How do Haskell programmers debug?

Ruanqianqian (Lisa) Huang , Elizaveta Pertseva , Michael Coblenz and Sorin Lerner

University of California San Diego, La Jolla, CA

Abstract

Functional programming is a paradigm that emphasizes avoiding shared mutable state. Compared to imperative programming, which focuses more on how the runtime state should be updated to perform a computation, functional programming adopts a more declarative syntax that highlights what the computation should achieve without involving mutable state. Since functional programming differs from imperative programming, programmers might adopt different debugging strategies in functional programming. However, how programmers debug in functional languages remains under-explored. As an initial step, we interviewed four expert functional programmers to gain insight into how they debug in Haskell, a popular functional programming language. Our preliminary findings show that while debugging strategies for Haskell are similar to strategies for other languages, some features of Haskell and functional programming introduce challenges to using these debugging strategies. Informed by the findings, we call for future work that gains more understanding of how debugging is done in Haskell and functional programming in general, and explores design opportunities for usable debugging aids in this domain.

Keywords: Functional programming. Debugging. Haskell.

1 Introduction

Functional programming emphasizes avoiding shared mutable state. Therefore, it frees programmers from worrying about state changes [1]. Hughes further argued that functional programming leads to well-structured programs and delivers benefits to constructing large, complex software [2].

Functional programming is different from imperative programming, which calls for code that specifies mutations in runtime state. Because of the difference, one might wonder whether debugging in functional programming requires different strategies than debugging in imperative programming. However, while there has been a fruitful line of research in how debugging is performed with imperative languages [3]–[6] and how programmers interact with debugging aids [6]–[10], little about debugging is known in the context of functional languages [11], [12]. In addition, some functional programming languages allow for lazy evaluation or *laziness*, which delays the evaluation of an expression until its value is needed, and results in efficiency in execution but nondeterminism in the order of code execution [13]. Even though laziness is a major feature of not only functional languages but also imperative languages such as R [14], while Hall and O'Donnell discussed how laziness might impede the use of conventional debugging techniques for lazy languages [15], [16], we are not aware of any empirical evidence to date on how laziness might affect debugging.

To understand what debugging strategies are employed in functional programming and whether the unique features of functional programming affect debugging, as a first step, we interviewed four expert Haskell programmers about their past experience in debugging Haskell programs. We chose Haskell as the subject of study due to the following reasons: (1) it is similar to other functional languages [17]; (2) it was rated as the second most popular functional language as of May 2022 (almost as popular as Scala) [18]; and (3) it is the only functional language with default laziness. Our study intends to address the following research questions:

RQ1: What strategies do people use to debug in Haskell?

RQ2: What distinguishes debugging in Haskell from debugging in other languages?

Our preliminary findings suggest that while debugging strategies for Haskell are similar to strategies for other languages, some features of Haskell and functional programming cause challenges in adopting these debugging strategies. Guided by the results, at the end of the paper we also call for future work that collects more evidence about how debugging is done in Haskell and functional programming in general, and explores the design space for usable debugging aids in this domain.

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

Table 1. Sumary of participants' demographics.
For the column **Types of Haskell Experience**, "Personal" refers to Personal Side Projects.

Participant	Occupation	Years of Programming Experience	Years of Haskell Experience	Types of Haskell Experience
P1	Undergraduate Student	8 - 9 years	8 years	Personal, Class, Open Source
P2	Ph.D Student	17 years	13 years	Research, Personal, Class, Teaching
P3	Ph.D. Student	14 years	10 years	Research, Personal, Teaching
P4	Ph.D. Student	10 years	5 years	Research, Personal, Industry

2 Methodology

To gain insight into how expert Haskell programmers debug, we interviewed four participants (three male, one female).

Recruitment. We recruited participants from students in the Computer Science and Engineering department of our institution. We screened participants based on their self-reported Haskell experience and their answers to a 15-minute Haskell proficiency survey. The survey questions included tasks of code comprehension, writing, and debugging, and some questions about Haskell types, monads, and higher order functions. All of our participants had at least five years of experience in Haskell and answered every question correctly. Each particpant was compensated with $15 for completing the study.

Participants. We interviewed four participants: three graduate students (P2, P3, P4) and one undergraduate student (P1). P1 took a graduate-level Haskell course, P2 and P3 have been teaching assistants for a graduate-level Haskell course several times, and P4 had two industry experiences in Haskell. All four participants were first exposed to imperative languages, but Haskell was the first functional language they learned. Further demographics are summarized in Table 1. Three of the interviews were conducted in person (P1, P3 and P4) and one using Zoom (P2).

