Usability Barriers for Liquid Types

CATARINA GAMBOA, Carnegie Mellon University, USA and LASIGE, University of Lisbon, Portugal
ABIGAIL REESE, Carnegie Mellon University, USA
ALCIDES FONSECA, LASIGE, University of Lisbon, Portugal
JONATHAN ALDRICH, Carnegie Mellon University, USA

Liquid types can express richer verification properties than simple type systems. However, despite their advantages, liquid types have yet to achieve widespread adoption. To understand why, we conducted a study analyzing developers' challenges with liquid types, focusing on LiquidHaskell. Our findings reveal nine key barriers that span three categories, including developer experience, scalability challenges with complex and large codebases, and understanding the verification process. Together, these obstacles provide a comprehensive view of the usability challenges to the broader adoption of liquid types and offer insights that can inform the current and future design and implementation of liquid type systems.

CCS Concepts: • **Theory of computation → Logic and verification**; • **Human-centered computing →** *HCI design and evaluation methods.*

Additional Key Words and Phrases: liquid types, usability, liquidhaskell, automated verification, human factors

ACM Reference Format:
Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Usability Barriers for Liquid Types. *Proc. ACM Program. Lang.* 9, PLDI, Article 224 (June 2025), 26 pages. https://doi.org/10.1145/3729327

1 Introduction

Strong type systems have helped developers uncover bugs early in the development process for decades, making them one of the most commonly used verification techniques in the world. Since type checking happens at compile time, developers can find bugs early in the development lifecycle where they are easier and less expensive to fix [59].

Liquid types [53] are more expressive than traditional type systems by adding logical predicates. As an example, a function that takes an input of type $\{v : \text{Int} \mid v > 0\}$ only accepts positive inputs. These logical predicates allow the filtering of valid inputs and the establishment of relationships between values, and they can be more complex (supporting polymorphism and dependent types) but are limited to decidable logics, which an SMT solver can validate.

Since their introduction in 2008 [53], there have been many attempts to integrate liquid types in different programming languages, ranging from functional languages like Haskell [56] to more popular languages like Java [29], C [52] and JavaScript [14], and Rust [40]. They detect a wide range of bugs, from simple divisions by zero or out-of-bounds array accesses to more complex security issues [5] and protocol violations [29].

Liquid types have also demonstrated a practical value across various applications. For example, STORM [41] employed LiquidHaskell to describe the data produced and consumed by different

Authors' Contact Information: Catarina Gamboa, Carnegie Mellon University, Pittsburgh, USA and LASIGE, University of Lisbon, Lisbon, Portugal, cvgamboa@fc.ul.pt; Abigail Reese, Carnegie Mellon University, Pittsburgh, USA, aereese@andrew.cmu.edu; Alcides Fonseca, LASIGE, University of Lisbon, Lisbon, Portugal, amfonseca@fc.ul.pt; Jonathan Aldrich, Carnegie Mellon University, Pittsburgh, USA, jonathan.aldrich@cs.cmu.edu.

ACM 2475-1421/2025/6-ART224
https://doi.org/10.1145/3729327

```
{-@ type Nat = {v: Int | v >= 0} @-}
{-@ abs :: Int -> Nat @-}
abs :: Int -> Int
abs n = if n > 0 then n else (-n)
```

Listing 1. Type alias and liquid type annotations in LiquidHaskell.

layers of an MVC application, LiquidJava [29] utilized liquid types to model typestate protocols in common Java libraries, and Flux, an implementation of liquid types in Rust, was used to track the semantics of database queries [42]. For more examples, Vazou et al. [56] examine other interesting properties in widely-used libraries and critical applications in Haskell that can be verified using LiquidHaskell. Despite the number of different implementations and their flexibility in finding a wide range of bugs, liquid types have yet to become mainstream.

This work aims to understand what prevents liquid types from being adopted by the general developer community despite their higher expressive power. To this end, we conducted an exploratory study to identify the usability barriers to adopting and using liquid types, aiming to outline the challenges that liquid type designs and implementations must overcome.

Given the research community's interest in extending languages with liquid types and the community's commitment to fostering their adoption through improving their implementation,[1] it is essential to understand the current issues to drive improvements. To this end, we aim to answer one main research question:

RQ: What are the current barriers developers face in adopting and using liquid types?

To answer this question, we interviewed and observed 19 developers from academia and industry, of which 12 were new users of liquid types, and 7 were experienced users who have used liquid types in the past or are currently using them. We used different qualitative research methods depending on their expertise, including interviews, observations, retrospectives, and think-aloud protocols, according to the best-fitting techniques for each case. The sample size, though small, is common in this type of qualitative research, where the emphasis is on using methods that produce in-depth insights and nuanced data, contributing to a deeper understanding of the research problem. For this study, we focused on the most mature implementation of liquid types, LiquidHaskell.[2]

We performed our study with 19 participants and identified nine barriers to adopting liquid types. These barriers span three themes, including developer experience, scalability challenges with complex and large codebases, and understanding the verification process.

In this paper we will present some background on liquid types and LiquidHaskell (Section 2), describe the study design (Section 3), present the results (Section 4), discuss the results in comparison to other implementations of liquid types and verification techniques (Section 5), present related work (Section 6), and draw conclusions (Section 7).

2 Background: Overview of LiquidHaskell

LiquidHaskell enriches the Haskell type system with refinement types. Refinement types are defined by logical predicates that describe the set of valid values [36]. By restricting these types, it is possible to reject programs with values outside of the allowed ranges.

[1] https://discourse.haskell.org/t/call-for-sponsors-first-class-liquid-haskell/6973
[2] https://ucsd-progsys.github.io/liquidhaskell/

```haskell
{-@ data SList a = SL {
  size :: Nat, elems :: {v:[a] | realSize v = size} } @-}
data SList a = SL { size :: Int, elems :: [a] }
```

Listing 2. Datatype invariants in LiquidHaskell.

Listing 1 shows the **abs** function refined in LiquidHaskell with the type alias `Nat` which is a synonym to the type integer, refined by the predicate `v > 0`. The idea of a type alias is to give a name to a possibly complex refined type that may appear throughout the program. The, during type checking, the implementation is checked against its liquid type by generating verification conditions for each branch of the if expression. For the else branch, the verification condition would be $\forall n, \neg(n > 0) \implies -n >= 0$ as in that branch, we know the branching condition is false, and the expression needs to have the same type as the return type of the function. This verification condition is discharged to an SMT solver (such as Z3 [22]). If it is able to find a counter-example for n, type-checking fails.

It is also possible to add refinements to datatypes to define invariants and properties of data values. For example, Listing 2 adds a refinement to the new datatype `SList`, where `size` is now a natural number (instead of a regular int), and the list of elements has exactly `size` elements.

Only measures (logical-level functions) can appear in refinements. However, total, terminating recursive functions, with non-overlapping constructor patterns, can be reflected to the type-level as well, such as `realSize`, in this example.

For a comprehensive understanding of liquid types, the book Refinement Types: A Tutorial [36] provides an excellent resource.

3 Interviews with Developers

To answer our research question, we designed an empirical study where we interviewed and observed developers using LiquidHaskell, the most mature implementation of liquid types. Alternative methods, such as analyzing public source code, fail to answer our research question due to survival bias. The code that survived the process of publishing a commit does not evidence the doubts, decisions, and issues that existed in the intermediate steps that led to that commit.

Due to the limited size of the LiquidHaskell community and to understand the issues that occur when first learning the language and the ones that persist after one gains proficiency with the tool, we considered developers with different levels of expertise in LiquidHaskell:

- **New Users**: those who are familiar with Haskell but not with LiquidHaskell;
- **Experienced Users**: those who have used LiquidHaskell in their projects, even if they are not currently using LiquidHaskell at the moment.

For each of these populations, we designed different study sessions, as described below.

This study was approved by IRBs in two institutions from different continents, and the artifact to reproduce the study is publicly available [30].

3.1 Study Design

Figure 1 gives an overview of the study design, with different highlights for two subgroups of participants with distinct protocols.

The entry point for our study was through a sign-up form, which we publicized on social media, mailing lists, relevant Slack channels, and through emails to researchers in the field. The sign-up form collected both background information and participants' availability for the synchronous session either in-person or online. The background information was used to categorize participants

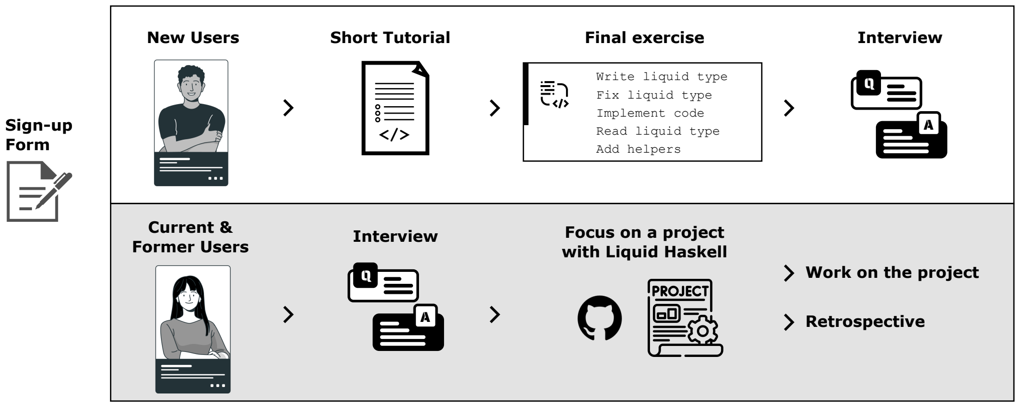


Fig. 1. Study design with new and experienced users of LiquidHaskell.

by expertise to decide which study protocol to follow, either one for **New Users** or **Experienced Users**. The design of each protocol is detailed below.

New Users- Tutorial and Observation. Because many developers have never heard of liquid types, we designed a tutorial to introduce LiquidHaskell to these users during our study session. Because LiquidHaskell requires Haskell expertise, we also asked for their self-assessed experience level with Haskell and included two basic Haskell exercises to validate it.

