever an expression starts being evaluated, and another entry whenever an expression finishes being evaluated to the outermost constructor. Each entry represents an explicit expression in the source code, which is the same as that used for the original HPC coverage: for any single expression e, the original coverage tracks the number of times that expression is evaluated. For example, if we look at an expression e_i with $i = 5$, at the beginning of the evaluation of e, the index number 5 would be incremented in the corresponding array in the `tix`-file. With our additions, the index 5 is written in a trace array both when the expression starts to be evaluated and when it has finished evaluating. Note that since Haskell is non-strict, the evaluation of an expression might not return a fully evaluated value, but rather a weak normal form represented by a constructor whose components might further, not yet fully-evaluated thunks. To log these evaluations, an additional register is introduced, in which the (possibly partial) value of an expression is saved. The completion is then recorded and the register is returned. This allows us to log the completion of evaluations even when they would have immediately returned, at the cost of additional overhead at runtime.

The Evaluation Depth Array. The second array keeps a log of the current *evaluation depth* before the start of the evaluation of an expression and the depth before the completion of the evaluation of an expression. Using the two in conjunction, the evaluation depth and trace arrays allow us to reconstruct a partial view of the full evaluation tree of the program and determine whether an entry in the trace array corresponds to the start of evaluation or the completion of evaluation of the indicated expression. It also lets us determine which evaluations have started and not finished, allowing us to determine the current call stack in terms of expressions. This allows us to see which evaluations were started and finished in the same subtree of the *evaluation tree*, allowing us to highlight the branches of the evaluation that are "close" in the tree.

The Global Index Array. The third array tracks a global counter, associating each index in the other two arrays with a unique integer timestamp. This allows us to reconstruct a global trace across

$$\dfrac{\Gamma_0 \vdash \texttt{head}^2 \Downarrow \lambda \texttt{xs. head' xs}, \Gamma_1 \qquad \dfrac{\dfrac{\Gamma_1 \vdash \texttt{divs}^4 \Downarrow \lambda \texttt{n. divs' n}, \Gamma_2 \qquad \Gamma_2 \vdash \texttt{n}^5 \Downarrow \texttt{n'}, \Gamma_3 \;\; \texttt{where} \;\; [\![n = n']\!] \qquad \dfrac{(\ldots)^6 \qquad (\ldots)^7}{\Gamma_3 \vdash \texttt{divs' n} \Downarrow \texttt{xs'}, \Gamma_{n-1}}}{\Gamma_1 \vdash (\texttt{divs n})^3 \Downarrow \texttt{xs'}, \Gamma_n \;\; \texttt{where} \;\; [\![xs = xs']\!]}}{\Gamma_1 \vdash \texttt{head' xs} \Downarrow \texttt{v}, \Gamma_n}}{\Gamma_0 \vdash (\texttt{head (divs n)})^1 \Downarrow \texttt{v}, \Gamma_n}$$

Figure 3: Evaluation tree for `head (divs n)` in Figure 1. Superscripts refer to indices of expressions in the Section 3.3 example.

modules, by gathering the trace for each module and sorting by the global counter.

Trace Length & Circular Buffers. Keeping track of an arbitrarily long run of a program would require a trace entry for each expression evaluated. For long-running programs, this would require an excessive amount of memory. As noted in the introduction, errors usually involve *recently evaluated data*. By keeping the length of the arrays constant and introducing a modulus operation to the running index, we effectively treat them as circular buffers. This allows us to keep track of only the most recently evaluated locations at the time of an error, giving us a "window" into what the program was executing right before the error occurred. Configurable with a compiler flag, this allows users to select how much memory overhead they are willing to trade off for a longer trace. Alongside the computational impact, there is also an information trade-off; some errors are captured only in longer traces, but unnecessarily long traces form a barrier to understanding. We investigate both trade-offs in Section 4.

3.3 Example

As an example, consider the evaluation of the expression head `(divs n)` and its evaluation tree shown in Figure 3. Here, E_1 corresponds to the expression superscripted with 1, that is, head `(divs n)`, E_2 to head, E_3 to `(divs n)`, and so on. Note that each expression has an annotation, as well as each of its subexpressions. In the interest of brevity, E_6 and E_7 are not shown, nor are any of their subexpressions. Using the annotations provided in the figure, a successful evaluation trace would be $[E_1, E_2, E_2, E_3, E_4, E_4, E_5, E_5, \ldots, E_3, E_1]$, with the associated evaluation depths $[0, 1, 2, 1, 2, 3, 2, 3, 2, \ldots, 2, 1]$. The global trace array would simply be $[1, \ldots]$, since there is only one module involved. The evaluation proceeds as follows: At the start of evaluation, the evaluation depth is 0. We start by evaluating `head (divs n)`, indicated by E_1. The evaluation depth is now 1. E_1 demands evaluation of head, i.e. E_2. Since we started evaluating E_2 and have not finished E_1, the depth of the evaluation is now 2. The function head is from a library, which returns directly, indicated by E_2, and the evaluation depth decreases to 1. Now the implementation of head, `head'` demands its first argument, which causes evaluation of `(divs n)`, i.e. E_3, resulting in an evaluation depth of 2. This continues until E_3 completes, which in turn lets E_1 complete, and the program is fully evaluated. *However*, if E_3 results in an empty list, the evaluation $\Gamma \vdash$ `head' xs` $\Downarrow v$ will throw an exception, aborting execution *before* logging that E_1 finished. The trace will show that E_3 was the last expression to complete evaluation before the error.

```
divs: Prelude.head: empty list
CallStack (from HasCallStack):
  error, called at
    libraries/base/GHC/List.hs:1749:3 in base:GHC.List
  errorEmptyList, called at
    libraries/base/GHC/List.hs:89:11 in base:GHC.List
  badHead, called at
    libraries/base/GHC/List.hs:83:28 in base:GHC.List
  head, called at Divs.hs:10:17 in main:Main
CallStack (from -prof):
  Main.smallestDiv (Divs.hs:10:17-29)
  Main.main (Divs.hs:13:15-28)
  Main.main (Divs.hs:13:8-29)
  Main.CAF (<entire-module>)
Recently evaluated locations:
  Divs.hs:4:25-4:26  ... = []
  Divs.hs:4:16-4:21  |...,i == n,...=... (was matched)
  repeats (11 times in window):
    Divs.hs:4:9-7:28   Main:divs>go
    Divs.hs:7:21-7:28  ... = go (i+1)
    Divs.hs:5:19-5:21  ...else d i
    Divs.hs:8:9-8:28   Main:divs>d
    Divs.hs:5:16-7:28  ... = if d i...
    Divs.hs:4:16-4:21  |...,i == n,...=... (not matched)
  Divs.hs:4:9-7:28   Main:divs>go
  Divs.hs:3:1-8:28   Main:divs
 Previous expressions:
  Divs.hs:10:1-10:29  Main:smallestDiv
  Divs.hs:13:1-13:29  Main:main
```

Figure 4: The improved error message includes a list of recently evaluated locations. The preceding number is the index of the expression in the `mix` file and is used to distinguish different expressions at a glance.