Procedure. Participants were asked to complete a 45-minute semi-structured interview. We divided our questions into three categories: programming background, sentiments towards functional programming and Haskell, and Haskell debugging experience. For each category we had a list of questions, and the interviewers were allowed to ask questions out of order or ask follow-up questions. The questions were formulated according to the research questions and attempted to identify both the causes of the difficulties participants faced during debugging and possible solutions that address them. Specifically, the questions in the last category focused on anecdotal examples of Haskell bugs, strategies to approaching Haskell bugs, prior experience with Haskell debugging tools, and desires for additional Haskell debugging support. All of the interviews were done by the first two authors. After the interview, each participant was asked to complete a short demographics survey.

Data Collection. The data collected from each participant consisted of an audio recording of the interview and demographics information from the post survey, which included the participant's gender, details of their programming experience, and math background. We transcribed the audio recordings using a transcription service. The transcribers were provided with a domain-specific glossary including but not limited to: Haskell, recursion, OCaml, IDE, and GHCi [19]. Afterwards the first two authors manually went through the final transcripts and corrected any discrepancies they found, seeking to minimize transcription error.

Analysis. To analyze the data, we conducted a thematic analysis [20]. The first two authors constructed codes using a bottom-up approach, making sure they agreed upon every code. They then derived and iterated on themes from the codes. We concluded our analysis with several themes, which we report as paragraph titles in Sec. 3.

3 Results

We elaborate on how our findings may address our research questions (RQs), and, from the findings, derive hypotheses to be validated in future research. In general, we hypothesized that Haskell programmers consider seven main strategies when debugging, and that some features of Haskell and functional programming impede the execution of these debugging strategies.

3.1 RQ1: Debugging Strategies for Haskell

To understand the debugging processes participants used when debugging Haskell programs, we asked the participants to walk us through a recent debugging session in Haskell. We asked them what strategies they used to locate and find the specific bug(s) in the given scenario, and followed up with questions of whether they used similar approaches for other debugging sessions in Haskell they had encountered. Although the bugs reported by the participants occurred in scenarios ranging from pretty-printing the content of a directory (P4) to implementing language interpreters (P3), we found that their debugging processes consisted of seven main strategies, which we describe below.

Running Program with Test Inputs. All participants suggested that after some code was written, the first reaction was usually to run the code against some test inputs, although some test inputs that depend on user input are hard to replicate. A test input, for example, could be one simple test case that the programmer assumes to cover all parts of the program to be tested (P1, P2, P3), or a test suite already written by other programmers if the code comes with starter code or is part of a collaborative codebase (P4). This step could be repeated during the debugging process, and P2 would specifically "[try] to make a minimal example of the input that would trigger the bug." However, P1 mentioned that it is generally hard to automate and replicate test inputs for I/O events.

Forming or Updating Mental Models. Participants claimed that they only started to thoroughly reason about (or update, if they were already in the middle of debugging) their mental models of the program execution until after they obtained some results from running the program. P1 explicilty mentioned that they would "look at the result, look at [their] code, [and] try to understand why [they] got [the run results]." Note that in our study, all participants reflected upon scenarios of debugging code written by *themselves*. It is possible that some vague mental model of program execution might have been already made simultaneously when the code was being written, as Katz and Anderson suggested [11]. We do not have enough data to explain whether such a vague mental model existed before the debugging session starts, which we leave to future work.

Singling out and Running Faulty Segments. To identify the locations of the faults [21], participants mentioned that they could pull out problematic subcomponents of the program and rerun the same test inputs against them until the fault(s) were found, but the approach would better work with code written separating concerns (P1, P4). P1 further pointed out that it is generally hard to separate and run faulty code that is part of some monadic code. Participants also used different approaches to test these segregated functions against the test inputs. P1, P3 and P4 would load the functions into GHCi [22], a REPL for Haskell, although P4 would rather rerun the whole program than using GHCi when the program came with a set of well-setup test cases. P2 did not use GHCi as much and usualy "just [compiled] and then [ran] the actual program.

Adding Code to Print. Participants suggested that they would sometimes add code for printing to reveal intermediate values that might go against their expectations, but printing is difficult in Haskell. Listing 1 shows the definition of a function `makeEven` that takes an argument `n`, and Listing 2 and Listing 3 illustrate how the code should be modified (highlighted in yellow) to print out the argument at the beginning of the function. Before printing, the expression has to be a `Show` instance, for which a `show` function (that creates a string representation of the instance) needs to be implemented. In our example, the expression `n` is an `Int` type, which is already a `Show` instance. Listing 2 shows one approach to print `n`: one could call `print n`, which is equivalent to `putStrLn (show n)` [23], with the premise that the code that calls `print n` is an `IO` action. Listing 3 shows another approach using functions such as `trace` from the `Debug.Trace` library [24]. Either approach of printing in Haskell requires the programmer to first ensure that the expression to be printed is a `Show` instance and implement the `show` function for it if needed, then decide between using (1) `print`, which may

ask for additional refactoring for the code to produce an `IO`, or (2) functions from the `Debug.Trace` library, which requires no `IO` actions but proper insertions of those trace calls. As such, we find printing in Haskell to be tedious, requiring a non-trivial amount of code insertion that results in code refactoring, as is confirmed by most participants (P1, P2, P3).

```
makeEven :: Int -> Int
makeEven n =
  case (even n) of
    True -> n
    False -> n+1
```

Listing 1. Without printing.

```
makeEven :: Int -> IO Int
makeEven n =
  do
    print n
    case (even n) of
      True -> return n
      False -> return (n+1)
```

Listing 2. Printing with `print`.

```
import Debug.Trace

makeEven :: Int -> Int
makeEven n =
  trace (show n) $
  case (even n) of
    True -> n
    False -> n + 1
```

Listing 3. Printing with `trace`.