To understand the challenges of novices first learning LiquidHaskell, we asked those familiar with Haskell to follow a short tutorial, complete an exercise, and answer some questions about their experience in a session that lasted up to 2 hours. Throughout the tutorial, we observed participants completing the exercises and answered any questions they had.

Short Tutorial. The tutorial follows a "computational notebook" style, similar to Jupyter notebooks,[3] where we introduce the concepts along with exercises and quiz questions. To minimize the impact of material quality, we used the official LiquidHaskell tutorial,[4] which is where prospective users typically start. However, we have condensed it to fit within the allotted time for the interview, keeping all core concepts covered during the first 30 minutes of the session. To reduce interviewer bias, we added engagement questions with user-available correct answers — a recommended approach for enhancing training [58].

To streamline the tutorial, we selected a specific exercise for participant observation, then included only the sections covering concepts essential for its completion. For this exercise, we chose the implementation of **Okasaki's queues** [49], as it is the first case study that requires a comprehensive coverage of LiquidHaskell features. Although implementing this complex data structure might not reflect participants' routine programming tasks, the exercise demonstrates core LiquidHaskell features and verification principles while remaining concise. Moreover, most LiquidHaskell users would have gone through a similar exercise in their learning process. The original tutorial already contains coding exercises, so we included these exercises and created others to organize the information in a flow that makes sense for a short introduction. Additionally, we introduced multiple-choice questions and made the *Answer* available, so developers could know if they were on track. We performed two pilot studies with the tutorial and improved it between iterations to ensure it was feasible to complete during the desired time frame.

[3]https://jupyter.org/
[4]https://ucsd-progsys.github.io/liquidhaskell-tutorial/

Exercise - Observation and Interview. During the final exercise, we asked participants to use the Think-Aloud methodology [25] to verbally capture the issues they had, and the rationale for each of their steps. Before starting the exercise, we explained this methodology and asked the participants to use it in the last exercise of the tutorial, as recommended in literature [1].

During the Okasaki exercise, participants followed the tutorial instructions, and the interviewer aimed to interfere only in two specific cases. First, when participants asked for help, the interviewer tried to understand the question and give a hint for how to correct the error. Second, if the interviewer saw that a participant was stuck and stopped verbalizing their thoughts, the interviewer waited up to 30 seconds before asking what was going through the participant's mind.

To understand the impact of different aspects of liquid types, we split the final exercise into smaller tasks. These tasks included writing and fixing liquid types, implementing a function given a specification, reading and reasoning about specifications, and defining measures and type aliases.

At the end of the tutorial and exercise, we conducted a semi-structured interview about their experience and the challenges they faced while using liquid types.

Experienced Users- Interview and Project Showcase. With more experienced liquid types users, we aimed to understand their challenges learning liquid types, challenges in adopting them in larger projects, and why former users no longer use them.

For these users, we started by interviewing them about their experience with liquid types and then asked them to showcase a project where they used liquid types. This project serves as a stimulus for participants to recall the challenges they faced, complementing what they mentioned in the interview, and also allows them to focus on specific instances where a given challenge appeared. Therefore, we started by conducting a semi-structured interview, and then we asked them to share a non-confidential project, explain its purpose and walk us through the codebase.

If the project was already completed, we asked the participant to do a retrospective on the project, following the methodology of retrospective protocol analysis [58], which allows the participant to talk about their past experience and describe past events by using a resource to effectively prompt their memory, in this case the codebase. We asked the participants to show us where and how they used specifications, what parts of the project were more challenging, and what was the overall development process using liquid types.

For ongoing projects, we had the opportunity to not only ask the participants about their experiences but actually observe them while they work on a project. This observation brings more in-depth data, since it allows us to see the participants' workflow, and we can see the issues that arise at the exact moment they are working on them, while in completed projects we can only rely on the participants' memory. This practice follows the idea of contextual inquiry [35, 51] where the participant and the interviewer adopt a master-apprentice relationship, and it is possible to capture the users' experience and thoughts while they work on their projects. Therefore, after the introduction about their project, we asked developers to work on it for around 30 minutes while we observed and gathered data about their development process. During this time, we let the developer work as they explained their task and how they were solving it, and asked them questions to better understand the thought process involved in completing the task at hand.

At the end of the project showcase, we asked more questions to clarify previous answers.

3.2 Recruitment

One of the main challenges of conducting user studies in software engineering is recruiting participants [10]. In addition, we needed participants for two different levels with very specific knowledge requirements. Therefore, we planned our recruitment in three main phases:

(1) Share the study on social media channels;

ID	Work Position	Time w/ Haskell	Purposes	Knowledge of Verification/ Proof Systems
NC6	Dev. in Industry	> 3 years	Personal	Hands-on experience
NC16	Dev. in Industry	> 3 years	Personal	Hands-on experience
NC2	Dev. in Industry	> 3 years	Personal	Small hands-on experience
NC4	Dev. in Industry	> 3 years	Personal	Aware, no experience
NC3	Dev. in Industry	> 3 years	Work+Personal	Aware, no experience
NC1	Dev. in Industry	1-3 years	Work+Personal	Small hands-on experience
NC7	Dev. in Industry	1-3 years	Work	Aware, no experience
NC15	Undergrad Student	1-3 years	Education	Not aware
NC8	Undergrad Student	< 1 year	Education	Not aware
NC10	Undergrad Student	< 1 year	Education	Not aware
NC13	Undergrad Student	< 1 year	Education	Not aware
NC17	Faculty Member	> 3 years	Education	PhD in theorem proving

Table 1. Background of **New Users** including their work positions, how long they have known Haskell for, and for which purposes they use Haskell at the moment. Additionally, we asked for their knowledge of verification and proof systems, ranging from no experience to hands-on experience. Participants are sorted primarily by work position, then by time with Haskell (descending), followed by purpose of use, and finally by level of verification experience. The ID starting with NC represents "New Users".

ID	Current Position	Context of using LiquidHaskell
FM1	Faculty Member	In academia as Graduate Student
FM2	Developer in Industry	In academia during an internship
FM3	Developer in Industry	In academia as Graduate Student
FM4	Developer in Industry	In academia as Graduate Student
FM6	Developer in Industry	In academia as PL researcher
FM7	Developer in Industry	Personal and industry projects
CR1	Developer in Industry and Graduate Student	In academia as Graduate Student

Table 2. Background of **Experienced Users** participants, including their current professional positions and the contexts in which they have used LiquidHaskell. Participants are arranged by their primary association. The ID starting with "FM" represents a "Former User" and CR represents a "Current User".

(2) Share the study on the LiquidHaskell slack and send it to relevant mailing lists;
(3) Contact authors of research papers, and active contributors to the Github repository.

From the first step, we got most of our **New Users**, but very few applications from the other categories, as we expected. Therefore, to reach **Experienced Users**, we shared the study in popular mailing lists for type system research and contacted authors of research papers that use and mention LiquidHaskell, inviting them to participate and share the study.

In total, we recruited **12 New Users** for the study, and their background is shown in Table 1. These participants differ in their work position, time using Haskell and their knowledge of verification/proof systems. In addition, we recruited **7 Experienced Users** for the study, and their background is shown in Table 2. The sample size, though small, is common in this type of qualitative research, where the emphasis is on using methods that produce in-depth insights and nuanced data, contributing to a deeper understanding of the research problem.

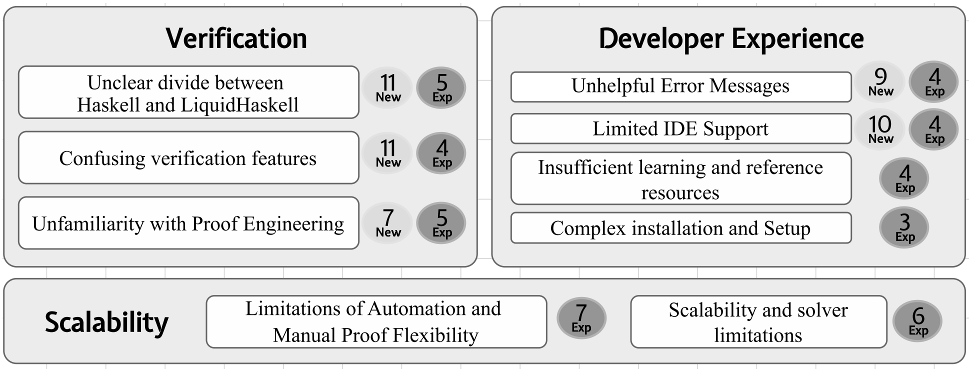

Verification
Unclear divide between Haskell and LiquidHaskell
11 New
5 Exp
Confusing verification features
11 New
4 Exp
Unfamiliarity with Proof Engineering
7 New
5 Exp
Developer Experience
Unhelpful Error Messages
9 New
4 Exp
Limited IDE Support
10 New
4 Exp
Insufficient learning and reference resources
4 Exp
Complex installation and Setup
3 Exp
Scalability
Limitations of Automation and Manual Proof Flexibility
7 Exp
Scalability and solver limitations
6 Exp

Fig. 2. Barriers encountered by **New Users** (New) and **Experienced Users** (Exp), with frequency counts showing how many participants in each group mentioned or experienced each barrier.

3.3 Qualitative Data Analysis

We recorded all the sessions with the participants, including their screens and their faces when permitted, and transcribed the interactions. Then, we performed a qualitative analysis of the data using Qualitative Coding and Thematic Analysis [54] with inductive coding, where the codes emerge during the analysis. For **New Users**, we considered the final exercise as the main source of data and, for **Experienced Users**, we considered the interviews and the project showcase.

For **New Users**, the video recording contains them solving the exercises, therefore we annotated the video with every time they struggled to get the correct answer for a given task, when they had questions about the exercises, and when they showed expressions of confusion with the task at hand. In each of these cases, we created codes related to what sparked the confusion or questions, and we added those codes to the relevant part of the video. The comments that participants made during this time were similarly were also analyzed, and we added in vivo codes for relevant quotes. For the interview at the end, we first separated the parts of the interview that were related to each question, and then created and applied codes to the answers.