3.4 Persistence and Tix Upgrades

As CSI: Haskell is integrated with HPC, we also extend the `tix` file format that HPC generates to persist information between runs to include the trace and evaluation depth information.

Apart from changes to the `tix`-format, the tracking is noninvasive and requires no modification of the programs on behalf of the user. Setting the size of the trace buffers to a sufficient length can be used to generate traces across multiple runs of a program. These extended `tix` files, along with the associated `mix` files that store the expression boxes, can be parsed by external tools for further analysis and presentation. One such presentation is a SARIF file derived from the a trace, allowing further integration of Haskell traces into IDEs [3]. This has been explored with a short prototype by the authors and is feasible; however, with respect to the scope,

we consider it future work (see Section 5). A non-textual presentation of the trace could be to visualize the behavior of the program as a graph, as shown in Figure 5.

3.5 Output

The additional information can be accessed via runtime reflection using the GHC-API, and consumed by external tools such as automatic program repair tools, testing frameworks, and IDEs. As one immediately accessible application, we adjust the current runtime error printing in GHC and add a message detailing the recently evaluated locations. Using the trace array in conjunction with the evaluation depth array, we generate a list of recently evaluated locations. By comparing the current evaluation depth on an error and the evaluation depth array, we determine the involved expressions whose evaluation was demanded by the expression on top of the call stack at the time of crash. We label the rest as "previous expressions", whose evaluation was complete before the evaluation of the expression on top of the call stack started and therefore were not triggered by the expression on top of the call stack. As an example, Figure 4 shows the new output generated for `divs` from Figure 1, which shares the evaluation tree with the example above.

3.6 Summarization and Presentation

Since we track all evaluated expressions, the traces can become quite long. To effectively display error messages, filtering and summarizing traces is important. The summarization of traces is a rich field [16, 22], but often involves the full instrumentation of the program from the beginning to the end, while CSI: Haskell has a limited window of recently evaluated locations. To be useful as the default when printing error messages, the summarization of the traces must be done quickly and efficiently, avoiding unnecessary delay when reporting errors. The current approach in CSI: Haskell is to remove all unconditionally evaluated expressions done before the last branch. This makes the traces much shorter while keeping the relevant information about the evaluated expressions immediately preceding the error. Another summarization that CSI: Haskell implements is to merge repeated patterns that occur in loops and indicate them as repetition in the output, with the caveat that it only captures repetition in the "window" that the trace offers and may miss out on some longer patterns. This technique struggles when there is variation in the loop, such as when it is conditionally different for each iteration, e.g. cases for odd and even numbers. We aim to mitigate such variations using graph-based trace modeling and using more of the information available on the structure of the code during summarization (see Section 5). As described earlier, we used the evaluation depth at the time of a crash in conjunction with the tracking of the evaluation depth to segregate expressions that were evaluated at the current evaluation depth or lower and those that occurred earlier. Looking at the evaluation depth array also allows us to construct a partial notion of the call stack at the time of the crash, though some information might have been lost due to truncation in long-running programs. In this way, we can track the call stack for any program without manual annotations of `HasCallStack =>` throughout the code. Since CSI: Haskell is integrated into the compiler and runtime system itself, it can be easily applied to external Haskell libraries and dependencies simply by adding an additional flag during compilation. This helps developers trace issues that originate in external libraries and understand the interaction of their code with the library.

As for presentation, the current implementation reads the relevant locations from the source file, and displays them in a manner appropriate to their form, whether it is a branching if statement, guard or qualifier in a list comprehension or a non-branching expression. To further shrink the output, we only show non-branching expressions up to the last branching expression in the trace. This allows the focus to be on the control flow up to the point where the evaluation of a non-branching expression might have caused the error. When the total number of evaluations exceeds the window, a short statement is appended to the error message showing the total number of evaluations and suggesting to increase the trace length before rerunning. We stress that the current presentation is a prototype and outline the need for further research in Section 5.

3.7 Data

Apart from the motivating example in Figure 1, we draw data from the `nofib-buggy` data set [32]. In this data set, Silva introduced artificial bugs of various categories to the data points of the `nofib` benchmark [23] used in the GHC test suite.

We utilize a subset of 21 bugs summarized in Table 1. Our biggest exclusion criteria of the original `nofib-buggy` was the category of *non-termination*; since our evaluation is based on crashes, non-termination does not provide the output we need. Similarly, `StackOverflowExceptions` are errors of the environment, not necessarily in the program. These exceptions come from the runtime system itself and not from the program, so such exceptions were excluded as well.

Lastly, for ease of comparison, we modified programs that merely produced incorrect results to fail with an exception using an `assert`. These assertions are constructed using the console output (`stdout`) of the correct programs. Due to the lack of annotations, the call stacks in these examined cases are all trivial and only show the call for equality in the assert, but the evaluation traces often span relevant locations in the code. We admit that the assertions based on string comparison are neither sophisticated nor best practice. In the spirit of a vertical prototype, we aimed to see *"can evaluation traces help with testing?"* Despite looking a bit ad-hoc, the insights might be as valuable as the inspection of runtime errors: a healthy project should address problems in the test suite and not at runtime. Additions to the testing toolkit may pay off earlier than post-mortem debugging tools.

