Thesis Proposal
Usable Liquid Types for Java

Catarina Ventura Gamboa

February 2026

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich (Carnegie Mellon University), Co-Chair
Alcides Fonseca (University of Lisbon), Co-Chair
Joshua Sunshine (Carnegie Mellon University)
Ruben Martins (Carnegie Mellon University)
Tiago Guerreiro (University of Lisbon)
Ranjit Jhala (University of California, San Diego)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Abstract

Liquid Types extend traditional type systems with logical predicates, enabling compile-time detection of bugs ranging from divisions by zero to complex protocol violations. Despite this potential, Liquid Types have not achieved mainstream adoption.

This thesis aims to make Liquid Types expressive and practical for verifying Java programs. We first identify the usability barriers that hinder Liquid Types adoption through an empirical study with 19 developers using LiquidHaskell, the most mature implementation of Liquid Types. Our analysis reveals nine distinct barriers spanning three themes: developer experience challenges such as unhelpful error messages and limited IDE support, scalability issues with complex codebases, and difficulties understanding the verification process. These findings provide a foundation for addressing the gap between Liquid Types' expressive power and their practical usability. Building on these insights, we design LiquidJava, a usability-oriented refinement type system for Java that follows developers' feedback. Through a participatory design methodology, we conducted surveys to determine the syntax and evaluated the prototype with 30 Java developers. Our results show that LiquidJava helps users detect and fix more bugs, with all participants expressing interest in adopting the tool. To further reduce the specification burden, we propose an agentic workflow that automatically generates object state protocols from class documentation using LLM-based techniques, transforming the informal specifications already present in Javadoc into verifiable LiquidJava annotations. We extend the type system with *Latte*, a lightweight aliasing tracking mechanism that enables verification of Java programs using mutable state and object references, a pervasive pattern in real-world code that previous approaches could not handle. Finally, we improve error diagnostics by providing counterexamples, simplified presentations, and interactive debugging tools that help developers understand and resolve verification failures.

Together, these contributions support our thesis that by understanding the usability barriers that hinder Liquid Types adoption, we can design a usable, developer-centric refinement type system for Java that, combined with automated synthesis, improved diagnostics, and aliasing tracking, makes Liquid Types expressive and practical for verifying Java programs.

Keywords: Liquid Types, Refinement Types, Usability, Java, Software Verification, LLM-based Synthesis

Acknowledgments

This work is supported by the Fundação para a Ciência e a Tecnologia (FCT) under LASIGE Research Unit, ref. (UIDB/00408/2020) and (UIDP/00408/2020), the project DACOMICO (PTDC/CCI-COM/2156/2021), the CMU-Portugal project CAMELOT (LISBOA-01-0247-FEDER-045915), the RAP project (EXPL/CCI-COM/1306/2021), and the CMU Portugal Dual PhD program (PRT/BD/154254/2021).

Contents

1 Introduction **1**
1.1 Thesis Statement . . . 2

2 Background and Related Work **3**

3 Usability Barriers of Liquid Types **5**
3.1 Motivation . . . 5
3.2 Background: Overview of LiquidHaskell . . . 6
3.3 Interviews with Developers . . . 7
3.3.1 Study Design . . . 7
3.3.2 Recruitment . . . 10
3.3.3 Qualitative Data Analysis . . . 10
3.4 Results . . . 12
3.4.1 **Experienced Users**' Project Contexts and Motivations . . . 13
3.4.2 Unclear Divide between Haskell and LiquidHaskell New Experienced 14
3.4.3 Confusing Verification Features New Experienced . . . 17
3.4.4 Unfamiliarity with Proof Engineering New Experienced . . . 19
3.4.5 Limitations of Automation and Manual Proof Flexibility Experienced 20
3.4.6 Scalability and Solver Limitations Experienced . . . 21
3.4.7 Unhelpful Error Messages New Experienced . . . 22
3.4.8 Limited IDE Support New Experienced . . . 24
3.4.9 Insufficient Learning and Reference Resources Experienced . . . 25
3.4.10 Complex Installation and Setup Experienced . . . 25
3.4.11 Threats to Validity . . . 26
3.5 Discussion and Implications . . . 27
3.6 Related Work . . . 30
3.7 Summary and Research Directions . . . 32

4 Design of LiquidJava **35**
4.1 Motivation . . . 35
4.2 LiquidJava Design . . . 36
4.2.1 Requirements . . . 37
4.2.2 Syntax Survey . . . 38
4.2.3 LiquidJava Features . . . 40

4.2.4 IDE Integration . . . 42
4.3 User Study . . . 42
4.3.1 Study Configuration . . . 44
4.3.2 Background of Participants . . . 45
4.3.3 Exercises and Results . . . 46
4.3.4 Study Conclusions . . . 54
4.3.5 Threats to Validity . . . 55
4.4 Key Takeaways and Research Directions . . . 55

5 Agentic Synthesis of LiquidJava Annotations 57
5.1 Motivation . . . 57
5.2 Related Work . . . 58
5.3 Approach . . . 59
5.3.1 DFA generation . . . 61
5.3.2 Synthesize tests . . . 61
5.3.3 Refinement Synthesis . . . 62
5.3.4 Implementation Details . . . 64
5.4 Evaluation Methodology . . . 64
5.4.1 Dataset Selection . . . 64
5.4.2 Manual Expert Annotation . . . 65
5.4.3 Automated Specification Generation . . . 66
5.4.4 Comparison Metrics . . . 66
5.5 Progress and Expected Contributions . . . 66

6 Liquid Types and Alias Tracking in LiquidJava 69
6.1 Introduction . . . 69
6.2 Approach . . . 70
6.3 Illustrative example . . . 75
6.4 Related Work . . . 77
6.4.1 Progress and Expected Contributions . . . 78

7 Verification Feedback in LiquidJava 79
7.1 Motivation . . . 79
7.2 Related Work . . . 80
7.3 Design Principles for Verification Feedback . . . 81
7.4 Current Implementation in LiquidJava . . . 83
7.5 Planned User Study . . . 86
7.5.1 Anticipated Threats to Validity . . . 88
7.6 Progress and Expected Contributions . . . 89

8 Timeline 91

9 Conclusion 93

Bibliography 95

1 Introduction

Software errors cost the industry trillions of dollars annually and can have catastrophic consequences for users' lives [90, 120]. The destruction of the Ariane 5 rocket in 1996 [15], the crash of an Israeli spaceship in 2019 [107], and failures and recalls in medical devices [46] all trace back to bugs that developers could have caught earlier in the development process. To reduce the costs and effort of fixing these bugs, developers aim to identify issues as soon as possible, following a "shifting left" methodology [148]. More recently, the rise of AI-assisted programming has introduced new challenges for software reliability. Recent empirical studies find that code generated by GitHub Copilot and similar models is often only partially correct and can embed security weaknesses that developers may overlook during review [106, 151, 154]. This shift amplifies the need for verification techniques that can check code correctness regardless of whether a human or an AI wrote it.