For **Experienced Users**, we first looked at the interview answers to create relevant codes, and then we applied them to the video during the project showcase. During the project showcase participants gave examples for what they had mentioned before, and also recalled other challenges and issues they faced by looking at specific parts of the code, which we also analyzed.

At the end, we grouped the codes into themes that represent the main challenges participants faced, and we collaboratively discussed them to ensure that we were interpreting the data correctly.

4 Results

From our analysis of the study with **New Users** and **Experienced Users**, we identified nine distinct themes representing the principal usability barriers participants encountered when working with LiquidHaskell. Figure 2 illustrates these barriers alongside the number of participants who mentioned or experienced each challenge. These numbers, however, should not be interpreted as measuring the significance or prevalence of each problem, as our methodology was not designed for this quantitative assessment.

As depicted in the figure, the barriers that **New Users** experienced during their session were also seen by experienced users, but with different levels of complexity. The additional challenges reported by **Experienced Users**, come from the use of LiquidHaskell outside a controlled environment, in larger projects and on more complex use cases.

This section presents the detailed results from our study. For **New Users**, the findings derive from their engagement with the tutorial, while for **Experienced Users** the findings are tied to their experience in the projects they shared. Therefore, this section begins by contextualizing the experienced users' projects and motivation to use LiquidHaskell (Section 4.1), and then details each identified barrier with examples from the study (Section 4.2 through Section 4.10). Each identified barrier includes tags indicating which participant groups reported it.

4.1 Experienced Users' Project Contexts and Motivations

Our seven **Experienced Users** showcased projects across three different application areas: Data Structure Analysis (FM3, CR1), such as implementing various tree structures while maintaining the same invariants; Information Flow and Property Verification (FM1, FM4, FM6), where liquid types were used to protect private information; and Applying PL Theory with Liquid Types (FM2, FM7), for instance writing an interpreter for lambda calculus. The participants developed these projects primarily as part of their academic work (6/7), with one participant working on personal and industry projects (1/7), resulting in a predominantly academic and research-oriented scope.

When asked why they were using LiquidHaskell for these projects, several participants (CR1, FM1, FM6, FM7) highlighted the automation provided by the SMT solver and type inference as a key factor. This automation allowed them to write less repetitive code and avoid proving everything manually when the solver could infer it. Additionally, participants (FM1, FM2, FM4, CR1) cited the usability of liquid types as an important factor in their decision to adopt this technology:

> ❝ *You can do the same as dependent types but with less code [...] and you are constantly being support by the SMT solver. (CR1)*
>
> *Liquid types are really nice and maybe nicer than dependent types for a more broad user base, right? I think the majority of people that already program could, in a way, program with refined types. (FM2)*

Other aspects that participants mentioned as reasons for using liquid types include the ability to add specifications directly to the implementation, thereby verifying the code that is executed (FM3, FM4, FM6); the maturity of the LiquidHaskell implementation compared to other liquid types tools (FM1, FM2); and the improved quality and reliability of verified code (FM2, CR1).

> ❝ *I really wanted to have my code and the additional proof, and that was a thing that fascinated me. (FM3)*
>
> *You are verifying properties in the language that you're developing in. So I think it's more approachable for developers. And you are actually verifying the code that you want to run. (FM4)*

Our **Experienced Users** do not use liquid types at the moment because of several factors: some indicated their current projects do not require the additional proof properties (FM1, FM3, FM7); others are not currently using Haskell (FM4, FM6); and some are using languages that do not support liquid types (CR1, FM2).

> ❝ *If there were liquid types for Dart - I would be using them. For Kotlin as well. (CR1)*

The additional factors contributing to participants' not using liquid types are detailed in the sections that follow.

4.2 Unclear divide between Haskell and LiquidHaskell New Experienced

LiquidHaskell was designed to be a superset of Haskell, where liquid type annotations further refine the types in programs. As liquid type-checking is enabled globally and due to type inference, the predicates in type annotations are propagated to all parts of programs. Several challenges arise from this weaved interaction. One participant commented:

> ❝ *So it's sort of like you're doing two things at once because you're implementing in Haskell. But you're also talking to GHC, but you're also talking to LiquidHaskell. Which I'd say is a*

```
1  {-@ rot :: f:SList a -> b:SListN a {size f + 1} -> acc:SList a ->
2              SListN a {size f + size b + size acc} @-}
3  rot f b acc
4    | size f == 0 = hd b `cons` acc
5    | otherwise = hd f `cons` rot (tl f) (tl b) (hd b `cons` acc)
```

Listing 3. Correct solution of the last exercise, where participants were asked to correct a specification of the `rotate` function.

layer of challenge, but it's good payback, and honestly, I got into Haskell because I liked that: It's challenging! (NC2)

4.2.1 Changing Haskell code to fit into verification. In total, 12 of our participants mentioned they had to change the way they write programs in Haskell to fit the verification process, as they had to combine both Haskell and the verification logic at the same time. Besides needing a solid knowledge of the target language (NC7, NC15), developers mentioned they need to think about both Haskell and the verification logic at the same time (NC2, NC13, NC15).

One of our participants, who already had a short hands-on experience with proof systems, commented during the interview:

I found myself writing code in certain ways, or I found the code being written in certain ways to convince the proof assistant of things that were not necessarily the most intuitive. (NC2)

Other participants (NC6, NC16, NC17) also reported that certain exercise implementations in the tutorial appeared non-idiomatic to them, as they would not implement those functions in Haskell in the same way. Listing 3 is one example of such exercises, where participants reported they would use pattern matching to handle different cases rather than calling functions.

This example also illustrates that an idiomatic implementation in liquid types may be different from the idiomatic implementation in Haskell. For instance, in Haskell, developers are used to include pre-condition checks directly in their code, but in LiquidHaskell these checks can be moved from the implementation to the type specification, creating a different coding pattern. During the study, the function `rot` followed this new pattern, which confused two participants (NC1, NC3). In this exercise, participants were given the implementation (lines 3-5) and properties described in natural language, with the task of correcting an incorrect liquid type. While completing the exercise, NC3 first examined the implementation before addressing the liquid type and found the base case (line 4) confusing because it only used the *head* of `b` while ignoring the *tail*, suggesting that the implementation failed to handle all cases. This implementation only becomes clear through the liquid type's pre-condition (highlighted in line 1) that `b` must always contain exactly one more element than `f`. Consequently, when `f` is empty in the base case, the LiquidHaskell checker knows that `b` contains a single element at its *head*, so it is unnecessary to add a case for the *tail* as they would typically do in Haskell. This example demonstrates that the specification not only represents the implementation's behavior but also influences the implementation by expressing the function's underlying intent.

Experienced Users also noted that they needed to make changes in their Haskell code to fit the verification paradigm, mentioning a different way of programming with liquid types (FM6, CR1):

I think that liquid types have, like, its own way to program. (CR1)

Participants reported needing to break functions into smaller pieces when they need auxiliary proofs (FM4, CR1). When working on their project, CR1 started adding liquid types to one function and ended up breaking it up into multiple smaller functions to prove each of the

```
#if NotLiquid
import qualified Data.List as List
import Data.Map (Map)
#else
import qualified Liquid.Data.List as List
import qualified Liquid.Data.Map as Map
#endif
```

Listing 4. Shadowing of modules for LiquidHaskell presented by FM4 during their project showcase.

branches separately. Specifically, they started with the function `delete_node`, and then created `delete_node_left` which was called inside one branch of the main function, and then created `delete_node_left_right_great` which was called inside the latter, and other similar functions were needed for the remaining branches. Breaking the function into smaller pieces helps convey the properties for the proofs but also makes the code more complex and harder to read.

These changes in implementation also hint at another challenge when adding liquid types to existing projects, where code refactoring might be needed to fit into the verification process. Reusing off-the-shelf components also becomes harder (FM6, FM4, and FM7), as users use different imports for the modules with liquid type annotations for verification and for executing the code. For example, FM4 showed that they had to shadow the list and map module, adding compiler directives (i.e., pragmas) similar to the ones in Listing 4.

The required changes in the implementation may vary depending on the project and the proofs needed. In the projects of our participants, they had some leeway to write the implementation and change it. Some participants started with already defined examples implemented and added the liquid types after while trying to modify the code as little as possible (FM1, FM7). FM3 also started with a base implementation, but realized a different approach would probably have helped more:

> “ *I had all the code. Then I added one property to my data structure, and then it was all red. And that's very frustrating, because no single line of code works anymore. And then you have to repair, repair, repair [the implementation], and the other way around would be more motivating and probably also easier. (FM3)*

Other participants started writing the types first, then the liquid types, and then the implementation following the expressed properties (FM2, CR1), but they were already familiar with the problem they were solving, so there were no surprising implementations. The remaining participants wrote the liquid type and the implementation side by side (FM4, FM6), sometimes leaving the parts of the proof to fill in after the implementation. Given this symbiotic relationship between the implementation and the liquid types, it may be difficult to reuse existing code or off-the-shelf components, as the code may need to be refactored to fit the verification process.

4.2.2 Naming mismatches in liquid types with type variables and implementation. Another issue that participants faced was the mismatch between the names of the variables in refinement types and variables in the program as experienced by NC8 and NC10. In LiquidHaskell annotations developers can give names to the arguments of functions, which do not need to match the variable name in the pattern matching. This freedom can result in confusing code. For example, the function **`cons`** had the following specification and implementation:

```
{-@ cons :: a -> xs:SList a -> SListN a {size xs + 1} @-}
cons x (SL n xs) = SL (n+1) (x:xs)
```

In this example, the variable `xs` in the liquid type refers to the first argument of the function (highlighted in line 1), while in the implementation, it is pattern matching as a list inside the second argument of type `SL` (highlighted in line 2). While reading this example, NC10, was quiet for three minutes before asking the interviewer for help, since they were not understanding what `xs` was referring to in. Therefore, this naming flexibility can lead to confusion and errors in the liquid type, as our participants experienced.