Forming an Explanation for the Fault. Based on the results they saw from testing faulty segments of the program or printing intermediate values, participants would attempt to explain to themselves what resulted in the overall faulty behavior. While this step seems straightforward, participants said that the mental reasoning occurred purely in their heads, and they would need to "[think] about it really hard until [figuring] out what the problem was." (P2) P1 deemed the process as "re-understanding the problem" that the code intended to solve, and if the process failed, "we cry a little," said P3.

Attempting Fixes. Although not directly mentioned by the participants, their description of the debugging process suggested that they would attempt to fix the code based on their hypotheses of what caused the fault(s). To confirm that their fixes work as expected, they would run the full program with the test inputs they had used, and repeat the rest of the process to "iteratively improve [the code] until [they] actually get it to the point where it produces the right output." (P4)

Using External Resources. Apart from the steps involved in debugging, participants also reflected on their use of external resources when debugging. All participants said that they used one main form of external resource when debugging: documentation for library functions that were part of their code. P1 further stated that unofficial resources such as StackOverflow would be useless when debugging, while acknowledging its usefulness in code authoring. As for external Haskell tooling, all participants commented on existing Haskell debugging tools (more in Sec. 3.2), P4 mentioned `ghcid` [25], which automates re-compilation every time the code is modified, and P1 and P4 claimed the usefulness of the Haskell Language Server [26], which provides more information through program analysis that could facilitate debugging.

> ***Hypothesis:*** *Haskell programmers consider seven main strategies when debugging: running program with test inputs, forming/updating mental model, singling out and running faulty segments, adding code to print, forming an explanation for the fault, attempting fixes, and using external resources.*

3.2 RQ2: Debugging in Haskell vs. Other Languages

Based on our interviews, we identified four key features that distinguish debugging in Haskell from debugging in other languages: laziness, the lack of good debugging support, the type system, and the declarative syntax and abstractions.

Laziness. Participants felt that the presence of laziness complicates debugging in Haskell. Laziness allows the evaluation of an expression to be delayed until it is needed, and benefits performance optimization (detailed in Sec. 5). It is default in Haskell but optional in other functional languages. The default laziness in Haskell makes it difficult for even experienced programmers to predict program behavior during development [17]. In addition, insertions of code for printing expression values can cause code evaluation order to vary significantly from one run to the next. For example, our example in Listings 1, 2, and 3 shows that without printing, the evaluation of `n` is not needed until `even n` is called (Listing 1), but printing `n` forces its evaluation (Listings 2 and 3) and thus changes the order of evaluation. Our participants experienced similar struggles during debugging. Indeed, laziness "makes

figuring out what [is] going wrong much less straightforward than [one] would like it to be" (P2) and prevents programmers from "stepping straight through a program" (P3), causing a mismatch between programmers' expectations of order of evaluation and the reality. P4 mentioned an existing mechanism to enforce non-lazy evaluation, but doing so brings in the risk of changing the overall behavior of the entire program, while non-lazy evaluation is only needed for specific debugging scenarios.

Despite struggling with laziness when debugging, all participants appreciated laziness in development for its help with performance (P3) especially when the code involves data structures that contain thunks (P1), which are subroutines used for lazy evaluation [27]. Therefore, instead of eliminating laziness, all participants expressed a desire for debugging support that handles laziness better. To better handle laziness in debugging, one solution is to show the existence of initialized but unevaluated structures (P1, P3), and another is to show the progress of lazy evaluations on partially evaluated data structures (P1).

> ***Hypothesis:*** *While default laziness is appreciated in development, in debugging it makes it challenging to predict the order of execution especially when printing is used, and causes confusion.*

Lack of Good Debugging Support. All participants stated that Haskell has exceptionally bad debugging support, which further distinguishes it from other languages regarding debugging. Despite having extensive Haskell experience, participants have only heard of the GHCi debugger [19] even though other debuggers exist (e.g., Hat [28]). Only two participants (P1, P3) have used the GHCi debugger [19] (the most recent attempt in an imperative-style debugger) and two were dissuaded from using it by their peers (P2) and managers (P4). P1 and P3 reported that people without sufficient expertise with the debugger would find it very difficult to use and that the debugger would not improve their debugging experience. P1 considered the step-wise experience offered by the GHCi debugger to be unintuitive. They suggested that showing how the code has been executed step-by-step only helps when the code is imperative, which is not the case for a Haskell program, and so "linearizing [the execution of non-imperative code] in some arbitrary way and then stepping through that is not helping you." Participants also mentioned that the GHCi debugger either does not give enough information about the program or gives information that is too fine-grained to be helpful. For example, P1 stated that the GHCi debugger would show that the execution had reached inside a bind function, but this information does not help resolving faults in the program.