Strong type systems present in modern languages like Java, C#, and Haskell help developers catch many classes of bugs at compile-time, often providing immediate feedback through IDE integration. However, these type systems limit the domain-specific information that developers can express, restricting the class of errors they can detect. Liquid Types [125] address this gap by extending type systems with logical predicates that restrict the valid values for variables and establish relationships between them. For example, a function can require that its input is a positive integer, and the compiler will reject any call that cannot guarantee this property. Since their introduction, researchers have implemented Liquid Types in several languages, including Haskell [142], Rust [94], and JavaScript [36], demonstrating their ability to detect bugs ranging from simple divisions by zero to complex security issues and protocol violations. Despite this potential and the variety of implementations, Liquid Types have not achieved mainstream adoption.

This thesis aims to address this gap by making Liquid Types practical for Java. To this end, we first identify the usability barriers that hinder their adoption through developer studies (Chapter 3). Then, we design a usability-oriented refinement type system for Java, following developers' feedback to adapt Liquid Types to Java (Chapter 4). Building on this foundation, we focus on reducing the specification burden through automated synthesis of liquid types and object state using LLM-based agentic techniques (Chapter 5). We then extend the type system with aliasing tracking to verify Java programs that use mutable state and object references (Chapter 6). Finally, we improve error diagnostics to help developers understand and fix verification failures more effectively (Chapter 7).

1.1 Thesis Statement

By understanding the usability barriers that hinder Liquid Types adoption, we can design a usable, developer-centric refinement type system for Java that, combined with automated synthesis, improved diagnostics, and aliasing tracking, makes Liquid Types expressive and practical for verifying Java programs.

2 Background and Related Work

Refinement types extend traditional type systems with logical predicates that constrain the valid values of variables. These refinements can express that a given variable is always positive `{height:Int | height > 0}`, or a function always returns a value within bounds `{v:Int | v >= x && v <= y}`. Freeman and Pfenning introduced refinement types for ML in 1991 [66], and Rondon et al. proposed Liquid Types in 2008 [125], a decidable subset that enables automatic verification via SMT solvers. LiquidHaskell [142] remains the most mature implementation, extending Haskell with refinement types, type aliases, and measure functions, and other features explore in depth in the *Refinement Types: a Tutorial* [82]. Despite their ability to detect bugs ranging from divisions by zero to security vulnerabilities and protocol violations, liquid types have not achieved mainstream adoption.

Since their initial proposal, there has been an effort to bring refinement types to more mainstream and object-oriented languages. Kazerounian et al. [84] introduced refinements for Ruby, translating to Rosette for verification. Schmid and Kuncak [130] and Vekris et al. [143] added refinements to Scala and TypeScript, respectively, introducing class invariants on immutable fields to model object-oriented aspects. In Java, Stein et al. [131] applied refinement types to stream-based processing but used a fixed type hierarchy, limiting expressiveness. More recently, Flux [94] combined liquid types with Rust's ownership mechanisms for verification of imperative code. However, none of these approaches addressed object state modeling or typestate protocols using refinements, leaving a gap for object-oriented languages where classes and their protocols are fundamental.

Java has a rich ecosystem of specification and verification tools. Design-by-Contract, first proposed within Eiffel [103], relies on pre- and post-conditions to establish contracts between clients and providers. JML (Java Modeling Language) [92] is a popular notation for writing specifications, and OpenJML [44] provides verification support. They provide behavioral interface specifications for Java methods using pre- and post-conditions, but specifications involving quantified assertions require undecidable logic, leading to incomplete verification and false positives that undermine developer trust. For modeling object protocols, Mungo [89] verifies that objects follow specified state machines, but requires separate protocol files and enforces linear usage patterns that prohibit common aliasing scenarios in Java code. The Java Typestate Checker [109] tracks aliases to typestate objects, but focuses on single-class APIs and cannot express constraints that span multiple objects. The Checker Framework [109] provides pluggable type checking with annotations like `@NonNull`, demonstrating that Java developers can work with annotation-based specifications, but its type qualifiers cannot express the rich value constraints that refinement types provide.

These limitations motivate our work, as we aim to combine expressive value constraints, multi-object reasoning, and decidable verification within Java's annotation system.

Applying Human-Computer Interaction methods to Programming Languages can reveal developers' actual needs and problems [33, 113]. Coblenz et al. proposed PLIERS [43], a process for designing languages with users' input, which guided the development of Glacier [41] and Obsidian [42]. Within verification tools, the KeY project [5] applied cognitive dimensions questionnaires and focus groups to understand user interactions with theorem provers. Recently, researchers have studied how languages that incorporate verification are used in practice. For instance, Mugnier et al. [110] interviewed experienced Dafny users to identify how verification impacts software development, and Oliveira et al. [116] study the challenges developers face when using verification-aware languages.

Prior to this thesis, no work had designed refinement type systems with users' input or systematically studied the usability barriers preventing their adoption. Furthermore, existing approaches for Java either lack the expressiveness of liquid types, require learning separate specification languages, or do not integrate object state modeling with refinement types. This thesis addresses these gaps by first understanding why developers struggle with refinement types, then designing a liquid type system for Java based on developers' feedback. Building on this foundation, we combine the type system with automated specification synthesis, actionable error diagnostics, and aliasing tracking to make liquid types more practical for verifying real-world Java programs. We present related work specific to each contribution in the respective chapters.

3 Usability Barriers of Liquid Types

This chapter presents the study on Usability Barriers for Liquid Types [69], published at PLDI 2025, that aims to identify the challenges developers face when using liquid types. We conducted an empirical study with 19 developers, including 12 new users and 7 experienced users of liquid types, to understand the barriers they face when adopting and using liquid types. Our results identify nine distinct barriers that span three main themes: developer experience, scalability challenges, and understanding the verification process.

3.1 Motivation

Liquid types offer more expressive power than traditional type systems, yet they have not achieved widespread adoption. Since their introduction in 2008 [125], there have been many attempts to integrate liquid types in different programming languages, ranging from functional languages like Haskell [142] to more popular languages like C [127], JavaScript [36], and Rust [94]. They detect a wide range of bugs, from simple divisions by zero or out-of-bounds array accesses to more complex security issues [16] and protocol violations [68]. Liquid types have also demonstrated a practical value across various applications. For example, STORM [93] employed LiquidHaskell to describe the data produced and consumed by different layers of an MVC application, LiquidJava [68] 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 [95]. For more examples, Vazou et al. [142] 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. 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

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

Listing 3.1: Example of a type alias and a liquid type annotations in LiquidHaskell.