Additionally, in Haskell's convention, lowercase identifiers in function signatures are used for type variables. However, in LiquidHaskell we can refer to program variables (also in lowercase) inside the liquid type signature, making it unclear whether a lowercase identifier is a type variable or a regular variable in annotations. NC8 encountered this issue when incorrectly writing the type alias `{-@ type NEList a = {v: SList a | size a > 0} @-}` where `a` is a type variable representing the type of its contents, not the list variable itself, so the predicate should be `size v`. A similar case happened to three other participants (NC1, NC2, NC10), and it resulted in a cryptic error message, prompting participants to seek help understanding the problem. The challenges with these error messages are detailed in Section 4.7.

4.3 Confusing verification features

New Experienced

Participants faced challenges with verification features, such as lifting functions as measures, using type aliases with bound and unbound variables, and understanding the SMT solver reasoning.

4.3.1 Lifting functions as measures. During the tutorial, we presented the concept of measures, where a Haskell function is lifted into the specification level and can be used in the predicates. Initially, this concept seemed easy to understand, but in the exercises, participants started questioning what kind of functions could be lifted as measures (NC3, NC4, NC10, NC13, NC15). They tried implementations that were not total and invoked functions that were not reflected in the verification logic leading them to express confusion between what is available in the type-level and term-level. As an example, NC10 wanted to use the attributes of the data type in the measure instead of using a previously defined measure (i.e., `qsize q = (size $ front q)+ (size $ back q)`), which could not be verified, and produced the error message *"Error: Bad measure specification measure X.qsize Unbound symbol q ##aYA – perhaps you meant:head, tail?"*. In this case, while the error message provided a suggestion, it was unclear why the suggestion was relevant, as the participant hadn't recognized that the issue was in the measure definition.

Not only **New Users** struggled with lifting functions, as CR1 also mentioned having a hard time understanding when to lift functions to measures to use them as predicates:

> *You have to know [when to] promote that function and, for me, it was not trivial (...) And it feels a little like luck that you have [tried it out]. (CR1)*

4.3.2 Bound and unbound variables in type aliases. When a predicate is used multiple times, it is common to create a type alias to avoid this repetition. These type aliases can have both type and value parameters so that the same predicate can be used in different contexts. As an example, we asked developers to write an alias for a queue of a given size, which could be described as:

```
{-@ type QueueN a N = {v: Queue a | qsize v = N} @-}
```

In this type alias, `a` is a type parameter, and `N` is a value parameter, and when using the type alias, it is necessary to provide both these arguments. Participants tried to use the value parameter of the alias in relation with other predicates. For example, NC8 tried to use the value parameter as a variable that could connect the size of an input and output list when an element was removed (i.e., `{-@ remove:: QueueN a N -> QueueN a {N-1} @-}`). However, `N` is not defined in the context of the function, and LiquidHaskell does not bind it directly to a *forall* quantifier automatically. Therefore, this code will return an error message since `N` cannot be used for the participants' intended purpose. Participant NC16, who had hands-on-experience with proof systems, said:

> *I need like a forall quantifier for n or something like it (...) I can't think of an example of that. (NC16)*

The unclear relation between bound and unbound variables and how they are written in code was a common challenge for **New Users** (NC6, NC8, NC15, NC16, NC17). This confusion might stem from the intuition that `N` should be bound in the same way as `a`. This expectation arises not only from Haskell's use of forall for all type arguments but also from participants' experience with other languages where this behavior is standard. For example, when discussing this error, NC6 reported:

> *Cause it feels to me like I should be able to say here that this is actually a list where I know the size, and then the resulting sizes the size minus one. Which I'm used to in [other languages]. So, I've done programming in Clash, which has size vectors, so they're inductive vectors [where] you can quite happily say things like, or even the number types. (NC6)*

4.3.3 Verification process and SMT solver reasoning. Overall, **New Users** only scratched the surface of LiquidHaskell and were not introduced to the details of the verification process, which caused some confusion whenever the verification failed. NC1 mentioned felling the need to learn more about LiquidHaskell's internal verification and how the SMT solver works to better use liquid types:

> *Having thought through how constraints are sent to an SMT solver and how that failure would work itself out into an error could have been helpful from the start. (NC1)*

Participants with experience with liquid types, reinforced the need of understanding how the verification is performed and understanding which conditions are being checked and which ones are missing to prove the desired properties (FM2, FM3, FM4, CR1). Participants (CR1, FM4) mentioned that when conditions became harder to proof it was difficult to understand which statements were true and which ones were missing for the proof to go through. FM3 mentioned that they would debug these issues with *assumptions* sent to the SMT solver, although they wished they would have known about that "trick" earlier. CR1 also mentioned a time when they introduced a measure that made all properties in the program pass verification, just to understand that the function was the opposite of another function and, it was adding an impossible predicate to the premises of the verification, making the negation of the predicate always true for the SMT Solver. These challenges show that understanding the verification process and knowing which verification features exist is important, but it also shows that participants need to have some familiarity with proof engineering, as we will discuss in the next section.

4.4 Unfamiliarity with Proof Engineering

New Experienced

Liquid types aim to provide developers a low entrance barrier for code verification through familiar type systems, a perspective that our participants shared:

> *So we kind of assume that [LiquidHaskell] is kind of this middle term between a good type system (...) and dependent types, which is like this really expressive and powerful type system, but that you need to have a doctorate's degree to understand what's going on. (FM2)*

However, for verifying complex code or properties, developers still need to have a solid background or at least some familiarity with proof systems.

4.4.1 Identifying pre- and post-conditions. Before actually writing the liquid types themselves, developers need to figure out what properties they want to prove for a given function. We removed most of this work with for **New Users** since we asked participants to complete most of the liquid types with the invariants we provided. But still, participants (NC6, NC15) recognized this as an active challenge:

> *I think that if I had to come up with the type of rotate by myself, trying to figure out what these invariants should be might be quite difficult. Particularly if you haven't been given the properties that you want. [...] Once you understand those invariants, then actually implementing them is not particularly difficult. (NC6)*

In our pilot sessions, we initially included an exercise for the participants to figure out a complex invariant, however we ended up removing this approach because pilot participants struggled to identify the proper invariants, with this single exercise consuming over 30 minutes of session time.

Our **Experienced Users** also mentioned this challenge (FM1, FM2, FM3, FM4, CR1). They reported they need to think carefully about the properties they want to prove, which edge cases they need to account for, and how to translate them into predicates that can be verified by the SMT Solver. Additionally, when a proof does not go through, developers need to understand what they need to add to the specification for it to go through. FM1 mentioned needing a "nudge" to start reasoning about proof terms, and FM2 said they need to have a large intuition of how to write every step of the proof, using more a manual-proof style, for the verification of complex properties:

> *Let's say, sometimes you have to prove things that you don't really know how to prove or what to use to prove, obviously automation helps in those things but, you can never automate everything. (FM2)*

This switch from using automation to manual proofs is a challenge by itself that we will discuss in Section 4.5.

4.4.2 Strongest predicate required. Another challenge was understanding how to appropriately define the strength and strictness of liquid types when verifying code properties (NC3, NC7, NC10, NC16). For exercises such as `remove`, to retrieve the head element of a queue, several participants (NC8, NC3, NC7) consistently wrote post-conditions that were weaker than optimal, only realizing this when reviewing the exercise solutions. For instance, NC3 specified a post-condition that the queue size should be less than the original size, when a stronger and more precise post-condition would have specified that the size should be exactly one less than the original size.

Experienced Users also mentioned the need to consider how strong the predicates need to be to prove the properties, given that verification can fail if they are too permissive or too restrictive (FM1, CR1). One of our participants, NC1, mentioned that concern even during their process of adding liquid types. They would write the liquid types and check their verification, if it failed, there was certainly a problem to be fixed, but if it passed, it did not necessarily mean that everything was correct because the specification might be too permissive. Therefore, they had to go back to the specification to check if the predicates were strong enough for the given task.

From a business perspective, FM7 mentioned that this process makes it is difficult to estimate the effort needed to verify a given part of the code, and a given customer might want to know this information right from the start. This, however, depends a lot on the complexity of the code and the properties that need to be proved, but having some familiarity with other similar proof systems can help in these estimates.

4.5 Limitations of Automation and Manual Proof Flexibility

Experienced

One of the main advantages of using LiquidHaskell is the automation that it provides, with the proofs being discharged to the SMT Solver to be verified, as mentioned by a participant:

> *You are constantly being supported by the SMT solver, [so] you don't have to be proving everything that you are doing. (CR1)*

However, this automation also brings challenges that most advanced programmers need to overcome, as mentioned by all seven of our **Experienced Users**. It is not straightforward to understand why automation fails, and when to change from automatic to manual proofs - those requiring explicit intermediate steps similar to interactive theorem prover techniques. Because liquid types are limited to decidable logics, in cases where non-decidable elements are needed in properties, it is necessary to use proof terms and use a manual-style proof. Therefore, these proofs require a mix of automatic and manual-style proofs, and the first challenge is for developers to understand when to mix these styles and get to an automation point.

> “ *So as long as you are relying on the automated proof checker to do things you can still work your way on it. But the moment we need actually manually written proof terms, I could not quite get it (...) how would you write proof terms? And when do you write proof terms? That's difficult because there's no hint given by the type checker to say, you know, you probably [need to add a] proof term manually. (FM1)*
>
> *You have to get to a point where you can automate the proof, and getting to that point is [the challenge]. (CR1)*

The other challenge, when the solver and the automation stop helping, is that there are not many elements that can help a developer move forward in a proof when compared with other interactive proof assistants that feature tactics and search automations like hammers [7]. Therefore, when proofs change to a manual mode, the flexibility of liquid types is hindered, and some participants such as FM6 and FM7 highly prefer to change to other proof assistants.

> “ *(...) at the moment there's no mechanism to make it nice to write manual proofs. If you wanted to switch contexts, stop doing proofs in Haskell, and try to prove the things in the style of that a proof-assistant would allow you to do. (FM7)*

Additionally, the intermediate steps in manual proofs make the verification process slower, introducing issues of scalability and solver limitations as presented in the following section.

4.6 Scalability and solver limitations

Experienced

As the complexity of proofs and code increases, verification performance and limitations in the SMT solver also become issues, as mentioned by 6 of our 7 **Experienced Users**.