4 INITIAL RESULTS

To analyze the results, we recompiled the `nofib-buggy` data set with a fork of GHC and HPC that implements CSI: Haskell as described in Section 3. After obtaining crash logs, two authors looked at each log separately, deriving data and judging the merits of the new output. All code, data points, logs, and evaluations are provided in the companion package archived at https://doi.org/10.5281/zenodo.10090375. The remainder of this section covers the summary and highlights of the findings.

Summary & Overview. Table 2 presents the results achieved by the `nofib-buggy` data as shown in Section 3.7: of the 21 data points,

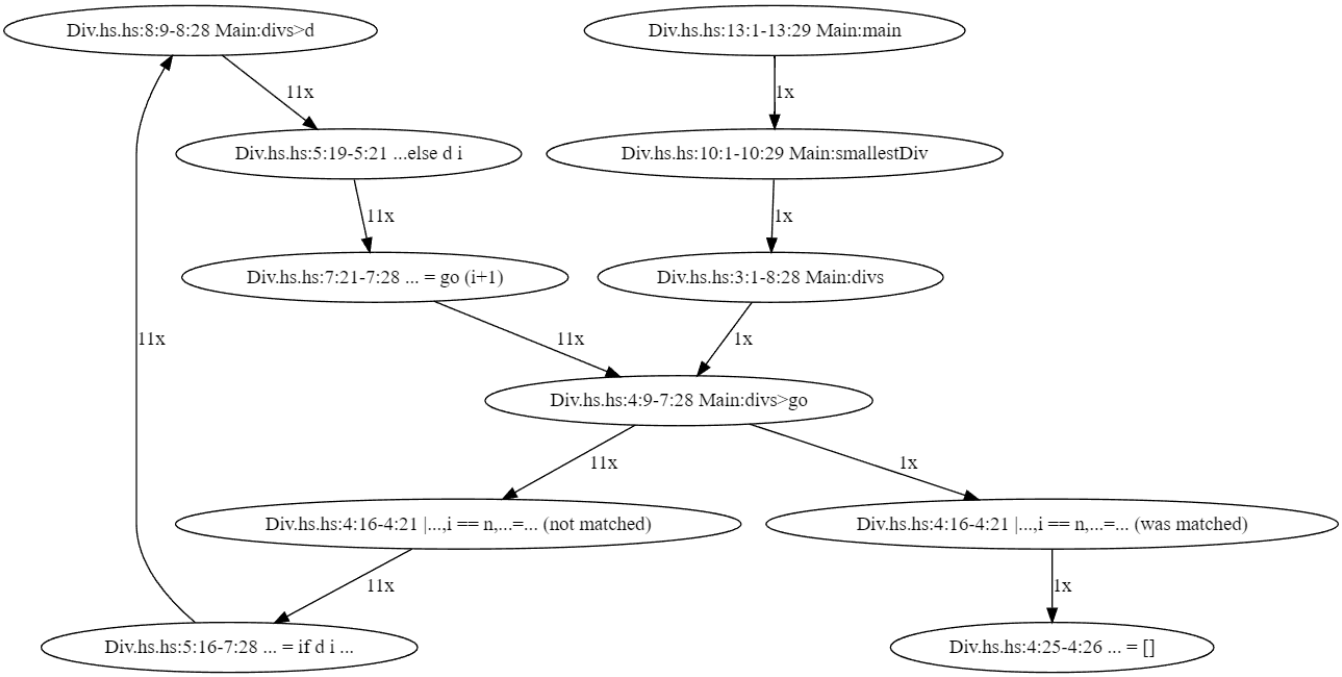

Figure 5: A graphical representation of the trace in Figure 4 generated by CSI: Haskell and an external script.

13 have the location of the error appear within a trace length of 50 and 19 in traces of length 1000 visualized in Figure 6 and Figure 7.

Visible in Figure 7[3] is that in data points with `exceptions` appear much earlier than their assert counterparts, and most issues are covered at the top of the exceptions. For most of the data points, the displayed position in the log was quite prominent (usually within the first 10 lines).

The required trace length did not directly depend on the size of the program, but rather on the amount of thunks that the program builds up during evaluation. We can see this behavior in Figure 9. Naturally, the `real` data points produce a lot of thunks and evaluations due to their complexity, but some of the spectral and imaginary data points (artificially) produce large amounts of thunks (`spectral/minimax`) or evaluations (`imaginary/rfib`). For a helpful exception, it is necessary that both the start of evaluation and end of evaluation of the involved expressions be in the window of recent evaluations. However, the window should be *as small as possible* - as seen in Table 2; for both `reptile` and `minimax` the position of the faulty statement appears later for a trace length of 1000 compared to the trace length of 50.

[3]Note the log-scale on the x-axis

Table 1: Overview of the `nofib-buggy` programs used

nofib-buggy Error Type	Imaginary	Spectral	Real
Exception	paraffins digits-of-e2	sorting primetest	anna
Assert	digits-of-e1 rfib tak integrate gen_regexps bernoulli wheel-sieve1 wheel-sieve2 x2n1	chichelli fish minimax	gg parser reptile lift

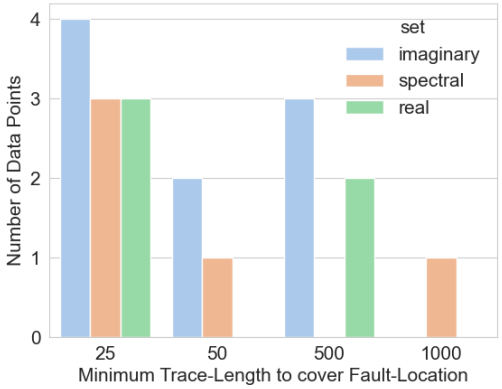

Figure 6: Minimum trace length to cover the error

Performance. We provide a summary of the compute time used in Figure 8 and of the allocated (peak) memory in Figure 10. All reported values are derived from a set of five measured runs on a

Table 2: Summary of `nofib-buggy` results. LOC indicates the location in the output after the initial exception, and minimum trace length the shortest length in which the error location appears out of [25, 50, 100, 500, 1000].