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.[1]

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.

This chapter will present some background on liquid types and LiquidHaskell (Section 3.2), describe the study design (Section 3.3), present the results (Section 3.4), discuss the results in comparison to other implementations of liquid types and verification techniques (Section 3.5), present related work (Section 3.6), and draw conclusions (Section 3.7).

3.2 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 [82]. By restricting these types, it is possible to reject programs with values outside of the allowed ranges.

Listing 3.1 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 [49]). 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 3.2 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,

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

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

Listing 3.2: Datatype invariants in LiquidHaskell.

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 [82] provides an excellent resource.

3.3 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 [70].

3.3.1 Study Design

Figure 3.1 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 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.

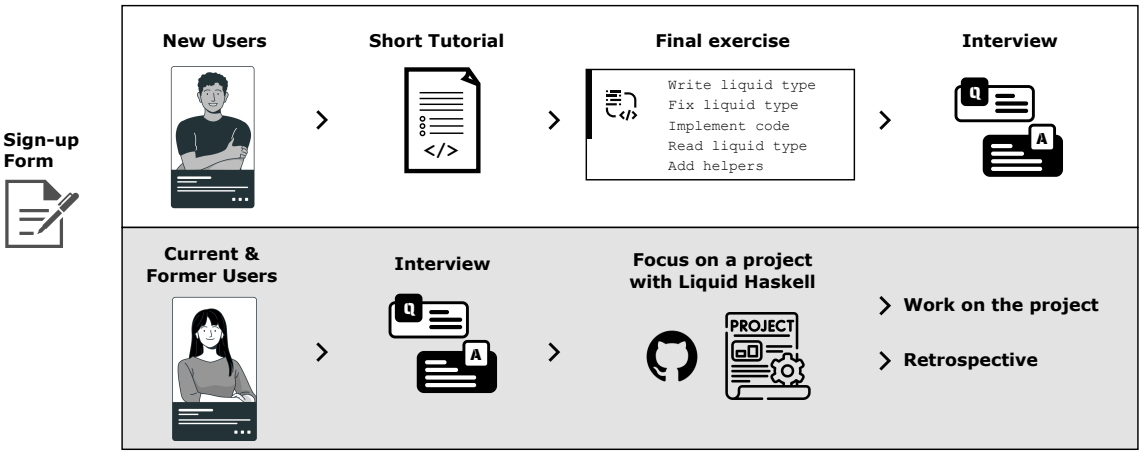


Figure 3.1: Study design with new and experienced users of LiquidHaskell.

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,[2] where we introduce the concepts along with exercises and quiz questions. To minimize the impact of material quality, we used the official LiquidHaskell tutorial,[3] 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 [147].

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** [115], 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

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

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.

Exercise - Observation and Interview During the final exercise, we asked participants to use the Think-Aloud methodology [59] 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 [9].

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 [147], 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 [80, 123] 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.

3.3.2 Recruitment

One of the main challenges of conducting user studies in software engineering is recruiting participants [24]. 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;
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 3.1. 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 3.2. 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.

3.3.3 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 [129] with inductive coding, where the codes emerge during the analysis. For **New Users**, we considered the final

Table 3.1: 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	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 3.2: 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".

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

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.

3.4 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 3.2 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 3.4.1), and then details each identified barrier with examples from the study (Section 3.4.2 through Section 3.4.10). Each identified barrier includes tags indicating which participant groups reported it.

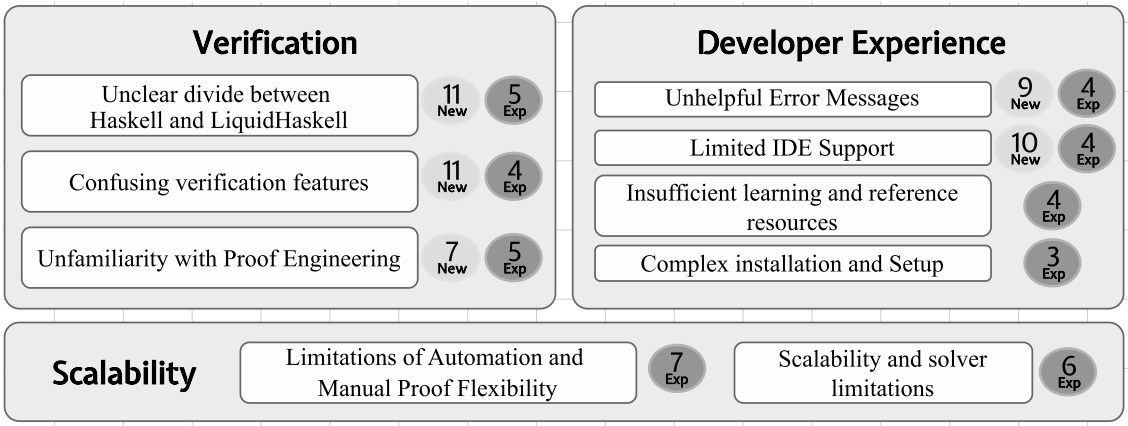


Figure 3.2: Barriers encountered by **New Users** (New) and **Experienced Users** (Exp), with frequency counts showing how many participants in each group mentioned or experienced each barrier.

3.4.1 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.

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

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 3.3 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

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

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 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 3.4.

```
#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 3.4: Shadowing of modules for LiquidHaskell presented by FM4 during their project showcase.

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.

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:

```
1 {-@ cons :: a -> xs:SList a -> SListN a {size xs + 1} @-}
2 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 3.4.7.

3.4.3 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.

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)*

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)*

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.

3.4.4 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.

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 3.4.5.

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.

3.4.5 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 [19]. 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.

3.4.6 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**.

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 [82]. 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.

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,[4] 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.

3.4.7 Unhelpful Error Messages

New Experienced

Error messages are a common source of issues for novices in any programming language [11]. 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.[5] 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.

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.