4.6.1 Increase of compilation time. For small examples, LiquidHaskell's compilation time is negligible, providing developers with almost instant feedback on their code. However, as programs and proofs become more complex, developers said the compilation time increases (FM3, FM4, FM7). One of the reasons is type inference, which requires calling a Horn solver, and not an SMT directly. The number of calls grows with the available variables in context and available predicates [36]. During development, this translates into an impractical feedback loop, where developers need to wait a long time to check if their code is correct or not:

> “ *So, I think if you ran the whole proof in the end, it took like, like, I don't know. 5h or so, maybe longer for the thing to verify. (FM4)*

The verifier also gets slower with manual proofs where it needs to reason about one equality at a time, and the solver ends up taking extra time to prove each of the conditions, as FM7 mentioned. To overcome this issue, FM7 and CR1 commented out the proofs that took the longest time or that were not necessary to prove at that moment, and then uncomment them when they were needed.

4.6.2 Internal Solver Limitations. During the liquid type verification, the SMT Solver works almost as a black box that receives the predicates and tries to prove them. However, there are cases where the SMT solver may fail unpredictably, and differences across solver versions or implementations can affect verification. When there is a failure of this kind, a developer cannot understand where the issue lies since there is no hint from the verification about what happened.

> *Z3 also can break down and fail. For example, it can enter an infinite loop and never type check. (...) [To understand the issue] it's a bit tough because you need to know also about Z3, and how to test these things to understand what is the piece that is malfunctioning. (FM2)*

Additionally, FM6 mentioned that some functions (e.g., non-linear) cannot be directly translated into SMT-LIB,[5] the library used for connecting with several SMT solvers, and the developer needs to have a good understanding of what can and cannot be proved in these cases.

4.7 Unhelpful Error Messages

New Experienced

Error messages are a common source of issues for novices in any programming language [3]. This barrier typically improves over time as developers adapt to the style and type of error messages.

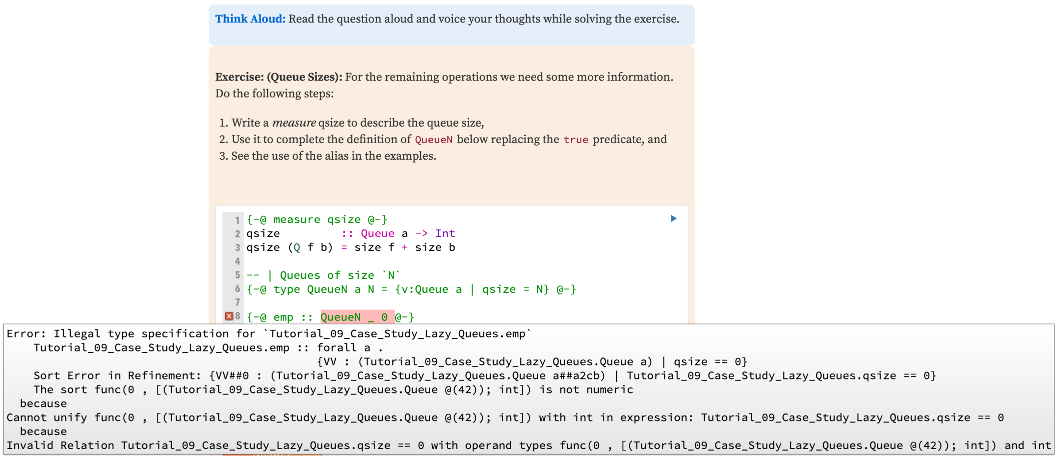
In liquid types, however, these messages have a higher complexity degree since they depend on the reasoning performed by the SMT-Solvers and the many indirections, as hinted by a talk at a workshop in 2022.[6] In our study, we watched **New Users** struggle with error messages from issues inside the liquid type predicates, while both novices and experienced users struggled when error messages didn't align with the code itself.

4.7.1 Errors inside the predicates. First, naturally, participants had issues with **syntax errors** inside the predicates. Error messages did not include enough information for the participants to understand the syntax issue and how to fix it. For example, the `tl` function has the following liquid type: `{-@ tl :: xs:NEList a -> SListN a {size xs - 1} @-}` where we get a non-empty list and return a list with the size of the input list minus one. This was given as an exercise for participants to fix the liquid type. Six participants such as NC8 and NC10 incorrectly used parenthesis ("()") instead of curly brackets ("{}"), and this small change produces the error message: *Error: Cannot parse specification: unexpected "-" expecting bareTyArgP*. This message focuses on an element (-) that is not the main issue and exposes an internal parsing representation (bareTyArgP) of which the developer has no knowledge. When getting this error, participants would try to change the predicates, erase everything, and start again. When the participants could not understand the problem, the interviewer intervenes to explain the small issue that required fixing.

Additionally, error messages produced from **typing errors** inside the predicates, seemed indistinguishable from those produced by verification errors. Error messages show what verification has failed, even when the issue exists within the predicate. This LiquidHaskell error is likely only found when the verification is triggered; however, it was hard for the participants to distinguish between the two types of errors. Figure 3 shows an example of an error message that participant NC1 got when completing the exercise. In this case, the participant created the measure `qsize` in lines 1-3, and then used it to define the type alias `QueueN` in line 6 that represents queues of size N. This type alias is then used in line 8 to define that empty lists have 0 elements in them. In this example, however, there is a mistake in line 6, since there is a missing argument in the use of the measure `qsize` where it should be applied to the queue `v`. The error message presented to the user lacks clarity about the issue's source since it shows a failed attempt to verify the conditions, making it confusing for developers to distinguish this typing error from a verification error.

4.7.2 Error Indirection and Code Mismatch. From the example in Figure 3, we can see that the highlighting appears in the line where the type alias is used, rather than on the line where the error actually occurs in the type definition. Again, it hints at where the error is found within the internal verification process, but it does not help the developer understand where the issue is in

[5] https://smt-lib.org/
[6] https://www.youtube.com/watch?v=_SBqwdSFLk8



Think Aloud: Read the question aloud and voice your thoughts while solving the exercise.

Exercise: (Queue Sizes): For the remaining operations we need some more information. Do the following steps:

1. Write a *measure* qsize to describe the queue size,
2. Use it to complete the definition of `QueueN` below replacing the `true` predicate, and
3. See the use of the alias in the examples.

```
{-@ measure qsize @-}
qsize          :: Queue a -> Int
qsize (Q f b) = size f + size b

-- | Queues of size `N`
{-@ type QueueN a N = {v:Queue a | qsize = N} @-}

{-@ emp :: QueueN _ 0 @-}
```

```
Error: Illegal type specification for `Tutorial_09_Case_Study_Lazy_Queues.emp`
    Tutorial_09_Case_Study_Lazy_Queues.emp :: forall a .
                                              {VV : (Tutorial_09_Case_Study_Lazy_Queues.Queue a) | qsize == 0}
    Sort Error in Refinement: {VV##0 : (Tutorial_09_Case_Study_Lazy_Queues.Queue a##a2cb) | Tutorial_09_Case_Study_Lazy_Queues.qsize == 0}
    The sort func(0 , [(Tutorial_09_Case_Study_Lazy_Queues.Queue @(42)); int]) is not numeric
  because
Cannot unify func(0 , [(Tutorial_09_Case_Study_Lazy_Queues.Queue @(42)); int]) with int in expression: Tutorial_09_Case_Study_Lazy_Queues.qsize == 0
  because
Invalid Relation Tutorial_09_Case_Study_Lazy_Queues.qsize == 0 with operand types func(0 , [(Tutorial_09_Case_Study_Lazy_Queues.Queue @(42)); int]) and int
```

Fig. 3. Error message produced when a participant did not apply `qsize` to the variable `v`, when completing the final exercise after the tutorial.

the code. This location mismatch was a challenge experienced by three of our **New Users** (NC3, NC16, NC17) and two of our **Experienced Users** (FM1, FM3).

Additionally, there is a mismatch between the code presented in the error message and the code that the developer wrote. Error messages present new, internally defined variables that are outside the context of the code written by the developer. Examples include `VV##0` and `Queue a##a2cb`, which are internal representations integral for verification but have no meaning to the developer, as they do not correspond directly to any implemented code.

Exposing the internal representations used in the verification is not helpful for new users unfamiliar with the tool's inner workings and can hinder the learning process. Interestingly, advanced users also reported similar issues with understanding the output of the SMT solver in the error messages.

> *Well, tell me what the type error is. No, you get a message like "5 is not less than 4". I mean, sure, I know that. But what does it mean? (FM1)*

In fact, all of our **Experienced Users** reported challenges with LiquidHaskell error messages, indicating this issue persists beyond initial use and worsens with larger, more complex codebases. They noted that error messages with large queries are challenging to navigate, often displaying a full screen of failed Horn Clauses that reveal the SMT Solver's internal reasoning but cannot be mapped back to the source code. Without this mapping or identification of the specific failing line, developers struggle to diagnose errors and implement appropriate fixes.

> *Sometimes [the error message] is useful, sometimes is noise that you see (...) a whole screen of [predicates] (...) that doesn't mean anything to you in that context. (CR1)*

In many cases, knowing about the internal implementation of LiquidHaskell helps understand the issues, as mentioned by FM2. But, a regular developer that wants to use liquid types should not be required to have a prior knowledge of the inner workings of the tool to understand the error.

Given this issue with error messages, developers came up with several ways for understanding the errors. While working on their project, CR1 told us that they read a specific part of the error message where it mentions the missing verification (e.g., *"is not a subtype of"*) and related that back to the source code that should have the mentioned type. However, that doesn't always work, and participants mentioned other debugging strategies, such as manually shrinking the verification conditions that are sent to the solver to pinpoint the error (FM6), executing the function to test edge cases (FM2), and making the proofs on paper to understand what steps are missing (FM3).

4.8 Limited IDE support

New Experienced

LiquidHaskell can also be used as an IDE plugin that detects errors and displays messages within the editor, similarly to what our participants had in the tutorial. Since developers are used to having a wide variety of tools to help them write code, they expect more support from the IDEs with suggestions and help from the tools they use.