Reflecting on their experience with debuggers for imperative languages, participants wished to have similar debugging support in functional programming in general: the ability to pause and terminate the program execution (P4), to enforce the order of execution (all participants), and to step through the execution (all participants). Although a survey of existing imperative debugging tools is beyond the scope of our study, participants were even slightly unsatisfied with the level of control provided by existing debuggers for imperative languages. P4 particularly would rather use printing mechanisms instead of debuggers to debug regardless of the paradigm. Nonetheless, in functional programming, all participants claimed that they would use a debugger if better debugging support were available, especially one that could help them visualize lazy execution.

> ***Hypothesis:*** *Haskell programmers tend not to use debuggers despite their existence, possibly due to their steep learning curve and potential lack of usability. There is a desire for better debugging support for Haskell that allows for more control in debugging.*

Type System. Participants also suggested that Haskell's strong type system distinguishes debugging in Haskell from debugging in other (typed) languages in that the type system both aids and impedes debugging. On the one hand, the type system catches some errors at compile time, allowing participants to quickly iterate and refine their implementations before running the program. This finding matches existing literature [29]. In fact, participants do not consider resolving type errors to be part of the *debugging* process but view it as part of the *development*. P3 particularly stated when writing code, they "always [hit] type errors and fix them" while debugging is a separate process that does not involve type errors. On the other hand, type errors are not always intuitive. After years of experience,

participants no longer struggle with compiler error messages but acknowledge that the learning curve for Haskell compile error messages was very steep.

Furthermore, Haskell supports custom types, the implementation of which could come with code inserted by the compiler. Nevertheless, the type system might impose additional burden on the programmer when implementing custom types, and even cause the compiler-inserted code to introduce unexpected bugs. First, the type system enforces that the custom types become `Show` instances so that printing mechanisms can be applied when needed (detailed in Sec. 3.1), forcing the programmer to extend the existing implementation of custom types before the instances can be printed during debugging (P3). Second, P2 pointed out that while the compiler can, based on type inference, automatically generate some functions for custom data types of which the instances could be `None`, these functions crash when called on the `None` instances, causing more defects for the programmer to resolve. Finally, P4 mentioned that sometimes passing type checks lured them into a false sense of security, as their confidence with Haskell's strong type system could lead to the false assumption that passing compilation checks guarantees being free of logical and runtime errors. Participants' comments on how Haskell's type system affects their debugging experience suggest the following:

> ***Hypothesis:*** *The Haskell type system can be a double-edged sword in debugging, and even expertise with the type system is not sufficient for the programmer to fully leverage its benefits.*

The Declarative Syntax and Abstractions. When asked how debugging in Haskell differs from an imperative context, participants appreciated the declarative syntax and support for abstractions of Haskell, but they stated that these features bring challenges to debugging. They believed that their observations can be generalized to other functional languages.

Participants liked that Haskell's declarative syntax and support for abstractions help them focus on *what* to achieve in the program as opposed to *how* to achieve the steps. These features make functional programming in general "a more natural way to write programs" because it "matches up with how [one thinks] about solving problems" without forcing them to consider "the mechanics of how to do things." (P2)

However, Haskell's declarative syntax and support for abstractions eliminate from the code *how* the program achieves its output. When debugging, participants said that they *do* want to see *how* the computations are done, but such information is not available in code written in functional languages including Haskell, which carry a declarative syntax. P3 frequently asked themselves a question when debugging: "how did I get here?" P1 also pointed out that the lines of code written in an imperative language "correspond to [their] intuitive breakdown of how [the code] does stuff," while in a functional language "their understanding of. . . the fundamental pieces" are not embedded in the code.

In addition, code readability is critical for comprehension and thus debugging [30], but participants worried that Haskell's abstractions (e.g., Algebraic Data Types and Typeclasses) reduce code readability and thus impede debugging. P4 particularly disliked the reduced code readability caused by abstractions, as one would need to keep track of the specifications of any abstractions they created somewhere in their working memory, such as "single line functions that do incredibly non-trivial things. . . in 80 characters," which was echoed by P3. To alleviate the additional burden on their working memory when working with abstractions, P1 and P2 used meaningful names for type constructors (e.g., `data LeafOrNode a = Leaf | Node (LeafOrNode a) a (LeafOrNode a)` that represents a binary tree) to embed some, if not all, of the abstracted information. Still, additional effort is required to cope with Haskell's abstractions, and the extra load it takes on the working memory would only lead to more frustration in debugging.