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

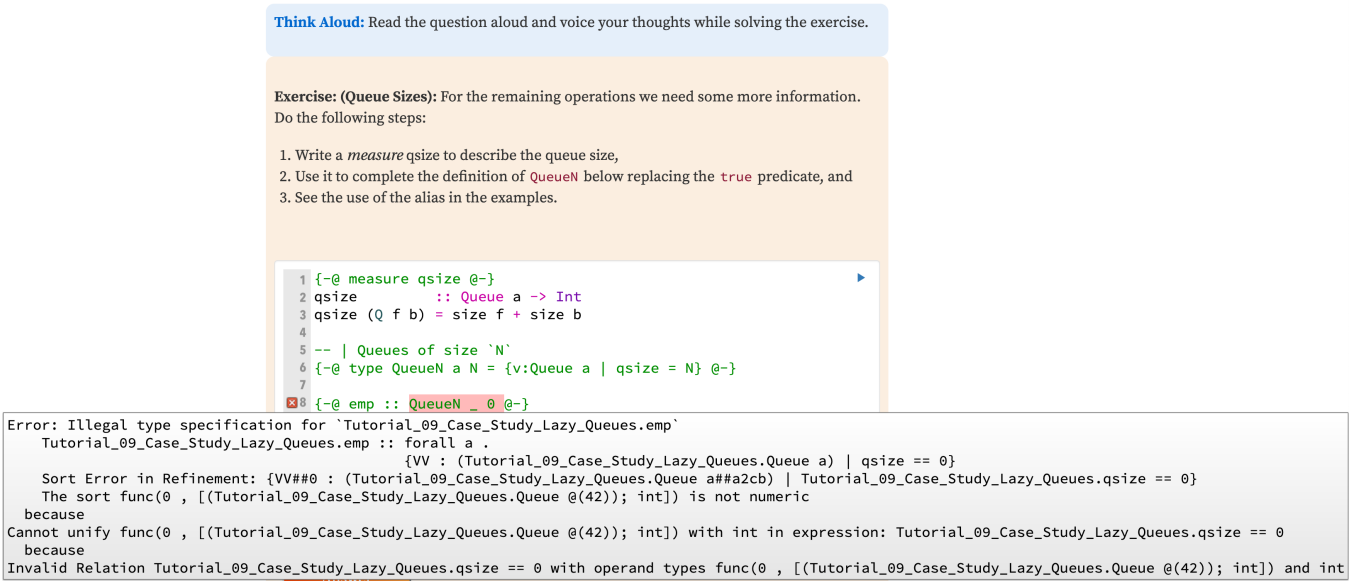


Figure 3.3: Error message produced when a participant did not apply `qsize` to the variable `v`, when completing the final exercise after the tutorial.

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 3.3 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.

Error Indirection and Code Mismatch

From the example in Figure 3.3, 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 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).

3.4.8 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."

3.4.9 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`[6](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.

3.4.10 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 faced several issues (FM1, FM2, FM7). Initially, there were two main options

[6] https://ucsd-progsys.github.io/liquidhaskell/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)*

3.4.11 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,[7] 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

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

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.

3.5 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 3.2, 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 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 3.4.1) can be hindered by the unclear divide between Haskell and LiquidHaskell (Section 3.4.2). 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 [64], 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 3.4.3). 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 [92] or Dafny [96], have similar features with pure methods [47] or functions,[8] 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

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

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 3.4.4). 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. [74], 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 3.4.7), limited IDE Support (Section 3.4.8), insufficient learning and reference resources (Section 3.4.9), and complex installation and setup (Section 3.4.10).

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 [101] but also for experts who are learning new features [140]. However, in systems with constraint solving the challenges are amplified as it is harder to identify and recover from errors [153]. Therefore, these messages should be adapted to characteristics such as having multiple locations as sources of the errors, exposing internal representation and reasoning about constraints. For example, Eremondi et al. [58] 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 [23] and Agda [20]. 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 [145], 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,[9] Frama-C [87]) 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. [104] 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

[9] https://andreacensi.github.io/contracts/

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 3.4.6) as well as limitations of automation and manual proof flexibility (Section 3.4.5) 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 [62]. 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. [119] use machine learning for premise selection in Lean or how Automated Program Repair selects ingredients [150].

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 3.4.5. 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,[10] but with additional 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 [55]. One possible mitigating strategy is mentioned in Explicit Refinement Types [73], 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

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

Automated Theorem Solvers or Interactive Theorem Solvers,[11] 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.

3.6 Related Work

Most studies comparing languages and their features analyze open-source repositories rather than conducting developer interviews or observations [28, 32, 114]. Studies with developers present numerous challenges for research teams [48], 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 [33, 113]. For example, Coblenz et al. [43] 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 [41] and Obsidian [42]. As another example, Lubin and Chasins [98] used grounded theory analysis to study how programmers write functional and statically typed code.

Within the specific context of our study, LiquidJava [68] 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 [13] applied the cognitive dimensions framework [77] 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. These results align with our findings, as our participants emphasized needing to understand which conditions are being checked

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

versus which are missing to prove the properties, along with requiring better documentation and learning resources.

Juhošová et al. [83] 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. [45] 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. [155] and Coblenz et al. [40] 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. [12]. 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 [18] 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 [67] 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 [72], 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.

3.7 Summary and Research Directions

This chapter 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.

For this thesis, these insights motivated our subsequent contributions in the process of adding Liquid Types to Java, as summarize in Table 3.3. For each chapter, we indicate which barriers are the *primary focus* of the work, which barriers were taken into account and we *contribute to* alleviating, and which barriers remain not addressed by any of the chapters, and are therefore still *open challenges*.

Beyond the barriers identified in this study, there are additional challenges drilling from the object-oriented nature of languages like Java that we will also take into account in the subsequent chapters. One example, is the presence of mutable state and aliasing, which complicates verification but is necessary for modelling many Java programs. Chapter 6 addresses this gap by extending LiquidJava with lightweight aliasing tracking.

Table 3.3: Mapping of usability barriers to thesis contributions. Barriers are grouped by the chapter that most directly targets them.

Barrier	**Relation**	**Approach**
Chapter 4: LiquidJava Design		
Unclear divide between base language and verification	◐	Designed with Java developer feedback; refinements as annotations keep syntax familiar
Confusing verification features	◐	Explicit annotations without lifting functions, though new features may introduce complexity
Chapter 5: Agentic Synthesis		
Unfamiliarity with proof engineering	◐	Reduces manual specification effort when documentation exists
Limitations of automation and manual proof flexibility	◐	Automates specification generation but does not eliminate manual effort for complex cases
Chapter 7: Error Diagnostics		
Unhelpful error messages	●	Counterexamples, hints, and simplified presentation
Limited IDE support	●	Syntax highlighting, hover information, diagnostic explorer
Complex installation and setup	●	Improved tooling and streamlined installation
Insufficient learning and reference resources	◐	Beginner-friendly tutorial
Open Challenges		
Scalability and solver limitations	○	Requires advances in SMT solver and its interaction with verification algorithms

● Primary focus ◐ Contributes to ○ Open challenge

4 Design of LiquidJava

This chapter presents the work on Usability-Oriented Liquid Types for Java [68], published at ICSE 2023, that provides the design of Liquid Types in Java for following developers' feedback and using typestate to model object protocols.