data point	Uses assert	minimum trace length	LOC 50	LOC 1000	LOC 1000 Strict
imaginary					
bernoulli	Y	50	6	6	6
digits-of-e1	Y	500	-	11	21
digits-of-e2	N	25	1	1	-
gen_regexps	Y	-	-	-	-
integrate	Y	-	-	-	-
paraffins	N	500	-	24	24
rfib	Y	500	-	4	7
tak	Y	25	4	4	2
wheel-sieve1	Y	25	2	2	-
wheel-sieve2	Y	50	7	8	-
x2n1	Y	25	2	2	2
spectral					
cichelli	Y	1000	-	36	-
fish	Y	25	3	3	1
minimax	Y	50	28	260	-
primetest	N	25	2	2	2
sorting	N	25	1	1	1
real					
anna	N	25	1	1	-
gg	Y	25	1	1	-
lift	Y	500	-	32	-
parser	Y	500	-	13	19
reptile	Y	25	29	35	94

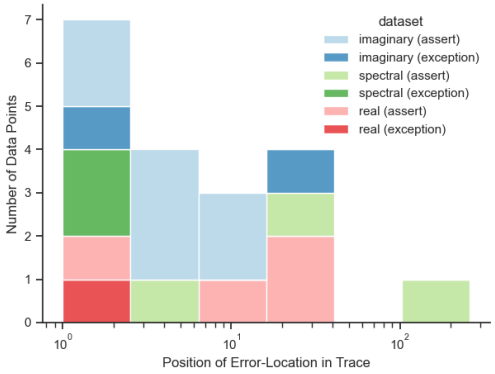

Figure 7: Histogram of position in the trace by data set and error type. Note that the x-axis is logarithmic.

dedicated machine, dropping the highest and lowest values (outliers) and averaging the remaining three. Measurements were conducted with the Linux `/usr/bin/time` executable and the bash `time` command on a cloud-based machine with 32GB of RAM and 6 Intel Xeon E312xx @2GHz 64bit vCPUs. We also performed a set of runs with profiling turned on, using the GHC flags `-fprof`, `-fprof-auto`, and `-fprof-auto-calls`, which yielded comparable increments. As profiling introduces more side effects, we prefer to report the non-profiling numbers in this work. Profile performance measures are included in the companion package.

Figure 8 is a kernel density estimate plot [8] summarizing the distribution of the calculated time deltas for all data points. It presents a smooth growth of wall-clock time to increase the trace length, with the majority of data points needing between ~100% and ~300% longer. We can also observe that *outliers* move respectively, keeping their relative position throughout increasing trace lengths. In particular, the *hungriest* data point was `rfib` from the imaginary data set that needed 760% longer to finish. We take a closer look at `rfib` in the paragraph *limitations*.

The box plot in Figure 10 shows the memory usage and we observe a trend towards slightly higher resource need. The data points in the imaginary set allocate ~15% memory at peak use and the data points in the spectral set ~60% in the median. For the data points in the `real` set the biggest difference was found, with one data point exceeding twice the memory usage. Due to the small amount of data points, and each data point in the `real` set being unique, we don't want to infer general assumptions about the memory usage of these programs.

Our recommendation is to investigate these individually when necessary. We also observe that memory use grows in general with the use of traces, but the size of the trace does not have a huge impact on the preliminary results; the overhead originates from data collection, and not from storing and bookkeeping.

We measured the size of the binary compiled for each program. The difference in most programs was negligible ($\leq 1M$), but we must note that the size increase can be notable for longer trace lengths[4]. For a run gathering *full-traces* (i.e., trace length set to 100k), each binary grew between two and ten Mb.

As traces are used to locate errors, the overhead presented in this work is expected to occur during development and maintenance and will not affect production environments.

Highlights. Among the best results are two data points for the `spectral` data set, `sorting` and `primetest`. The errors are division-by-zero and a non-exhaustive pattern match, respectively. These errors have little information by default, with no location or stack trace. The extended output (see `sorting` in Figure 11) with the trace information that CSI: Haskell adds shows the starting positions where incorrect data was produced and does so quite precisely.

The second group of promising results is demonstrated by the data points for `minimax` and `gg`: the bug introduced to minimax consists of not applying a minimax algorithm to Tic-Tac-Toe, but instead performing a *minimin*. Figure 12 catches this behavior by repeating the `Game:min'` function, while we would expect alternating min and max functions. This is not exactly unique to evaluation traces, but we get "`a bit of coverage for free`". Without

[4]This is due to an in-binary representation of the tick-arrays, to address internal mechanics such as garbage collection. For normal coverage, the addition is bounded by the modules and their expressions, while our additions can vary in length and thus grow the binary to varying degrees.

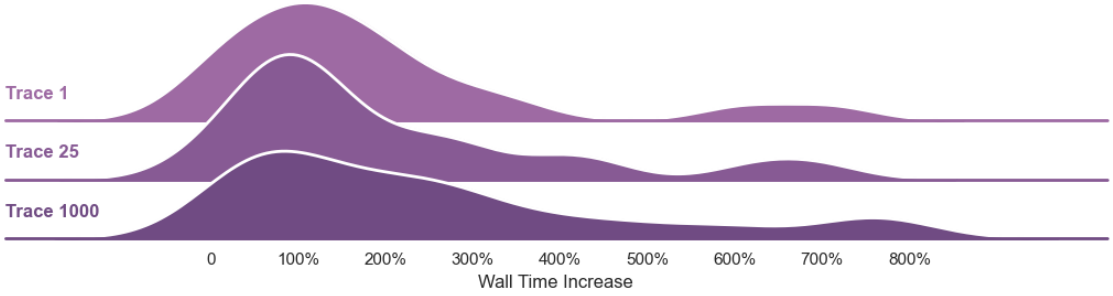

Figure 8: Kernel density estimate plot of increased compute time with varying trace lengths

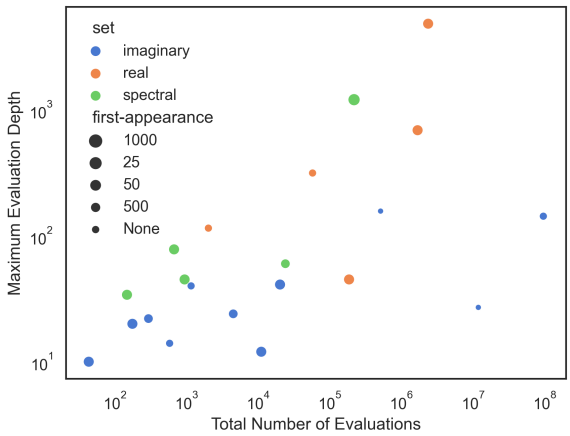

Figure 9: Distribution of maximum evaluation depth and total number of evaluations