> ***Hypothesis:*** *While Haskell's declarative syntax and support for abstractions provide benefits in code authoring, they can also impede debugging.*

3.3 Threats to Validity

Our work is a step towards a more thorough investigation, and the preliminary results imply hypotheses that need to be validated in future work to further address the following threats to validity:

Participants. We recruited a small sample of participants from the same department of one institution, all of whom are currently students. We mitigated this limitation of recruitment by only recruiting and screening expert Haskell programmers with a variety of experience in the language, who provided us with a valuable first peek into debugging in Haskell leveraging their past experience.

Study Setup. We only gathered anecdotes of debugging code written by the participants themselves but not code written by others, while debugging code written by others might lead to a different debugging experience than debugging their own code [5], [11], [31]. We leave it to the future work for addressing this limitation. The findings are also based on participants' reflection on their past debugging experience, the details of which could be missing or inaccurate, bringing noise to the data collected. To alleviate the threat, the interviewers asked follow-up questions when they felt that clarifications on the anecdotes were necessary.

4 Discussion

Our findings show that Haskell programmers use several approaches to resolve bugs, and consult external resources when needed to facilitate the debugging process. The findings indicate that, when debugging their own code, strategies adopted by expert Haskell programmers are similar to strategies mentioned in the literature [3], [5], [11], [32]: programmers use an isolation-based strategy [31], making/updating assumptions of bug locations and causes using the information obtained from the edit-run cycles [33] (which includes results from printing and consulting external resources), attempting fixes based on the assumptions, and iterating this process until the bugs are resolved. Connections with existing literature suggest that when debugging Haskell programs, programmers consider similar strategies to the debugging strategies used for other languages. Nevertheless, our preliminary study only reveals that programmers adopt these debugging strategies individually but does not suggest the *ordering* of these strategies, while existing literature indicates that the use of some strategies depends on the results of using other strategies. We leave it to future work to gather more evidence to synthesize a full picture of the debugging workflow of Haskell programs and how the strategies interact with one another in the workflow.

Furthermore, our study reveals several features of Haskell that impede the execution of some of the identified debugging strategies. While both our participants and respondents of the 2005 GHC survey [19], [34] pointed out that debugging in Haskell is challenging due to its laziness, lack of good debugging support, and type system, our participants further found Haskell's declarative syntax and support for abstractions confusing when debugging, which also applies to other functional languages. In particular, abstractions can be powerful when carefully crafted, but programmers might need to remember additional information regarding the components that have been abstracted, which adds to the difficulty in debugging. Future research is needed to further understand the challenges in debugging that are caused by language- and paradigm-dependent factors, in Haskell and functional programming in general.

5 Related Work

5.1 Functional Programming and Haskell

Functional programming is a paradigm that emphasizes avoiding shared mutable state. Example functional programming languages include Lisp [35], OCaml [36], and Haskell [37]. Hughes further argued that the paradigm results in well-structured programs, which is beneficial for building complex software [2]. As such, functional programming has gained popularity in both academia and industry [38].

Haskell is a functional language that first appeared in 1990 [39], rated as the second most popular functional language as of May 2022 [18]. Hudak et al. provided an extensive review of its history, technicalities, tooling, and impact [17]. *Laziness* or lazy evaluation refers to the ability to delay evaluations of expressions until absolutely needed, preventing performance overheads and nonterminating code [2]. Haskell has laziness by default, which Hudak et al. considered to be one of its highlights, the other one being its strong type system [17]. Haskell is purely functional, but it also uses monads [40] to model effects that might involve computations typically done in an imperative manner and with side effects, such as I/O operations. Such monadic code, as a result, may add to the complexity

of debugging in Haskell, as it "can be difficult to read." [40] As is noted by Hudak et al. [17], conventional debugging approaches to tracing and printing (which are "side effects" to be done with monads) are rather hard to apply, and that debugging support for Haskell is itself a separate line of research (such as the GHCi debugger [19], more in Sec. 5.2 below).

Our work concerns how programmers debug when using functional programming languages. In particular, while laziness is an important concept in some functional languages and popular imperative languages such as R, the usability of laziness is barely studied [14], let alone its potential impact on debugging. Furthermore, research in human factors in functional programming is still limited. Despite the existence of evidence in how functional programmers *author* code [41], we have little knowledge of how functional programmers *debug* their code except for one study with novice Lisp programmers [11]. Our work is a step towards bridging this gap in the literature: to better understand how debugging is done in functional programming, and to what extent the unique features of functional programming affect the debugging process.

5.2 Debugging

Debugging is about locating and resolving defects in computer software [4]. As such, the process of debugging can be broken down into two main segments: *locating bugs* (or *fault localization* [21]) and *fixing bugs*.