We investigate how Liquid Types can be integrated into Java by proposing a new design that aims to lower the barriers to entry and adapts to problems that Java developers commonly encounter at runtime. Following a participatory design methodology, we conducted a developer survey to design the syntax of LiquidJava, our prototype. To evaluate if the added effort to writing Liquid Types in Java would convince users to adopt them, we conducted a user study with 30 Java developers. The results show that LiquidJava helped users detect and fix more bugs and that Liquid Types are easy to interpret and learn with few resources. At the end of the study, all users reported interest in adopting LiquidJava for their projects.

4.1 Motivation

Type systems have been widely adopted in programming languages to detect errors at compile time. To encode domain-specific knowledge, however, we need more expressive type systems such as Refinement Types [82], as they extend the language with predicates that restrict the allowed values in variables and methods. For example, listing 4.1 shows a simple refinement on the variable `r`, restricting the allowed values to the range of 0 and 255. Liquid Types [125] represent the decidable subset of refinements that allow automatic verification by SMT solvers.

Refinement Types have been used to detect simple division by zero errors and out-of-bounds array access bugs [149], as well as more complex security issues [16] and protocol violations [26]. Despite these advantages, there are several usability barriers that developers

```
1 @Refinement("r >= 0 && r <= 255")
2 int r = 90;   // Correct
3 r = 200 + 60; /* Raises a Refinement Type Error:
4                  Type expected: (r >= 0 && r <= 255);
5                  Refinement found: (r == 200 + 60) */
```

Listing 4.1: Variable refinement and verification in LiquidJava.

face as we mentioned in Chapter 3.

Since previous works on Liquid Types did not include a human-focused design or evaluation, and are mostly targeted at functional programming languages, we set out to explore how Liquid Types can be designed for usability in Java, one of the most popular programming languages in the world [4].

In this work, we follow a user-oriented approach to design and evaluate Liquid Types for one of the most popular programming languages in the world, Java. Our approach is comprised of two parts: First, we design a Liquid Types extension for Java following a participatory design methodology, where users guide the decision on how to express Liquid Types. Then, we conduct a user study to evaluate if using this extension is more beneficial when compared to using plain Java. Therefore, our contributions are:

- The design of LiquidJava, an extension of Java that supports Liquid Types, based on user feedback, and subsequent prototype implementation.
- A user study that evaluates the usability of LiquidJava, namely how understandable the code with refinements is, and whether it is more useful for developers to detect and fix bugs, when compared with plain Java.

Data Availability

The data supporting this paper is available online at https://doi.org/10.5281/zenodo.7545674. The online package includes the study guides, the data gathered during their execution and a virtual machine to simulate the required environment.

4.2 LiquidJava Design

Designing interactive systems with the input of prospective users is a popular strategy in the field of HCI (Human-Computer Interaction) [52, 65]. Designers of new systems are encouraged to develop prototypes that aim to fulfill the users' needs and apply formative evaluations of the prototypes to include the users' feedback in their design solutions.

Programming languages are a form of human-computer interaction, and they should be designed to meet the developers' needs and expectations, having a clear focus on usability. While most of the effort in studying programming language interaction has been focused on learning (e.g., Hedy [79]), advances are being made in studying the interaction of experts. Coblenz et al. [43] discuss the difficulties of applying HCI methods to the design of languages and propose a process, PLIERS, for designing languages focused on the users. These principles were used in the development of two languages: Glacier [41], a language for immutability in Java, and Obsidian [42], a language to model blockchain protocols.

For LiquidJava, we started by defining usability requirements (Section 4.2.1). Then, we applied a formative study with users to drive the choice of the syntax based on our proposed alternatives (Section 4.2.2). Based on the results of the formative study, we added Liquid Types to Java, covering local variables, fields, method definitions, and method invocations.

4.2.1 Requirements

To promote the usability of Liquid Types in Java, we identified three requirements to guide the language design. These requirements were based on previous implementations of Refinement Types and popular characteristics of successful static verification techniques, such as the `@NonNull` annotation [61], as well as feedback from developers [53, 128]. The requirements are the following:

R.1 Refinements must be optional: Without refinements, a Java program must be successfully validated by the liquid type-checker, allowing specifications to be introduced after the implementation, including to pre-existing codebases. This grants the developer the possibility of incrementally building the program's specification.

R.2 Refinements need to be expressive: Refinements need to cover a wide range of specifications while being written with a syntax similar to Java to enable developers to write specifications without learning a completely new language.

R.3 Refinement type-checking should be decidable: The type-checking process should be decidable to prevent unnecessary overhead to the compilation process, provide interactive feedback to developers as they code, and reduce false positives — all concerns mentioned in previous static analysis tools [53, 128]. To this end, the refinements language is restricted to Liquid Types, which are verifiable by SMT solvers. Thus, the predicates are restricted to decidable logics using quantifier-free linear integer arithmetic with uninterpreted functions and accepting only SMT-decidable operations.

The design solution to fulfill the requirements can be split into two main topics: the design of the refinements language and how it is incorporated into Java programs; and the design of the verification system to validate these programs.

Refinements Design

For the design of the refinements, we first had to decide how to introduce the predicates in the Java source code while following the aforementioned requirements, specifically regarding the need for a flexible and understandable language. Moreover, in previous refined typed languages [36, 82], variable and method declarations have been the main target of refinement annotations. However, classes are also a core concept in Java and a desirable target for refinements.

Considering this, we decided to encode the refinements as Java Annotations in the source code since annotations are optional, support all necessary targets, and have been commonly used in Java since their introduction in JDK 1.5. This decision fulfills **R.1** since these annotations are always optional as the Java type checker does not verify them. Additionally, annotations are attached to target code elements including local variables, parameters, methods, fields, and classes, allowing us to introduce refinements to all the desired code elements considered in this study.