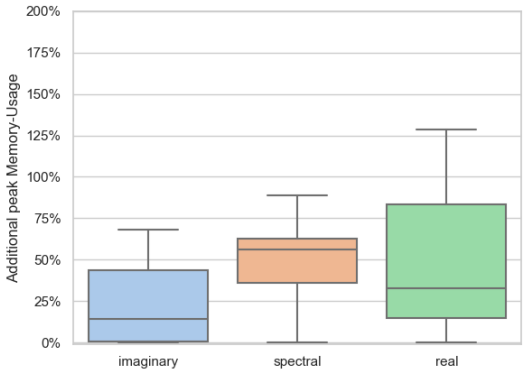

Figure 10: Additional memory usage per data set for a trace length of 1000

enhanced traces, this would also be spotted when running a HPC coverage report and seeing the uncalled max function.

```
Main: divide by zero
Recently evaluated locations:
  ./Sort.hs:146:25-146:25  2
  ./Sort.hs:146:23-146:23  2
  ./Sort.hs:146:22-146:26  (2-2)
  ./Sort.hs:146:14-146:26  k `div` (2-2)
 Previous expressions:
  ./Sort.hs:146:5-146:26   Sort:heapSort>div2
  ./Sort.hs:128:52-128:67  ... =
  repeats (4 times in window):
    ./Sort.hs:128:5-132:84   Sort:heapSort>to_heap
  Main.hs:14:36-14:43      ... =
  Main.hs:13:5-22:57       Main:mangle>sort
  Main.hs:10:1-22:57       Main:mangle
  Main.hs:5:1-7:33         Main:main
There were 668 evaluations in total but only 86 were recorded.
Re-run again with a bigger trace length for better coverage.
```

Figure 11: The improved error log for `Sorting` - the first locations of the trace are the precise consumers and producers of the division-by-zero error[5].

Similarly gg from the `real` data set uses a wrong variable, leaving large parts of a **`where`** block unevaluated.

This second group of bugs can be quickly noticed using program coverage, and it is possible to get the same information from a coverage report. Unfortunately, we must admit that this is an enlightened guess – we knew what was going wrong, and thus we found patterns and clues in the traces. These bugs can be found quite easily when program coverage is visualized, and thus we hope that a visualization of traces would also yield such benefits, motivating more complex tooling.

Before we leave the highlights, we want to emphasize the possibilities of generated traces for mechanical evaluations. Some of the traces presented throughout this paper are a bit crowded or hard to understand, but nevertheless, they contain the information necessary for better fault-localization and other warnings. We see potential tooling that spots mismatches in coverage and evaluation, or that warns about potential performance issues with a lot of thunks, like we see in the `rfib` example.

[5]Note that some of the right hand sides are missing here, due to a mismatch between the locations reported by HPC and the actual location in the file... caused by the mixing of tabs and spaces! Fixing this is beyond the scope of the prototype.

```
Main: Assertion failed
CallStack (from HasCallStack):
assert, called at Main.hs:12:5 in main:Main
CallStack (from -prof):
Main.main (Main.hs:(10,1)-(12,57))
Recently evaluated locations:
 ./Game.hs:32:59-32:59
 ./Game.hs:31:30-31:30
 ./Game.hs:36:23-36:23   e
 ./Board.hs:57:53-57:56  Board:showsPrec
 ./Board.hs:57:53-57:56  Board:show
 ./Game.hs:31:27-31:31   ... =
 ./Game.hs:31:9-33:71    Game:best>best'
 ./Game.hs:32:51-32:65   ... =  s bs ss
 ./Game.hs:32:37-32:47   |..., s') = best,...=...
                                      (was matched)
 ./Game.hs:63:15-63:18   ... = OWin
 ./Game.hs:63:1-68:47    Game:min'
 ./Board.hs:57:70-57:71  Board:(==)
 ./Board.hs:26:33-26:55  ... = [[r1,r2,insert p r3 x]]
 ./Board.hs:23:26-23:46  |...,not (empty pos board),...=...
                                      (not matched)
 ./Board.hs:41:24-41:27  ... = True
 ./Board.hs:39:1-42:18   Board:empty'
 ./Board.hs:36:26-36:36  ... = empty' x r3
 ./Board.hs:34:1-36:36   Board:empty
 ./Board.hs:23:1-26:55   Board:placePiece
 ./Game.hs:31:9-33:71    Game:best>best'
 ./Game.hs:32:51-32:65   ... =  s bs ss
 ./Game.hs:32:37-32:47   |..., s') = best,...=...
                                      (was matched)
 ./Game.hs:63:15-63:18   ... = OWin
 ./Game.hs:63:1-68:47    Game:min'
...
```

Figure 12: The improved error log for `minimax` - notice the repetition of `min'`, without the appearance of a `max'`.

Full Evaluation Record versus Suffix. A thread running through this paper is the initial scenario: is it enough to determine what happened immediately before a crash in order to locate the fault? We consider the recent evaluations the *suffix* of all evaluations. Shown in Table 2, the errors appear in 90% of the data points examined. Furthermore, a relatively short trace of only 50 locations per module is sufficient in 62% of the cases. When running the program in Figure 13, there are **95845589** evaluations in total, of which only **500** were in the final recorded trace, which is enough to cover the faulty location. Despite losing analytical benefits of the complete trace, we are able to locate the fault while keeping only **0.0005217%** of the trace. We thus recommend capturing the only the last evaluations, with a little fine-tuning in trace lengths depending on the number of evaluations (unless necessary for follow-up analysis).

Strict vs. Lazy Behavior. For comparison, we conducted experiments with the `-XStrict` language extension, in addition to the `-fno-strictness` and `-fno-full-laziness` flags to observe changes in evaluation behavior. Without the extension, the trace for each data point was identical, with or without the flags. Our initial (naive) assumption was that for strict programs consumption and production of errors would align, resulting in always perfect locations. The heavy-handed use of the `-XStrict` extension meant that some of the programs would no longer terminate, as many of them rely on laziness to be computable. This resulted in 8 data points that do not finish when strict evaluation is forced.