Existing evidence shows a general pattern of how fixing bugs depends on locating bugs, which requires a thorough understanding of the code and the problematic segment. Katz and Anderson suggested that programming experience determines the ability to comprehend the context of debugging [11], which further determines the efficiency and success of debugging [3], [42]. However, little is known about how language- and paradigm-specific factors could result in variations in debugging strategies. The majority of literature of the debugging activity has been done with imperative languages such as Fortran [3], Java [5], [6] and JavaScript [33], with both novice and expert programmers. With functional languages, while there is some research into automated fault localization techniques for Haskell [43], [44] and OCaml [45], few studies focus on how *humans* locate bugs in functional languages, and the sole study that we found was done with novice Lisp programmers [11]. In 2005, the Haskell maintainers surveyed Haskell programmers for future improvements to the language [19], [34], which unexpectedly surfaced challenges in debugging Haskell programs. Yet, evidence lacks in how debugging in Haskell is done and impeded by these challenges.

As a step towards understanding the effects of programming language- and paradigm-specific factors on debugging experience, we study how expert Haskell programmers debug their code and whether features of the Haskell language and functional programming in general lead to convenience or challenges in debugging. Our preliminary results find similarities between strategies for debugging Haskell programs and strategies for debugging programs written in other languages, as existing literature suggests [3], [5], [11], [32]. Nevertheless, our study also indicates that some features in Haskell and functional programming (some of which agree with findings from the 2005 GHC survey [34]) might impede the execution of the identified debugging strategies, such as forming a mental of execution and adding code to print, which requires further investigation.

In addition, while debugging aids exist for functional languages [19], [28], [46]–[53], whether or not debugging aids are widely adopted in functional programming remains unknown. Indeed, while WinHIPE [51], a novice-oriented IDE for a functional language called Hope, has been evaluated with novice programmers [12], to the best of our knowledge there is no usability study of any debugging aids for functional languages with expert programmers. Although we study the broader context of how expert Haskell programmers debug, we are also able to collect evidence regarding the usage pattern of debugging aids for Haskell (mainly the GHCi debugger [19] and Freja [52], Hood [53], and Hat [28] that precede it), which could be one step towards evaluating and improving the usability of debugging aids for Haskell and for functional languages.

6 Conclusion and Future Work

We interviewed four expert Haskell programmers to understand how debugging is done in Haskell and different from debugging in other languages. We identified several hypotheses to be validated in future research, mainly: while programmers adopt similar debugging strategies in Haskell to the strategies used in other languages, there are Haskell- and functional programming-specific features that might impede executing these strategies. Although some of our findings agree with the 2005 GHC survey [19], [34], our study further reveals programmers' frustration with existing debugging support and Haskell's declarative syntax and abstractions, which may apply to other functional languages.

Future work could involve an extended interview study that includes more participants with diverse backgrounds and adopts similar methodology and questions to gather more evidence of debugging in Haskell. In addition, since Katz and Anderson [11], Romero et al. [5], and Yoon and Garcia [31] all suggested that programmers might consider different debugging strategies when debugging unfamiliar code, participants could be interviewed to compare debugging strategies for Haskell code with different authorship.

Future research could also lead to an observational study in which Haskell programmers debug code and potentially used existing debugging tools, which could complement the anecdotal findings from an interview study. An observational study as such could solicit additional insight into debugging in Haskell, and further identify design opportunities for Haskell debugging aids.

Acknowledgement

This work was supported in part by the National Science Foundation under Grant No. 2107397.

References

[1] S. L. Peyton Jones and P. Wadler, "Imperative functional programming," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93, New York, NY, USA: Association for Computing Machinery, Mar. 1993, pp. 71–84, ISBN: 978-0-89791-560-1. DOI: 10.1145/158511.158524.

[2] J. Hughes, "Why Functional Programming Matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, Feb. 1989, ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/32.2.98.

[3] J. D. Gould, "Some psychological evidence on how people debug computer programs," *International Journal of Man-Machine Studies*, vol. 7, no. 2, pp. 151–182, Mar. 1975, ISSN: 0020-7373. DOI: 10.1016/S0020-7373(75)80005-8.

[4] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: A review of the literature from an educational perspective," *Computer Science Education*, vol. 18, no. 2, pp. 67–92, Jun. 2008, ISSN: 0899-3408, 1744-5175. DOI: 10.1080/08993400802114581.

[5] P. Romero, B. du Boulay, R. Cox, R. Lutz, and S. Bryant, "Debugging strategies and tactics in a multi-representation software environment," *International Journal of Human-Computer Studies*, vol. 65, no. 12, pp. 992–1009, Dec. 2007, ISSN: 1071-5819. DOI: 10.1016/j.ijhcs.2007.07.005.

[6] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, Gothenburg Sweden: ACM, May 2018, pp. 572–583, ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180175.

[7] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive record/replay for web application debugging," in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, St. Andrews Scotland, United Kingdom: ACM, Oct. 2013, pp. 473–484, ISBN: 978-1-4503-2268-3. DOI: 10.1145/2501988.2502050.