Using annotations to express restrictions on variables has become more popular over the years, with `@NonNull` and `@NonEmpty` being present in many Android and enterprise applications, as well as in verification tools for Java (e.g., Jakarta Bean Validation [108], Checker Framework [117]). The popularity of annotations gives us some confidence that

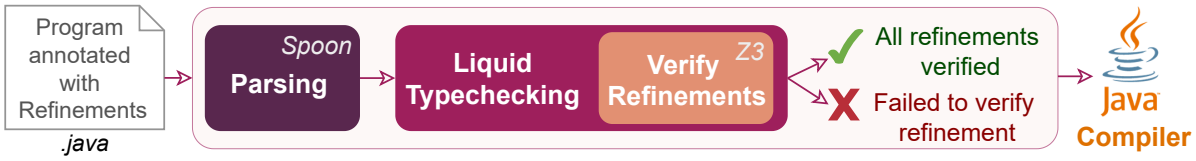


Figure 4.1: Pipeline of the LiquidJava system.

their usage within LiquidJava will not constitute a barrier to the system's adoption. Thus, a new `@Refinement` annotation was created to express refinements, and other annotations were also created as syntactic sugar (e.g., `@StateRefinement`).

However, annotations by themselves are not flexible or very expressive. Therefore, the refinements are written as strings inside the annotations and follow a custom language developed to be as similar to Java as possible, addressing **R.2** and reducing the obstacles in writing specifications. To improve the understandability of the language, we conducted an online survey to assess the best syntax for the features of the language while keeping it verifiable by SMT Solvers (respecting **R.3**). The syntax survey is detailed later in this paper in section 4.2.2.

System Design

To verify a Java program annotated with refinements, a LiquidJava verifier either proves that all refinements are respected throughout the program, or shows that there is a violation of the specification (design in fig. 4.1).

The verifier receives a Java program annotated with refinements as input and performs a static verification before the execution of the program. The first step of the system involves parsing the input to an abstract syntax tree (AST) used for the verification, using the Spoon [118] framework. Taking the AST representation of the program, the liquid type checker traverses the AST and checks all expressions against their expected type through subtyping relationships. These relationships are then discharged to an SMT Solver, which proves their satisfiability. The details of the verification process are out of the scope of this paper but are available in online resources. [1]

4.2.2 Syntax Survey

To design the refinement syntax, we decided to get feedback from Java developers instead of following a personal preference, which is the most common approach in designing new languages. Therefore, to assess the best syntax for the language of refinements, we created and shared an online survey with possible syntaxes for different LiquidJava features, some based on other implementations of Refinement Types and others based on the Java language. These features include type refinements of variables and methods and the use of predicate aliases and anonymous variables. To minimize the time needed to complete the survey and have more participation, we proposed two to three syntax options for each language feature

[1] https://catarinagamboa.github.io/papers/master_thesis.pdf

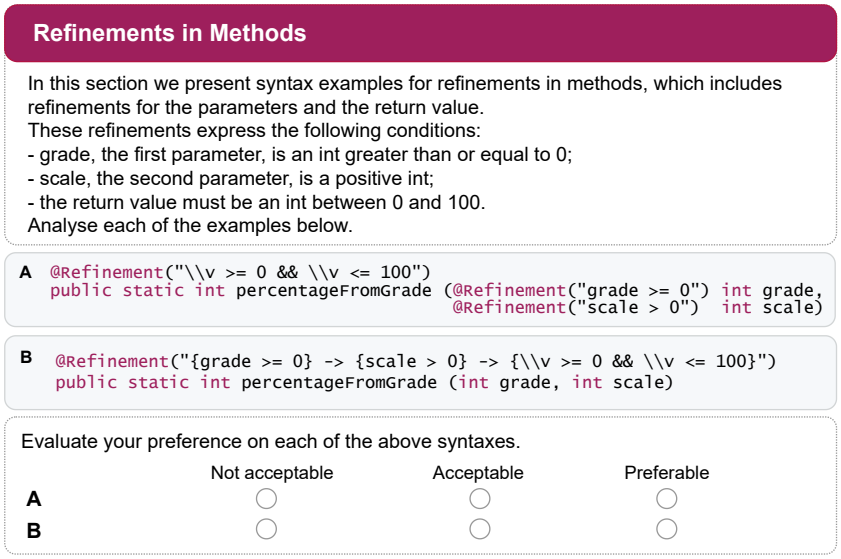


Figure 4.2: Question on the preferable syntax for refinements in parameters and return value of methods.

instead of asking the participants for their syntax proposal. Specifically, we asked the participants to evaluate each of the syntax options we designed with one of three preference levels: *Not Acceptable*, *Acceptable*, and *Preferable*.

The study was sent to academic students, researchers, and industry Java developers. As a result, we obtained 50 answers from participants acquainted with Java. The survey started with questioning the participants' background and briefly explaining the concept of Refinement Types before questioning the preferred syntax options for LiquidJava features. The background answers of participants show that more than half of the participants (a total of 56%) are *Not Familiar* at all with Refinement Types and that only 2% consider themselves *Very Familiar* with the concept. These percentages show that most developers are not familiar with Refinement Types, highlighting that they are not widely spread.

Figure 4.2 represents one of the survey questions, where participants could read the description of the refinements for each of the parameters and the return type of the method `percentageFromGrade` and analyze each of the proposed syntaxes to then evaluate with their preference. These syntax options follow two different designs. The first option attaches each refinement to the type of variable that it is refining. Therefore, the parameters have the refinements just before their basic type, and the return refinement is above the method before the return type. The latter option has a syntax inspired by the type signatures used in functional languages, such as Haskell. Therefore, the parameters and return refinements are written in the same line, split by the `->` symbol, with a specific order starting with the parameters and finishing with the method's return type.

The participants evaluated their preference for both proposed syntaxes, and fig. 4.3 shows the gathered results. The first option had more *Preferable* answers and fewer *Not Acceptable* answers, which made us choose the first syntax for this feature.

The remaining features had similar syntax questions and preference evaluation (available

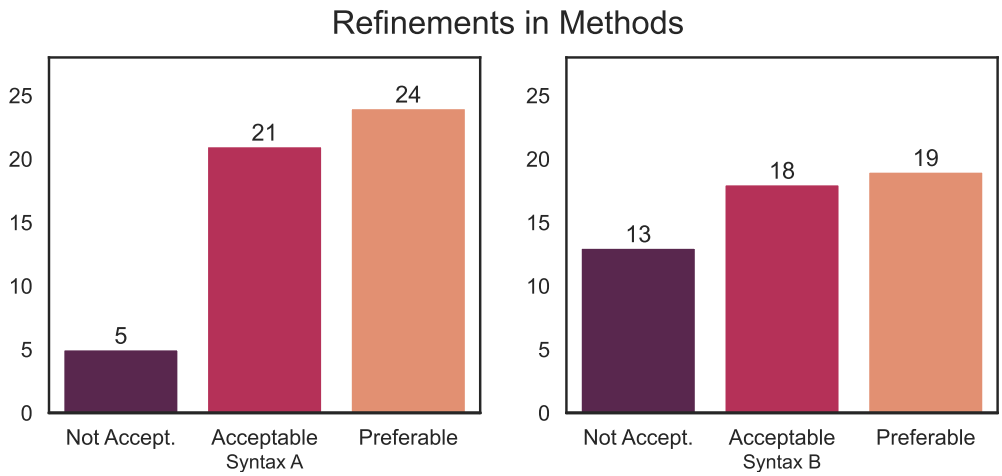


Figure 4.3: Preferences on the Syntax for Methods' Refinements.

in additional resources). We concluded that throughout the survey, participants preferred syntaxes with a flavor similar to Java, discarding syntaxes closer to previous implementations of Refinement Types that had a style closer to the syntax of functional languages.