New Users, since they were inside a very controlled environment, mentioned a lack of context about what functions and type aliases were available in the context. For example, three participants (NC3, NC4, NC10) mentioned that did not know what built-in functions were available and annotated with liquid types. During the tutorial, for creating the measure for the size of a list, NC3 tried to use the function `length`, but since it was not imported, it was impossible to use in this case. Additionally, five participants found it hard to remember what type aliases were already defined and what type aliases related to a certain type. This made, NC16, for example, consider writing the full predicates inline instead of using type aliases. Therefore, the lack of these simple suggestions would help developers use the available features.

Experienced Users also mentioned that there was a lack of feedback about the reasoning of the SMT solver and not enough context for them recover from the errors (FM6, FM7). Moreover, the lack of syntax highlighting for the predicates can make it harder to read and understand them, and even find syntax errors (FM6). Writing liquid types as comments by itself can also be a challenge for developers, since comments are usually seen as just optional information in the code and not something that is directly used by the compiler. One participant even mentioned that writing liquid types as comments made it feel like what they were writing was "fake."

4.9 Insufficient learning and reference resources

Experienced

LiquidHaskell is not yet a very popular tool, and therefore, the resources available for developers to learn it are scarce, which is to be expected as mentioned by 4 of our **Experienced Users**. Developers, however, are used to having many resources at their disposal when they try to learn and use a new language. Therefore, they struggled when googling and not finding for solutions to their problems (FM2, FM3), or when they could not find documentation with examples and explanations for more complex features outside of the tutorial (FM1, FM3, FM6). To find more references, participants mentioned copy-pasting their problems on gitgub issues to check if there are similar questions (FM2), leaving new issues if they don't find the answer and checking for the source code itself (FM6). Additionally, they mentioned talking to the researchers behind the creation of LiquidHaskell when they could not find the answer to their problems (FM1, FM3).

There are also examples in research papers using LiquidHaskell, but they are usually "quite easy, because it's always for beginners" (FM3), and some features that are helpful during the debugging process might not appear in these examples:

> *[It would have helped] If I knew the trick with the assumptions earlier, as well as assertions. (FM3)*

There are also options for theorem proving that are commonly used in advanced settings, but a participant mentioned they were not easy to understand such as `reflection` or `ple`[7](FM6). Advanced features like these, however, are not taught in the tutorial's original version, so developers need to learn them by themselves, which can be a barrier for using LiquidHaskell in their projects.

4.10 Complex Installation and Setup

Experienced

The first step to move from LiquidHaskell 's learning playground to applying it in a complete project is installing it locally. The installation process, however, is not straightforward, and developers

[7]https://ucsd-progsys.github.io/liquidhaskell/options/

faced several issues (FM1, FM2, FM7). Initially, there were two main options for installing the tool, either via `cabal` or `stack`, and developers had to choose one. Even these options, however, can be confusing for someone not very familiar with Haskell. During installation, there was also a need to manage dependencies and ensure their versions were compatible with the version of LiquidHaskell being installed. For example, for some time, Z3, the inner SMT solver, was not statically linked, so the developer also needed a Z3 binary present, as FM1 commented.

All of these small steps might seem trivial when one already knows how to deal with them, but for most developers, this is an extra effort that they need to put in to start using the tool.

> “ *I want to open a Haskell project, tick a box and say, yes, I want liquid types there, and the whole thing needs to be managed for me, and the only thing I want to do is say okay. (FM2)*

Since the start of the LiquidHaskell implementation, however, there have been several improvements to how the installation process is done, and the tool is now more user-friendly, as mentioned by FM7, but it is still not up to par with most of the tools that developers are used to. One of our **New Users** tried to install LiquidHaskell before the interview and reported that he failed:

> “ *I mean, I tried to install LiquidHaskell (...) before with this interview today, and I failed miserably. So that's one reason why I might not be too inclined to use it in the future. (NC17)*

4.11 Threats to validity

Our study has several internal threats to validity. First, the version of LiquidHaskell used for the tutorial is not the most recent, as the latest version includes breaking changes that would make the tutorial infeasible. We acknowledge that newer versions can have several improvements, given, for example, the active improvement on error messages,[8] but the causes for confusion reported by participants still stand. The versions used by the **Experienced Users** might also be outdated, potentially influencing participants' experiences, as feature changes could impact usability. Additionally, the tutorial and exercise for **New Users** was a condensed 2-hour version, designed for practical participant recruitment, which excluded some advanced features of LiquidHaskell. This limitation may have affected participants' understanding and the usability barriers they faced. Individual differences, such as varying prior exposure to Haskell and formal verification, may have influenced how participants engaged with the tutorial and used LiquidHaskell. Finally, while most analysis was conducted by a single researcher, results were discussed collectively amongst all authors to ensure diverse perspectives and enhance the credibility of the findings.

External threats to validity include the study's small sample size and the artificial nature of the tutorial environment. With only a few participants from diverse backgrounds, our findings may not generalize to a broader population of potential LiquidHaskell users.

The tutorial exercise focused on implementing a complex data structure, which may not represent the typical tasks developers would use liquid types for. Nonetheless, three of our experienced participants had projects related to data structure analysis. Additionally, the controlled tutorial setting with simplified exercises and direct interviewer assistance doesn't mirror real-world development challenges where developers work independently on complex projects.

5 Discussion and Implications

We distilled the nine barriers identified in our study into three main categories that capture the issues developers faced: *Verification* challenges, *Developer Experience* challenges, and *Scalability* challenges. Figure 2, presented at the start of the results section, summarizes the distribution of these challenges across the categories. We have encountered these barriers in LiquidHaskell, and some specific examples are unique to this language, but there are connections with other systems

[8] https://github.com/ucsd-progsys/liquidhaskell/pull/2473

that use liquid types, other verification tools, and advanced type systems. In this section we will discuss the implications of our findings for the implementation of these systems.

Verification Challenges. These barriers are related to characteristics of verifying code properties, which include unclear divide between Haskell and LiquidHaskell, confusing verification features and unfamiliarity with proof engineering.

The benefits of performing the verification directly in the implementation (Section 4.1) can be hindered by the unclear divide between Haskell and LiquidHaskell (Section 4.2). Given that Haskell code needs to be changed in the presence of liquid types, other implementations of liquid types should not rely on the idea that the underlying code will not have to change and will easily adapt to the verification, especially if there are complex properties to verify. As for the naming mismatches that confused participants, it could be helpful to define or make explicit the language conventions, for example through the use of a style checker with warnings for mismatches. These challenges are mostly related to liquid types being a layer on top of a base language, so other implementations that use liquid types as an add-on, such as Flux or LiquidJava, should be aware of these issues. Another option could be using a language that has liquid types natively, as mentioned by NC13 during the interview. Languages, such as Aeon [27], that have liquid types as first class citizens could help in this regard.

This blurred division between the layers does not help developers understand the verification features (Section 4.3). Lifting functions as measures was a source of doubt for **New Users**, and it directly mixes these two layers, reflecting the Haskell implementation into the specification logic. This idea of lifting functions to the specification level is not unique to LiquidHaskell, since other languages like JML [39] or Dafny [43], have similar features with pure methods [20] or functions,[9] respectively, so it could be helpful to provide more guidance to developers on which functions could be reflected.

Additionally, it is important to help developers understand which parts of the program are used for verification, so they can see the steps of verification more easily. However, to understand these steps, developers need to have familiarity with proof engineering (Section 4.4). Findings in this section apply broadly to formal methods techniques, since developers need to identify system requirements, and how to translate them into invariants that are robust and flexible enough for the proofs. Goldstein et al. [33], also identified this challenge in property based testing, finding that developers do not go out of their way to write specifications but take a more opportunistic approach. In the study, participant CR1 shared that they only see their colleagues using liquid types as a lightweight verification tool with pre- and post-conditions for methods and verifying method calls, rather than for full verification which demands a deeper understanding and finer grained control over proofs. This suggests an opportunity to focus on lightweight verification techniques that can be more easily understood by developers, and still provide some level of verification.

Developer Experience Challenges. All the aspects that affect how developers interact with the tool are part of our category of developer experience challenges, including the barriers of unhelpful error messages (Section 4.7), limited IDE Support (Section 4.8), insufficient learning and reference resources (Section 4.9), and complex installation and setup (Section 4.10).

None of these barriers is unique to LiquidHaskell, but the inner workings of liquid types make some of them more complex. Error messages, for example, are a known challenge in any programming language, specially for novices [45] but also for experts who are learning new features [55]. However, in systems with constraint solving the challenges are amplified as it is harder to identify and recover from errors [61]. Therefore, these messages should be adapted to characteristics such as

[9]https://dafny.org/dafny/DafnyRef/DafnyRef#51431-well-founded-functionmethod-definitions

having multiple locations as sources of the errors, exposing internal representation and reasoning about constraints. For example, Eremondi et al. [24] tackled this issue by using replay graphs and counter-factual solving in the type checking algorithm, and compared the new messages with the ones produced by similar programs in Idris [9] and Agda [8]. There is also recent work in LiquidHaskell that shows that the feedback from failed typing is not just an implementation issue but also a consequence of the design and the underlying use of automation [57], raising more interesting research questions on how to relay the relevant information to developers.

Limited IDE support also affects the developer experience, and the problems found in LiquidHaskell can be extended to other languages. For example, the impression that specifications written in comments without highlighting are taken less seriously can be extended to other languages that use similar approaches (e.g., JML, PyContracts,[10] Frama-C [38]) and simple efforts like syntax highlighting can already help developers find syntactic problems. Having more learning resources available for different expertise levels may enable more developers to use a new tool, but it requires significant community effort to create these resources. These findings relate to prior work on the adoption of programming languages, where Meyerovich et al. [46] saw that extrinsic factors such as existing code, existing expertise, and open source libraries are dominant drivers of adoption. Finally, complex installation and setup is a common issue for tools, but that has a large impact on developers' first impression, as one participant mentioned:

> “ *[Developers] will probably go on the first day of their job and test it, and if they can use it in the first 5 min, they'll probably try to use it later on. If they cannot set it up on the first 5 min, they'll probably give up and don't touch it for at least a couple of months. (FM2)*

Scalability Challenges. The scalability challenges are mostly related to the performance and limitations of automation with larger projects with complex proofs and implementations. Therefore, the barriers of scalability and solver limitations (Section 4.6) as well as limitations of automation and manual proof flexibility (Section 4.5) are captured in this category.