[8] A. J. Ko and B. A. Myers, "Finding causes of program output with the Java Whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Boston MA USA: ACM, Apr. 2009, pp. 1569–1578, ISBN: 978-1-60558-246-7. DOI: 10.1145/1518701.1518942.

[9] E. Schoop, F. Huang, and B. Hartmann, "UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, Yokohama Japan: ACM, May 2021, pp. 1–16, ISBN: 978-1-4503-8096-6. DOI: 10.1145/3411764.3445538.

[10] B. T. Vander Zanden, D. Baker, and J. Jin, "An explanation-based, visual debugger for one-way constraints," in *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology - UIST '04*, Santa Fe, NM, USA: ACM Press, 2004, p. 207, ISBN: 978-1-58113-957-0. DOI: 10.1145/1029632.1029670.

[11] I. Katz and J. Anderson, "Debugging: An Analysis of Bug-Location Strategies," *Human-Computer Interaction*, vol. 3, no. 4, pp. 351–399, Dec. 1987, ISSN: 0737-0024. DOI: 10.1207/s15327051hci0304_2.

[12] M. Á. M. Sánchez, C. A. L. Carrascosa, C. P. Flores, J. U. Fuentes, and J. Á. V. Iturbide, "Empirical Evaluation of Usability of Animations in a Functional Programming Environment," Universidad Complutense de Madrid, Technical Report 141/04, 2004, p. 22.

[13] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, Sep. 1989, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/72551.72554.

[14] A. Goel and J. Vitek, "On the design, implementation, and use of laziness in R," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, Oct. 2019, ISSN: 2475-1421. DOI: 10.1145/3360579.

[15] C. V. Hall and J. T. O'Donnell, "Debugging in a side effect free programming environment," in *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, ser. SLIPE '85, New York, NY, USA: Association for Computing Machinery, Jun. 1985, pp. 60–68, ISBN: 978-0-89791-165-8. DOI: 10.1145/800225.806827.

[16] J. T. O'Donnell and C. V. Hall, "Debugging in applicative languages," *LISP and Symbolic Computation*, vol. 1, no. 2, pp. 113–145, Sep. 1988, ISSN: 1573-0557. DOI: 10.1007/BF01806168.

[17] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of Haskell: Being lazy with class," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, San Diego California: ACM, Jun. 2007, ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856.

[18] *Stack Overflow Developer Survey 2022*. [Online]. Available: https://survey.stackoverflow.co/2022/?utm%5C_source=social-share%5C&utm%5C_medium=social%5C&utm%5C_campaign=dev-survey-2022.

[19] S. Marlow, J. Iborra, B. Pope, and A. Gill, "A lightweight interactive debugger for haskell," in *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop - Haskell '07*, Freiburg, Germany: ACM Press, 2007, p. 13, ISBN: 978-1-59593-674-5. DOI: 10.1145/1291201.1291204.

[20] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, Jan. 2006, ISSN: 1478-0887. DOI: 10.1191/1478088706qp063oa.

[21] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016, ISSN: 1939-3520. DOI: 10.1109/TSE.2016.2521368.

[22] *3. Using GHCi Glasgow Haskell Compiler 9.4.3 User's Guide*. [Online]. Available: https://downloads.haskell.org/ghc/latest/docs/users%5C_guide/ghci.html%5C#id2.

[23] *Print | Prelude*. [Online]. Available: https://hackage.haskell.org/package/base-4.17.0.0/docs/Prelude.html%5C#v:print.

[24] *Debug.Trace*, https://hackage.haskell.org/package/base-4.17.0.0/docs/Debug-Trace.html.

[25] N. Mitchell, *Ghcid*, Nov. 2022. [Online]. Available: https://github.com/ndmitchell/ghcid.

[26] 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*, Sep. 2020. DOI: 10.1145/3462172.

[27] P. Z. Ingerman, "Thunks: A way of compiling procedure statements with some comments on procedure declarations," *Communications of the ACM*, vol. 4, no. 1, pp. 55–58, 1961.

[28] O. Chitil, C. Runciman, and M. Wallace, "Transforming Haskell for Tracing," in *Implementation of Functional Languages*, G. Goos, J. Hartmanis, J. van Leeuwen, R. Peña, and T. Arts, Eds., vol. 2670, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 165–181, ISBN: 978-3-540-40190-2 978-3-540-44854-9. DOI: 10.1007/3-540-44854-3_11.

[29] L. Prechelt and W. Tichy, "A controlled experiment to assess the benefits of procedure argument type checking," *IEEE Transactions on Software Engineering*, vol. 24, no. 4, pp. 302–312, Apr. 1998, ISSN: 1939-3520. DOI: 10.1109/32.677186.