Among the terminating data points, we see mixed results - `fish` and `tak` perform slightly better, while some evaluations appear later than in their non-strict configuration. We attribute this to the general offset in consumption and production that is also observed in strict & imperative programming languages (e.g., in the work of Zhang et al. [39]): The distance between fault-introduction and fault-consumption also exists in Haskell, but non-strict evaluation can *shrink* the gap between fault-introduction evaluation and fault-consumption evaluation.

To paraphrase, there is always a gap between fault and error, but non-strict evaluation can bridge this gap by postponing evaluations.

Thus, laziness modulates the distance between bug occurrence and consumption. This affects our configuration for the trace lengths: for short traces, a faulty location can be covered but might have been rotated out of the current trace buffer. With long and short traces alike, there is a chance that the location is reported later in the output, missing the user's attention. An example of this is `paraffins`, where sharing is a source from which an incorrect value is evaluated long before it is used. Potentially, this can be further adjusted by introducing more laziness into programs by making other adjustments, such as explicitly disabling sharing [34].

Limitations. The first limitation is represented in `rfib`, which needed a surprisingly long trace for a rather simple program (calculating Fibonacci numbers). Inspecting Figure 13, we observe that the `rfib` program performs a cascading recursion and postpones evaluation, with a lot of redundant re-computation producing a lot of thunks. For our current reporting, it is necessary that the trace length covers a coherent sequence (i.e., covers both creation and resolution of thunks), but this coherence is only perceived when the trace length is long enough. To mitigate this, users are presented with a message when we detect that the trace length is not long enough to cover the entire execution of the program.

We are slightly divided about this topic: On the one hand, many functional pearls utilize recursion and laziness, and thus will trigger a similar behavior for our traces. Especially for these cases, the insights in the evaluation would have great potential for learning and visualization. On the other hand, recursion of this type should usually be written in a tail call-optimized fashion (see Figure 14), which is less graceful but is preferable in performance and also benefits the traces introduced by this work.

```
1 nfib :: Double -> Double
2 -- BUG: The following line contains a bug:
3 nfib n = if n < 1 then 1 else nfib (n-1) + nfib (n-2)
```

Figure 13: nofib-buggy's `rfib`: The code first builds up a large number of thunks using recursion before completing any evaluation, posing a challenge for evaluation traces

Current evaluation traces are also limited by *sharing* [18]. Consider the function in Figure 15: Here, the call to `three_partitions (n-1)` is used in line 3 to generate triples of integers to partition a list. There is an error in this function that

```haskell
fib :: Double -> Double
fib n = fib' n 0 1

fib' :: Double -> Double -> Double -> Double
fib' 0 a _ = a
fib' 1 _ b = b
fib' n a b = fib (n-1) b (a+b)
```

Figure 14: Alternative Fibonacci implementation that utilizes tail-call optimization

```haskell
rads_of_size_n radicals n =
 [(C ri rj rk)
 |(i,j,k)  <- (three_partitions (n-1)),
  (ri:ris) <- (remainders (radicals!i)),
  (rj:rjs) <- (remainders (if (i==j) then (ri:ris)
                                     else radicals!j)),
  rk       <- (if (j==k) then (rj:rjs) else radicals!k)]
```

Figure 15: Part of the paraffins example showcasing *sharing*

causes the *k*s to be invalid out-of-bound indices for the `radicals` list. Since `i` and `j` are used in lines 4 and 5 respectively, the triplets are evaluated and then the *same result* is *shared* later when the invalid `k` is used in line 7. This means that the distance from production to consumption increases due to sharing, which means that there will be more unrelated evaluations prior to the error in the trace. This could be addressed by post-processing traces and removing those evaluations that are "unrelated" (such as those in lines 4 and 5), but this would require a richer view of which values are involved in each expression. This view could possibly be created by adding *provenance* information to the values (see Section 5).

We spare the reader examples for readability, but it is easy to imagine that evaluation traces are not always useful. Mainly we see that traces either don't contain relevant information, or there is a major overhead attached, and we do not expect people to work through 100+ lines of trace information. A prominent example of this issue is `minimax`, for which the fault-location is *covered*, but only in the sense that the relevant statement was touched. It is not immediately clear what to do, as the issue originates from the unused parts of the program. Providing too much information can also discourage developers from reading error messages[28]. and time spent in the wrong places is a waste and reduces trust in traces and error messages[6, 33]. Thus, another crucial improvement is to determine what criteria constitute the relevance of a trace for the problem and only present them when applicable.

Discussion. Based on the limitations and highlights, our current suggestion is to show evaluation traces for certain types of exceptions. The prime candidates are index errors, failed pattern matches, and exceptions for dynamically typed values, such as those from `Data.Dynamic`. These programs showed great results without any real overhead and are a perfect point-in-case for evaluation traces.

From the data points that yield wrong results and have been investigated using assertions, we see a trend that unit-level tests provide better evaluation traces than system-level tests. In particular, the `nofib-buggy/real` data points that use a string comparison for `stdin` and `stdout` did not really benefit from the evaluation traces. We expect that lower-level tests and assertions are far more useful, especially when combined with a sound approach to testing and coverage.

We also recognize the size of the errors and sometimes *mechanic* coverage of traces - as shown in Figure 6, some faults require long traces to be covered, and the resulting output is bound to be verbose. We do not consider these traces to be *actionable* due to their size and the effort necessary to comprehend them. Nevertheless, we hope that the tools can pick up the verbose trace information to further filter and visualize critical elements of the code.

Currently, `Prelude` provides two functions `error` and `errorWithoutStackTrace`. We suggest expanding this to errors with (only) evaluation traces and a combination of stack and evaluation. The choice is left to individual exceptions as to whether evaluation traces make a worthwhile addition. Another necessary step is to provide a starting guide on how to read and use evaluation traces. Typically, people google their exceptions to find some help [6, 26], but with this newly introduced addition, that is not an option. Thus, some kind of central starting point and tutorial should accompany any changes.