With complex proofs and large implementations, participants saw a slowdown in the verification process, and even mentioned that this was a reason for not using LiquidHaskell in projects that already have a large compilation time (FM2). This issue can also be present for other verification tools that also use SMT-solvers in their backend, such as Dafny and Why3 [26]. However, improving the performance of the verifier and of the SMT Solver are both significant research problems, both from algorithmic and operational standpoints. Open questions include selecting the best tactics within the SMT solver, what are the best strategies for caching and parallelization, or exploring how liquid type checking can be deferred to Continuous Integration. One participant (FM4) proposed using SMT certificates for library verification, to allow programs to incorporate these libraries without having to verify them again. Moreover, the polymorphic liquid type inference can be costly in large programs as it grows with the number of variables and predicates. To mitigate this problem, it could be helpful to prioritize a subset of context variables similarly to how Piotrowski et al. [50] use machine learning for premise selection in Lean or how Automated Program Repair selects ingredients [60].

Additionally, there is a lack of understanding of what can be proved with the SMT Solver, which also leads to the issues mentioned in Section 4.5. The theory of liquid types assumes that predicates are decidable, however, actual tools are lenient and allow developers to write syntactically undecidable predicates since transformations within the SMT solver can sometimes convert them into decidable ones. For instance, multiplication between two integer variables belongs to the Non-linear arithmetic theory which is not complete in the Z3 SMT solver,[11] but with additional

[10]https://andreacensi.github.io/contracts/
[11]https://microsoft.github.io/z3guide/docs/theories/Arithmetic/

constraints on the variables, it can be converted into a decidable form (e.g., $a * b > 0$ && $b == 2$ can be converted to $a + a > 0$). Therefore, it would be helpful for liquid type implementations to give users more feedback on what is decidable or not inside the language of predicates.

The tension between automatic and manual-style proofs is another challenge developers felt. Manual-proof style constructs are helpful to complement liquid types, but it is not clear when to introduce them, and they do not provide the flexibility of other interactive provers like Agda or Rocq [23]. One possible mitigating strategy is mentioned in Explicit Refinement Types [32], addressing this tension between implicit and explicit proof objects. This tension also occurs in other systems like Why3. WhyML specifications can be discharged to either Automated Theorem Solvers or Interactive Theorem Solvers,[12] but the documentation does not help users in understanding when to use each.

Overall, the barriers found in our study have implications for LiquidHaskell, liquid types and other verification tools. Therefore, in the next section we situate our research within the existing literature on usability barriers of advanced type systems and verification tools.

6 Related Work

Most studies comparing languages and their features analyze open-source repositories rather than conducting developer interviews or observations [11, 12, 48]. Studies with developers present numerous challenges for research teams [21], including designing appropriate tasks, building and integrating tools for the study, and recruiting participants with the required expertise, all difficulties we encountered in our own study. However, overcoming these challenges is important since applying Human-Computer Interaction (HCI) methods into the area of Programming Languages can help reveal developers' actual needs and problems [13, 47]. For example, Coblenz et al. [17] presented such an approach in PLIERS, a process for designing languages with users' input and evaluating the resulting design with user studies and evaluated different advanced features in two programming languages, Glacier [18] and Obsidian [16]. As another example, Lubin and Chasins [44] used grounded theory analysis to study how programmers write functional and statically typed code.

Within the specific context of our study, LiquidJava [29] stands out as the only other research that has focused on the usability of liquid types. The authors proposed a liquid types extension for Java and evaluated them with 30 participants who, like our **New Users**, were introduced to liquid types for the first time. All participants showed interest in adopting liquid types, motivating our present work. However, participants also found it difficult to understand complex specifications without access to documentation, encountered unclear error messages, desired additional plugin features (such as autocomplete), and struggled with the installation process. These negative comments align with our results for developer experience challenges, suggesting that the barriers faced by developers using liquid types are not unique to LiquidHaskell but are common across different implementations of liquid types. The prior study, however, does not go in-depth into these challenges, and our study complements it as we focus on the more mature Liquid Haskell tool, we identify more barriers, and we explore in detail why these challenges are relevant and how they can occur.

Beckert and Grebing [4] applied the cognitive dimensions framework [34] to the KeY verification framework, which verifies Java code with JML specifications. The authors created a questionnaire based on the cognitive dimensions and had participants evaluate the tool using the questionnaire. Their findings reveal that participants valued features such as proof presentation and guidance, documentation, change management and efficient automatic proof search. Participants had at least a few months of experience with the tool, similarly to our more experienced LiquidHaskell users.

[12] https://www.why3.org/doc/itp.html

These results align with our findings, as our participants emphasized needing to understand which conditions are being checked versus which are missing to prove the properties, along with requiring better documentation and learning resources.

Juhošová et al. [37] conducted a study on the learning obstacles with interactive theorem provers, using Adga, and found that participants were faced with obstacles from both theory and implementation. These obstacles include unfamiliar concepts and complex theory, which required more familiarity with the underlying principles and a different way of thinking when compared to mainstream programming languages, and inadequate ecosystem support, mentioning that installation is difficult, error messages unclear, and supporting materials were incomplete. These findings are also in line with our results from both verification and developer experience.

Recent research examining Rust's ownership system has also identified challenges similar to those faced by developers in our study. Crichton et al. [19] identified users' misconceptions of ownership and created a conceptual model to improve learning and visualization resources. Some of these misconceptions have similar roots to the issues in our study, since in both cases developers need to understand deeper reasons for the failed verification and how to fix them. Additionally, liquid types could also benefit from a similar study and improvement in learning materials. Both Zhu et al. [62] and Coblenz et al. [15] identified syntactic challenges, late delivery of error messages and the opacity of Rust errors, similarly to what our participants reported of liquid types.

For error messages, there have been multiple studies focused on the challenges and guidelines for designing better error messages in compilers and IDEs. The issues we report on error messages fall into the categories identified by Becker et al. [2]. For example, our error indirection and code mismatch barriers are related to both the mapping problem, where code transformations complicate error messages, and error localization, where the reported lines are not the actual problem.

In the presence of constraint-based type inference these issues become harder [6] since the error messages reflect the underlying constraint solving algorithms and there are several locations in the code that can be the source of the error. Liquid types might need solutions like those introduced in the ChameleonIDE [28] to overcome problems in traditional IDE's such as limited location, message bias, and poor explanations. Participants of their user study acknowledge that their particular features helped them complete harder tasks.

Our results on liquid types also align with an expert survey on formal methods presented in 2020 [31], where participants mentioned that formal methods are not being more adopted in industry due to many human factors, such as engineers lacking proper training, and developers being reluctant to change their way of working. The survey also defines as research priorities the need for scalability, with more efficient verification algorithms, applicability, for developing more usable software tools and acceptability, with enhanced integration into software engineering processes. These priorities and limitations also fit into our three categories of challenges, and show that most of the barriers we found are not unique to LiquidHaskell, but are common to other verification tools and advanced type systems.

7 Conclusion

This paper presents the first study on the barriers to adopting liquid types, focusing on LiquidHaskell. We conducted interviews and observations with 19 participants, including 12 newcomers and 7 experienced users, to explore their challenges when using liquid types in their projects. Our findings were distilled into nine key barriers, many of which not only highlight limitations in the implementation but also suggest promising directions for future research in the field of liquid types and verification tools. The insights gained through collaboration with developers can inform the design and development of current and future implementations of liquid types.

8 Data-Availability Statement

The materials from our qualitative study are publicly accessible through our research artifact [30]. This repository includes all study materials: recruitment documents, protocols, interview guides, and a link to the customized LiquidHaskell tutorial used during observation sessions with novice users. We have also included our thematic analysis codebook with definitions and representative quotes that directly connect to the findings discussed in the paper. These materials document our methodological approach and provide the foundation for future replication efforts.

9 Acknowledgements

We are deeply grateful to all study participants whose time and contributions made this research possible. We thank the reviewers for their thoughtful feedback that strengthened our work. Additionally, we would like to thank Laura Dabbish, Bogdan Vasilescu, Tiago Guerreiro, and Paulo Canelas for their suggestions and feedback on this work. This work is supported by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under the fellowship PRT/BD/154254/2021, the LASIGE Research Unit, ref. UID/00408/2025, and the RAP project (https://doi.org/10.54499/EXPL/CCI-COM/1306/2021).

References

[1] Pascal W. M. Van Gerven Babu Noushad and Anique B. H. de Bruin. 2024. Twelve tips for applying the think-aloud method to capture cognitive processes. *Medical Teacher* 46, 7 (2024), 892–897. https://doi.org/10.1080/0142159X.2023.2289847 arXiv:https://doi.org/10.1080/0142159X.2023.2289847 PMID: 38071621.

[2] 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 *Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE-WGR*. ACM, 177–210. https://doi.org/10.1145/3344429.3372508

[3] Brett A. Becker and Keith Quille. 2019. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. ACM, 338–344. https://doi.org/10.1145/3287324.3287432

[4] Bernhard Beckert and Sarah Grebing. 2012. Evaluating the Usability of Interactive Verification Systems. In *International Workshop on Comparative Empirical Evaluation of Reasoning Systems*, Vol. 873. CEUR-WS.org, 3–17. https://dblp.org/rec/conf/cade/BeckertG12.bib

[5] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. In *ACM Trans. Program. Lang. Syst.*, Vol. 33. 8:1–8:45. https://doi.org/10.1145/1890028.1890031

[6] Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (2023), 29 pages. https://doi.org/10.1145/3622812

[7] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 2016. Hammering towards QED. *J. Formaliz. Reason.* 9, 1 (2016), 101–148. https://doi.org/10.6092/ISSN.1972-5787/4593

[8] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda,a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

[9] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.