4.2.3 LiquidJava Features

LiquidJava supports the refinement of variables and fields whose assignments must always respect the introduced predicates. Furthermore, it is possible to add refinements to return values and parameters of methods, verifying the return value against the expected type and verifying the method's invocations with the expected arguments.

Listing 4.2 depicts an example of these features. The `inRange` method receives two parameters, `a` and `b`, where the first parameter has no annotation, staying with the default refinement of `true`. In contrast, the second parameter has a refinement dependent on the first, informing that it should be greater than the value of the first parameter. Above the method's signature (line 1) is the return value refinement, which informs that the return value (represented by the `_`) is expected to be within the range of both parameters. All return values are also checked against the declared return refinement.

In line 5, a variable `x` is declared with a refinement that restricts its value to integers greater or equal to 10. In the following assignment, the method invocation must respect the parameters' refinements, and the invocation's return should respect the variable refinement. In this case, both verifications hold, validating the operation. However, the second invocation produces an error since the second argument is not greater than the first one, and an error message depicting that information is shown to the user.

In LiquidJava, refinements can also be used to model the state of class objects. Although previous extensions of Refinement Types did not focus on object modeling, classes are considered a fundamental programming element of the Java language [85] and, therefore, we model them using typestates [7].

Although classes themselves do not have a specific value that can be refined, they

```java
@Refinement ("a <= _ && _ <= b")
public static int inRange(int a, @Refinement("b > a") int b) {
    return a + 1;
}
@Refinement ("x >= 10") int x;
x = inRange(10, 20 - 5); // Correct invocation and assignment
inRange(10, 2); /* Refinement Type Error
                   Type expected: (b > a);
                   Refinement found: (b == 2) && (a == 10) */
```

Listing 4.2: Refinement annotation of a method and a variable and verification of related operations.

can have methods that produce changes to the internal state of the objects. Therefore, we can refine the object state by restricting which state the object has to be in when a method is invoked and what is the state of the object after the execution of the method. Both predicates are encoded in methods with the annotation `@StateRefinement(from ="predicate", to="predicate")`, where the `from` argument indicates the object state from which the method can be called, and the `to` argument contains the resultant object state.

A method can have different combinations of source and destination states. These combinations are encoded as multiple `@StateRefinement` annotations above the method. To model the class state, one can create ghost functions that represent class properties (e.g., the size of a list) or define a set of states that the class objects can have. The states and the ghost properties are invoked inside the predicates as functions that take the current object as an argument using the `this` keyword. Moreover, if it is necessary to refer to the object's previous state, it is possible to use the `old` keyword with the current object, resulting in the expression `old(this)`.

Using `@StateRefinement` allows users to define protocols that a class must follow by encoding a finite state machine within the refinements. Java classes usually define protocols that the client programs must follow. However, these protocols are primarily defined through informal documentation using natural language (e.g., Javadoc). Therefore, most protocols are not enforced during code development, leading to runtime exceptions. By encoding the protocol definitions in LiquidJava, developers get feedback on the correct use of classes before the program execution, allowing them to, for example, use external libraries with more confidence.

Listing 4.3 shows the annotation of the native library `java.net.Socket`.[2] The specification models the implicit state machine semantics (shown in fig. 4.4), documented only in natural language. The `@StateSet` annotation describes all possible states, and each method describes the states in which it is available (`from`) and the state in which the object is after the method execution (`to`). Using this specification, a program that invokes `sendUrgentData` without first invoking `connect` will not be accepted.

[2]https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html

```java
@ExternalRefinementsFor("java.net.Socket")
@StateSet({"unconnected", "bound", "connected", "closed"})
public interface SocketRefinements {
    @StateRefinement(to="unconnected(this)")
    public void Socket();

    @StateRefinement(from="unconnected(this)", to="bound(this)")
    public void bind(SocketAddress add);

    @StateRefinement(from="bound(this)", to="connected(this)")
    public void connect(SocketAddress add, int timeout);

    @StateRefinement(from="connected(this)")
    public void sendUrgentData(int n);

    @StateRefinement(from= "!closed(this)", to="closed(this)")
    public void close();
}
```

Listing 4.3: Socket class object state refinement.

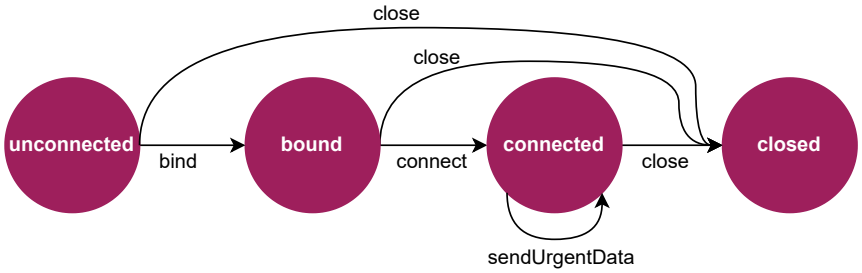


Figure 4.4: DFA representing the Socket class states and transitions.

4.2.4 IDE Integration

To enhance the usability of LiquidJava, we developed an IDE plugin to provide immediate liquid typechecking feedback, while developers are programming. It also provides localization of bugs by underlining the relevant incorrect code. Figure 4.5 shows how errors are reported. While we support it as a plugin for VSCode, it is available as a Language Server [105], which can be easily integrated in other IDEs.