5 NEXT STEPS

Evaluation Asserts. A potential new area is the construction of *evaluation asserts* - using the enhanced coverage information, and a known expression in the source code, one can formulate tests that check for the (full) evaluation of an expression. While this comes with some difficulties in implementation (e.g. not evaluating the expression through the assert), there are certain areas where this can support developers: One application of this is in debugging, for which developers might want to check the *state* of their variables. Although this is not exactly in the spirit of functional paradigms, existing research [11] shows that Haskell developers often fall back to imperative approaches during debugging. Furthermore, we face functions such as `foldr`, `foldr'` and co. whose results are identical, but their internal traversal strategies differ. Another application is for systems that revolve around or provide evaluation strategies such as GHC itself. It can provide capabilities to test, e.g. BangPatterns and data structures.

Study. An obvious next step is a detailed study. The examples presented in this work highlight initial results but hardly represent *the real world*. Thus, the authors plan to conduct a broader study utilizing most of the `nofib-buggy real` data points and modern examples from the HasBugs data set [4]. Such a study should help to grasp how often evaluation trace information covers bugs and, if so, how long the trace should be.

Furthermore, a study is necessary to estimate the computational feasibility. Additional instrumentation always comes with a performance cost, and the exploration in `nofib-buggy` is unfortunately not representative of a complete evaluation.

Provenance of Values. One problem that arises when strictness and sharing are involved is that an expression might have been evaluated long before usage, such as the `k` in the paraffins example (Figure 15). This means that many unrelated evaluations occur between the production and consumption of a value, making the

trace less useful to find the source of the error. One way to address is to attach *provenance* to values, highlighting the part of a trace involved in the production of any values touched on in an error.

Environment Integration and Presentation. This work presents basic steps and low-level implementation for evaluation traces, but the findings might be *diamonds in the rough*. Especially for longer traces of the `real` data points, guidance and assistance are necessary. We touched on potential tools and extensions throughout the work, which we would like to summarize:

First, summarizing and filtering traces is necessary to keep the output human-readable, especially for long traces. Solutions could cover filtering modules or limit the depth and width of the presented evaluation tree. In addition to trace data, there are opportunities to accumulate data from multiple sources (test success, program coverage, etc.) and perform program slicing [37]. This is essential to scale to large programs. Another important integration is with test and build frameworks. At the moment, traces are reported on runtime exceptions, which is arguably not the best state of a program to be in. Most of the time, software engineering utilizes tests, and therefore evaluation traces should be presented in an accessible way for test failures. We hope that in the future, Haskell developers can write unit tests and investigate their evaluation for anomalies, finding potential issues before they become problems. Lastly, we did early sketches of integrating SARIF[3] based on `tix`- and `mix`-files with a prototype. Transforming the information is quite easy and can then be picked up from other popular tools such as VSCode. Especially in light of the Haskell Language Server that also targets VSCode, we hope that representation of coverage and evaluation in the IDE can be a result of this work. However, such tools should not only be based on solid data (this work), but must also meet standards and needs of developers, drastically expanding the scope. Thus, this work focus‘ lies on the creation, maintenance and mapping of evaluation-traces.

Automated Fault Localization. Although this work covers fault-localization as a manual task, automated fault localization is a popular research topic with often great results [1, 17]. Automated fault localization often exploits a spectrum of coverage per test to find code that is *suspiciously often* involved in failing tests. These approaches are based on the program coverage of strict languages (Java, C), and revolve around expression or statement coverage. Directly copying these approaches might not be applicable to Haskell — due to laziness, we might call expressions but not evaluate them. Thus, focusing on evaluation over coverage is necessary to build a spectrum of code that was executed, and not only touched.

Apart from adjustments necessary to reproduce existing approaches, the evaluation information can also form the basis of novel techniques: normal spectrums are *binary*, things are covered or not. With evaluation, we express the concept of full or partial evaluations and can derive a continuous spectrum.

Optimization. We are aware that this is merely a prototype implementation. We hope that producing a non-invasive method for gathering and reporting information on evaluation resonates positively in the community, but know that we have made some arbitrary design decisions. In this spirit, we do not consider the implementation *done* but are looking forward to feedback on this work and towards an eventual GHC proposal.

Required trace length estimation. One pain point with the current design of CSI: Haskell is that the trace length is fixed and a value must be provided by the developer. One way to address this could be to have a more dynamic trace, discarding entries not involved in the current evaluation and keeping only the parts of the trace which involve values which are currently accessible and have not been garbage collected. This would involve a much deeper integration with the runtime system and memory management, but could be vital for tracing long-running programs, keeping both relevant parts of the trace but still keeping memory requirements manageable. Another approach would be to do static analysis of the program to suggest a useful length for the trace, using the call graph and structure of expressions to approximate the required length within some order of magnitude. However, this would involve more advanced termination checking than feasible for this paper, but would reduce the guesswork in finding a good length. In the interim, we suggest using a trace length of approximately 100 for smaller programs and approximately 1000 for larger ones (as suggested by our experiments on the `nofib-buggy` data set) and increase or decrease as necessary.

6 CONCLUSION

This paper presented an initial implementation to gather evaluation traces and report them alongside current stack traces on runtime exceptions. The approach utilizes *boxes* similar to regular HPC and only requires additional flags for compilation — extending from the program even into dependencies. This novel data was used to improve the runtime exceptions reported with information on the evaluation. We ran the changes on a subset of the `nofib-buggy` data set, investigating at which point of the trace the faulty location was reported. For 19 of the 21 data points, the fault was covered in a trace of 1000 feet long, and most of the locations appeared in the first 50 lines of the trace. In general, valuable information is covered by the trace, but a current limitation is the size and verbosity of the output. Most data points required two to 2 to 3 times more runtime and about 50% more memory. Outliers in performance were based on excessive amounts of thunks and a large number of modules.

Providing evaluation traces can help to spot certain errors, especially those related to lazy evaluation. The examples provided in this paper show that evaluation traces help to establish the chain of events behind certain errors better than a plain stack trace, as due to lazy evaluation the origin of a problem and its occurrence can be offset.

ACKNOWLEDGMENTS

We thank David Sands for his input on evaluation trees and theory, and Matthew Sottile for his efforts on visualization and advice on the design of CSI: Haskell. We thank the attendants of the IFL workshop for their input, particularly the concept of evaluation asserts. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knuth and Alice Wallenberg Foundation.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, IEEE, Windsor, UK, 89–98.