[10] Chris Brown. 2022. Nudging developers to participate in SE research. https://api.semanticscholar.org/CorpusID:252539209

[11] Oscar Callaú, Romain Robbes, Éric Tanter, David Röthlisberger, and Alexandre Bergel. 2014. On the use of type predicates in object-oriented software: the case of smalltalk. *SIGPLAN Not.* 50, 2 (oct 2014), 135–146. https://doi.org/10.1145/2775052.2661091

[12] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. 2017. Exploring regular expression comprehension. In *International Conference on Automated Software Engineering, ASE*. IEEE Computer Society, 405–416. https://doi.org/10.1109/ASE.2017.8115653

[13] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: better together. *Commun. ACM* 64, 8 (2021), 98–106. https://doi.org/10.1145/3469279
[14] Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent types for JavaScript. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 587–606. https://doi.org/10.1145/2384616.2384659
[15] Michael Coblenz, April Porter, Varun Das, Teja Nallagorla, and Michael Hicks. 2023. A Multimodal Study of Challenges Using Rust. https://doi.org/10.1184/R1/22277326.v1
[16] Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2020. Can advanced type systems be usable? An empirical study of ownership, assets, and typestate in Obsidian. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 132:1–132:28. https://doi.org/10.1145/3428200
[17] Michael J. Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2021. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. *ACM Trans. Comput. Hum. Interact.* 28, 4 (2021), 28:1–28:53. https://doi.org/10.1145/3452379
[18] Michael J. Coblenz, Whitney Nelson, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2017. Glacier: transitive class immutability for Java. In *International Conference on Software Engineering (ICSE)*. IEEE / ACM, 496–506. https://doi.org/10.1109/ICSE.2017.52
[19] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types in Rust. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 265 (Oct. 2023), 29 pages. https://doi.org/10.1145/3622841
[20] Ádám Darvas and Peter Müller. 2006. Reasoning About Method Calls in Interface Specifications. *J. Object Technol.* 5, 5 (2006), 59–85. https://doi.org/10.5381/JOT.2006.5.5.A3
[21] Matthew C. Davis, Emad Aghayi, Thomas D. LaToza, Xiaoyin Wang, Brad A. Myers, and Joshua Sunshine. 2023. What's (Not) Working in Programmer User Studies? *ACM Trans. Softw. Eng. Methodol.* 32, 5 (2023), 120:1–120:32. https://doi.org/10.1145/3587157
[22] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
[23] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chetan Murthy, Catherin Parent, Christine Paulin-Mohring, and Benjamin Werner. 1992. *The COQ Proof Assistant: User's Guide: Version 5.6.* INRIA.
[24] Joseph Eremondi, Wouter Swierstra, and Jurriaan Hage. 2019. A framework for improving error messages in dependently-typed languages. *Open Comput. Sci.* 9, 1 (2019), 1–32. https://doi.org/10.1515/comp-2019-0001
[25] K. Anders Ericsson and Herbert A. Simon. 1993. *Protocol Analysis: Verbal Reports as Data.* The MIT Press. https://doi.org/10.7551/mitpress/5657.001.0001
[26] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming ESOP (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8
[27] Alcides Fonseca, Paulo Santos, and Sara Silva. 2020. The Usability Argument for Refinement Typed Genetic Programming. In *Parallel Problem Solving from Nature - PPSN (Lecture Notes in Computer Science, Vol. 12270)*. Springer, 18–32. https://doi.org/10.1007/978-3-030-58115-2_2
[28] Shuai Fu, Tim Dwyer, Peter J. Stuckey, Jackson Wain, and Jesse Linossier. 2023. ChameleonIDE: Untangling Type Errors Through Interactive Visualization and Exploration. In *International Conference on Program Comprehension, ICPC*. IEEE, 146–156. https://doi.org/10.1109/ICPC58990.2023.00029
[29] Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. In *International Conference on Software Engineering (ICSE)*. IEEE, 1520–1532. https://doi.org/10.1109/ICSE48619.2023.00132
[30] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Artifact for "Usability Barriers for Liquid Types". Zenodo repository. https://doi.org/10.5281/zenodo.15044759 Artifact containing research materials for the qualitative study on developer experiences with LiquidHaskell.
[31] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. 2020. The 2020 Expert Survey on Formal Methods. In *Formal Methods for Industrial Critical Systems, FMICS* (Vienna, Austria). Springer-Verlag, 3–69. https://doi.org/10.1007/978-3-030-58298-2_1
[32] Jad Elkhaleq Ghalayini and Neel Krishnaswami. 2023. Explicit Refinement Types. *Proc. ACM Program. Lang.* 7, ICFP (2023), 187–214. https://doi.org/10.1145/3607837
[33] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *International Conference on Software Engineering (ICSE)* (Lisbon, Portugal). Association for Computing Machinery, New York, NY, USA, Article 187, 13 pages. https://doi.org/10.1145/3597503.3639581
[34] Thomas R. G. Green and Marian Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *J. Vis. Lang. Comput.* 7, 2 (1996), 131–174. https://doi.org/10.1006/JVLC.1996.0009

[35] Karen Holtzblatt and Hugh Beyer. 2016. *Contextual Design, Second Edition: Design for Life* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[36] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* 6, 3-4 (2021), 159–317. https://doi.org/10.1561/2500000032

[37] Sára Juhošová, Andy Zaidman, and Jesper Cockx. 2025. Pinpointing the Learning Obstacles of an Interactive Theorem Prover. *International Conference on Program Comprehension, ICPC* (2025). https://sarajuhosova.com/assets/files/2025-icpc.pdf Accepted.

[38] Nikolai Kosmatov and Julien Signoles. 2016. Frama-C, A Collaborative Framework for C Code Verification: Tutorial Synopsis. In *Runtime Verification (RV) (Lecture Notes in Computer Science, Vol. 10012)*. Springer, 92–115. https://doi.org/10.1007/978-3-319-46982-9_7

[39] Gary T Leavens, Albert L Baker, and Clyde Ruby. 1998. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*. Citeseer, 404–420. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=397cb3c2ad6569aef081c282671d17937df483d0

[40] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1533–1557. https://doi.org/10.1145/3591283

[41] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. STORM: Refinement Types for Secure Web Applications. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 441–459. https://www.usenix.org/conference/osdi21/presentation/lehmann

[42] Nico Lehmann, Cole Kurashige, Nikhil Akiti, Niroop Krishnakumar, and Ranjit Jhala. 2025. Generic Refinement Types. *Proc. ACM Program. Lang.* 9, POPL (2025), 1446–1474. https://doi.org/10.1145/3704885

[43] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (Lecture Notes in Computer Science, Vol. 6355)*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[44] Justin Lubin and Sarah E. Chasins. 2021. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485532

[45] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind your language: on novices' interactions with error messages. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, Oregon, USA) *(Onward! 2011)*. ACM, 3–18. https://doi.org/10.1145/2048237.2048241

[46] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical analysis of programming language adoption. In *International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 1–18. https://doi.org/10.1145/2509136.2509515

[47] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (2016), 44–52. https://doi.org/10.1109/MC.2016.200

[48] Sebastian Nanz and Carlo A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *International Conference on Software Engineering, ICSE*. IEEE Computer Society, 778–788. https://doi.org/10.1109/ICSE.2015.90

[49] Chris Okasaki. 1995. Simple and Efficient Purely Functional Queues and Deques. *J. Funct. Program.* 5, 4 (1995), 583–592. https://doi.org/10.1017/S0956796800001489

[50] Bartosz Piotrowski, Ramon Fernández Mir, and Edward W. Ayers. 2023. Machine-Learned Premise Selection for Lean. In *Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2023 (Lecture Notes in Computer Science, Vol. 14278)*. Springer, 175–186. https://doi.org/10.1007/978-3-031-43513-3_10

[51] Mary Elizabeth Raven and Alicia Flanders. 1996. Using Contextual Inquiry to Learn about Your Audiences. *SIGDOC Asterisk J. Comput. Doc.* 20, 1 (feb 1996), 1–13. https://doi.org/10.1145/227614.227615

[52] Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. 2012. CSolve: Verifying C with Liquid Types. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 744–750. https://doi.org/10.1007/978-3-642-31424-7_59

[53] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Programming Language Design and Implementation (PLDI)*. ACM, 159–169. https://doi.org/10.1145/1375581.1375602

[54] Johnny Saldaña. 2009. *The Coding Manual for Qualitative Researchers.* SAGE Publications.

[55] V. Javier Traver. 2010. On Compiler Error Messages: What They *Say* and What They *Mean*. *Adv. Hum. Comput. Interact.* 2010 (2010), 602570:1–602570:26. https://doi.org/10.1155/2010/602570

[56] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: experience with refinement types in the real world. In *ACM SIGPLAN symposium on Haskell*. ACM, 39–51. https://doi.org/10.1145/2633357.2633366

[57] Robin Webbers, Klaus von Gleissenthall, and Ranjit Jhala. 2024. Refinement Type Refutations. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 305 (Oct. 2024), 26 pages. https://doi.org/10.1145/3689745

[58] Christopher D. Wickens, John Lee, Yili D. Liu, and Sallie Gordon-Becker. 2003. *Introduction to Human Factors Engineering (2nd Edition)*. Prentice-Hall, Inc., USA.

[59] Hyrum Wright, Titus Delafayette Winters, and Tom Manshreck. 2020. *Software Engineering at Google*. O'Reilly Media, Inc.

[60] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, Haoye Tian, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F. Bissyandé. 2021. Where were the repair ingredients for Defects4j bugs? *Empir. Softw. Eng.* 26, 6 (2021), 122. https://doi.org/10.1007/S10664-021-10003-7

[61] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proc. ACM Program. Lang.* 8, POPL, Article 68 (Jan. 2024), 28 pages. https://doi.org/10.1145/3632910

[62] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. 2022. Learning and programming challenges of rust: a mixed-methods study. In *International Conference on Software Engineering, ICSE* (Pittsburgh, Pennsylvania). ACM, 1269–1281. https://doi.org/10.1145/3510003.3510164

Received 2024-11-15; accepted 2025-03-06