[30] A. Von Mayrhauser and A. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, Aug. 1995, ISSN: 1558-0814. DOI: 10.1109/2.402076.

[31] B.-D. Yoon and O. Garcia, "Cognitive activities and support in debugging," in *Proceedings Fourth Annual Symposium on Human Interaction with Complex Systems*, Mar. 1998, pp. 160–169. DOI: 10.1109/HUICS.1998.659974.

[32] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct. 2013, pp. 383–392. DOI: 10.1109/ESEM.2013.43.

[33] A. Alaboudi and T. D. LaToza, "Edit - Run Behavior in Programming and Debugging," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct. 2021, pp. 1–10. DOI: 10.1109/VL/HCC51201.2021.9576170.

[34] *The 2005 GHC survey*, Apr. 2009. [Online]. Available: https://web.archive.org/web/20090419132601/http://www.haskell.org/ghc/survey2005-summary.html.

[35] *Common Lisp*. [Online]. Available: https://lisp-lang.org/.

[36] *OCaml*. [Online]. Available: https://ocaml.org.

[37] *Haskell Language*, https://www.haskell.org/.

[38] Z. Hu, J. Hughes, and M. Wang, "How functional programming mattered," *National Science Review*, vol. 2, no. 3, pp. 349–370, Sep. 2015, ISSN: 2053-714X, 2095-5138. DOI: 10.1093/nsr/nwv042.

[39] *Report on the Programming Language Haskell, A Non-strict, Purely Functional Language*, 1990. [Online]. Available: https://www.haskell.org/definition/haskell-report-1.0.ps.gz.

[40] P. Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90, New York, NY, USA: Association for Computing Machinery, May 1990, pp. 61–78, ISBN: 978-0-89791-368-3. DOI: 10.1145/91556.91592.

[41] J. Lubin and S. E. Chasins, "How statically-typed functional programmers write code," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, Oct. 2021, ISSN: 2475-1421. DOI: 10.1145/3485532.

[42] L. Gugerty and G. Olson, "Debugging by skilled and novice programmers," *SIGCHI Bull.*, vol. 17, no. 4, pp. 171–174, Apr. 1986, ISSN: 0736-6906. DOI: 10.1145/22339.22367. [Online]. Available: https://doi.org/10.1145/22339.22367.

[43] F. Li, M. Wang, and D. Hao, "Bridging the Gap between Different Programming Paradigms in Coverage-based Fault Localization," in *13th Asia-Pacific Symposium on Internetware*, Hohhot China: ACM, Jun. 2022, pp. 75–84, ISBN: 978-1-4503-9780-3. DOI: 10.1145/3545258.3545272.

[44] V. Vasconcelos and M. A. S. Bigonha, "HaskellFL: A Tool for Detecting Logical Errors in Haskell," *International Journal of Computer and Systems Engineering*, vol. 15, no. 8, pp. 479–493, Aug. 2021.

[45] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala, "Learning to blame: Localizing novice type errors with data-driven diagnosis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–27, Oct. 2017, ISSN: 2475-1421. DOI: 10.1145/3138818.

[46] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen, "DrScheme: A programming environment for Scheme," *Journal of Functional Programming*, vol. 12, no. 02, Mar. 2002, ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796801004208.

[47] J. Whitington and T. Ridge, "Visualizing the Evaluation of Functional Programs for Debugging," 9 pages, 2017. DOI: 10.4230/OASICS.SLATE.2017.7.

[48] ——, "Direct Interpretation of Functional Programs for Debugging," *Electronic Proceedings in Theoretical Computer Science*, vol. 294, pp. 41–73, May 2019, ISSN: 2075-2180. DOI: 10.4204/EPTCS.294.3.

[49] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy, "Functional programs that explain their work," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '12, New York, NY, USA: Association for Computing Machinery, Sep. 2012, pp. 365–376, ISBN: 978-1-4503-1054-3. DOI: 10.1145/2364527.2364579.

[50] D. Ungar, H. Lieberman, and C. Fry, "Debugging and the experience of immediacy," *Communications of the ACM*, vol. 40, no. 4, pp. 38–43, Apr. 1997, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/248448.248457.

[51] C. Pareja-Flores, J. Urquiza-Fuentes, and J. Á. Velázquez-Iturbide, "WinHIPE: An IDE for functional programming based on rewriting and visualization," *ACM SIGPLAN Notices*, vol. 42, no. 3, pp. 14–23, Mar. 2007, ISSN: 0362-1340, 1558-1160. DOI: 10.1145/1273039.1273042.

[52] H. Nilsson, "How to look busy while being as lazy as ever: The Implementation of a lazy functional debugger," *Journal of Functional Programming*, vol. 11, no. 6, pp. 629–671, Nov. 2001, ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S095679680100418X.

[53] A. Gill, "Debugging Haskell by Observing Intermediate Data Structures," in *Proceedings of the 4th Haskell Workshop*, 2000, p. 12.