[2] Tristan O.R. Allwood, Simon Peyton Jones, and Susan Eisenbach. 2009. Finding the Needle: Stack Traces for GHC. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh, Scotland) *(Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/1596638.1596654

[3] Paul Anderson, Łucja Kot, Neil Gilmore, and David Vitek. 2019. SARIF-Enabled Tooling to Encourage Gradual Technical Debt Reduction. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE/ACM, Montreal, QC, Canada, 71–72. https://doi.org/10.1109/TechDebt.2019.00024

[4] Leonhard Applis and Annibale Panichella. 2023. HasBugs - Handpicked Haskell Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, Melbourne, Australia, 223–227. https://doi.org/10.1109/MSR59073.2023.00040

[5] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, IEEE/ACM, Buenos Aires, Argentina, 575–585.

[6] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[7] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 308–318. https://doi.org/10.1145/1453101.1453146

[8] Yen-Chi Chen. 2017. A tutorial on kernel density estimation and recent advances. *Biostatistics & Epidemiology* 1, 1 (2017), 161–187. https://doi.org/10.1080/24709360.2017.1396742 arXiv:https://doi.org/10.1080/24709360.2017.1396742

[9] Olaf Chitil, Colin Runciman, and Malcolm Wallace. 2002. Transforming Haskell for tracing. In *Symposium on Implementation and Application of Functional Languages*. Springer, Springer, Madrid, Spain, 165–181.

[10] Maarten Faddegon and Olaf Chitil. 2015. Algorithmic debugging of real-world haskell programs: deriving dependencies from the cost centre stack. *ACM SIGPLAN Notices* 50, 6 (2015), 33–42.

[11] Kasra Ferdowsi. [n.d.]. Towards Human-Centered Types & Type Debugging. Plateau Workshop.

[12] GHC Contributors. 2021. GHC 8.10.4 users guide. https://downloads.haskell.org/~ghc/8.10.4/docs/html/users_guide/index.html

[13] Andy Gill. 2000. Debugging Haskell by Observing Intermediate Data Structures. *Electron. Notes Theor. Comput. Sci.* 41, 1 (2000), 1.

[14] Andy Gill and Colin Runciman. 2007. Haskell Program Coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Freiburg, Germany) *(Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1291201.1291203

[15] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019. Lazy Counterfactual Symbolic Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 411–424. https://doi.org/10.1145/3314221.3314618

[16] A. Hamou-Lhadj and T. Lethbridge. 2006. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, Athens, Greece, 181–190. https://doi.org/10.1109/ICPC.2006.45

[17] Tom Janssen, Rui Abreu, and Arjan JC Van Gemund. 2009. Zoltar: A toolset for automatic fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, IEEE, Auckland, New Zealand, 662–664.

[18] John Launchbury. 1993. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, Charleston, SC, USA, 144–154.

[19] Simon Marlow. 2012. *HIW 2012: Why can't I get a stack trace?* ACM, Copenhagen, Denmark. https://www.youtube.com/watch?v=J0c4L-AURDQ

[20] Lee Naish and Tim Barbour. 1996. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications* 18 (1996), 401–408.

[21] Henrik Nilsson and Jan Sparud. 1997. The evaluation dependence tree as a basis for lazy functional debugging. *Automated software engineering* 4 (1997), 121–150.

[22] Kunihiro Noda, Takashi Kobayashi, Tatsuya Toda, and Noritoshi Atsumi. 2017. Identifying Core Objects for Trace Summarization Using Reference Relations and Access Analysis. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, Turin, Italy, 13–22. https://doi.org/10.1109/COMPSAC.2017.142

[23] Will Partain. 1993. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992*. Springer, Springer, Ayr, Scotland, 195–202.

[24] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, Espoo, Finland, 41–50.

[25] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, Tacoma, WA, USA, 74–82.

[26] Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K. Roy. 2014. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, Antwerp, Belgium, 194–203. https://doi.org/10.1109/CSMR-WCRE.2014.6747170

[27] Ranjit Jhala. 2020. LiquidHaskell is a GHC Plugin. https://ucsd-progsys.github.io/liquidhaskell-blog/2020/08/20/lh-as-a-ghc-plugin.lhs/

[28] Peter C. Rigby and Suzanne Thompson. 2005. Study of Novice Programmers Using Eclipse and Gild. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange* (San Diego, California) *(eclipse '05)*. Association for Computing Machinery, New York, NY, USA, 105–109. https://doi.org/10.1145/1117696.1117718

[29] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE working conference on mining software repositories (MSR 2010)*. IEEE, IEEE, Cape Town, South Africa, 118–121.

[30] Peter Sestoft. 1997. Deriving a lazy abstract machine. *Journal of Functional Programming* 7, 3 (1997), 231–264.

[31] Dale Shaffer, Wendy Doube, and Juhani Tuovinen. 2003. Applying Cognitive load theory to computer science education.. In *PPIG*, Vol. 1. Citeseer, M. Petre & D. Budgen (Eds.), Keele, UK, 333–346.

[32] Josep Silva. 2007. *The Buggy Benchmarks Collection*. Universitat Politecnica De Valencia. Josep Silva self-published on his website / university.

[33] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Adv. in Hum.-Comp. Int.* 2010, Article 3 (jan 2010), 26 pages. https://doi.org/10.1155/2010/602570

[34] Marco Vassena, Joachim Breitner, and Alejandro Russo. 2017. Securing Concurrent Lazy Programs Against Information Leakage. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, Santa Barbara, CA, USA, 37–52. https://doi.org/10.1109/CSF.2017.39

[35] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 269–282.

[36] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. 2001. Multiple-View Tracing for Haskell: a New Hat. In *2001 ACM SIGPLAN Haskell Workshop*, Ralf Hinze (Ed.). Firenze, Italy. https://kar.kent.ac.uk/13566/ Universiteit Utrecht UU-CS-2001-23. Final proceedings to appear in ENTCS 59(2).

[37] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 1, 4 (1984), 352–357.

[38] John Wrenn and Shriram Krishnamurthi. 2017. Error Messages Are Classifiers: A Process to Design and Evaluate Error Messages. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 134–147. https://doi.org/10.1145/3133850.3133862

[39] Zhenyu Zhang, W. K. Chan, T. H. Tse, Bo Jiang, and Xinming Wang. 2009. Capturing Propagation of Infected Program States *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/1595696.1595705