4.3 User Study

Evaluating software engineering tools with user studies is not a recent practice, but it is still scarce. Mainly because most researchers find these experiments too difficult to design and conduct, have difficulties recruiting participants and believe that the results might

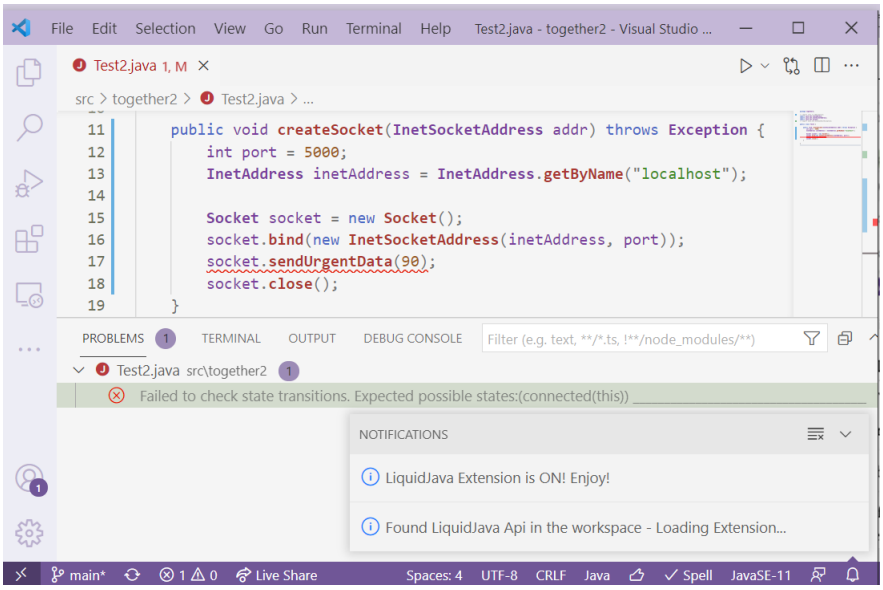


Figure 4.5: IDE Plugin reporting an error in the incorrect usage of the Socket class.

be inconclusive [86]. However, researchers who conducted user studies agree that these evaluations provide useful insights that outweigh the study costs while increasing the impact of the work [27].

In previous research, authors have employed user studies to understand how programmers write code [98, 138] and compare different programming language designs. For example, comparing the benefits of using static versus dynamic type systems for maintainability and undocumented software, [78, 102] analysing the usability and learnability of API aspects [56, 132], and advanced language features such as lambdas [141] and garbage collection [39]. In PLIERS, the authors also present strategies for conducting summative usability studies and apply them to two novel programming languages. These studies either apply usability studies or randomized control trials (RCTs) [43]. Usability studies usually have participants complete tasks where relevant data is retrieved, such as the time spent on the task, the correctness of the answers, and the errors made. In RCTs, the study configuration aims to compare two or more design options having one option as a control condition and assigning each participant a random design to fulfill the tasks.

To evaluate LiquidJava, we developed a user study that combines the two types of tests to answer the questions:

Q1 Are refinements easy to understand?

Q2 Is it easier and faster to find implementation errors using LiquidJava than with plain Java?

Q3 Is it hard to annotate a program with refinements?

Q4 Are developers open to using LiquidJava in their projects?

Given these research questions, we planned the study with tasks to assess the usability of LiquidJava, and compare its benefits against Java. This section depicts the study

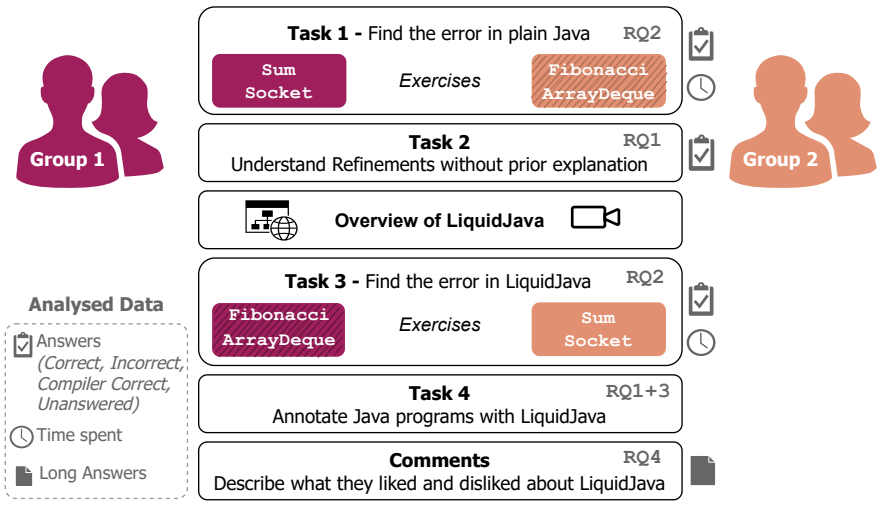


Figure 4.6: Configuration of the user study.

configuration (section 4.3.1), the participants' background (section 4.3.2), and the detailed tasks with results (section 4.3.3) and their discussion (section 4.3.4). In the end, the threats to the validity of the study are presented (section 4.3.5).

4.3.1 Study Configuration

The study was designed to introduce participants to LiquidJava and answer the aforementioned research questions.

The study was conducted individually with each participant, and everyone received the study plan with the described tasks and the answer sheet to input their answers. The participants were free to quit the study at any time and skip questions if they felt unable to answer them; however, they could not go back to previous study sections.

Figure 4.6 schematizes the configuration of the study. When starting, each participant was randomly assigned to one group. All participants within a group followed the same order of exercises. However, the two groups had different exercises for two tasks to compare the participants' performance using Java and LiquidJava. Each section of the study is described as follows:

- **Task 1: Find the error in plain Java** – Participants were asked to detect implementation errors in Java code and provide a possible correction. The implementation errors make the execution incorrect against the informal documentation presented in the method's Javadoc. The participants could use all the IDE environment resources, including reading documentation, changing the code, and even executing the code.
- **Task 2: Interpreting refinements without prior explanation** – Although participants must be familiar with Java, they are not required to be familiar with refinements. Thus, all participants had their first contact with LiquidJava in this task. Here, the participants had to interpret the refinements present in different sections of the code (variables, methods, and classes) and provide a correct and incorrect use of

the annotated code without ever being introduced to the language. This task aims to see if the participants find the refinements intuitive and easy to use without a prior explanation. Thus, this study section is related to **Q1**, and the data gathered captures the correctness of the answers the participants gave to the correct and incorrect uses of the refinements.

- **Overview of LiquidJava** – Participants were exposed to a 4-minute video and a webpage explaining the concepts of LiquidJava using the examples of the previous task. Both resources were available for the participants to use in the remainder of the study. These materials help make the study reproducible while reducing bias from the interviewers.
- **Task 3: Find the error with LiquidJava** – This was a repetition of Tasks 1, but using the LiquidJava plugin. There were two exercises to reduce the bias of already knowing the solution. Each half of the participants did one problem in Task 1, the control conditions, and the alternative problem in Task 3. Comparing the performance between the two tasks allows us to answer to **Q2**.
- **Task 4: Annotate Java programs with LiquidJava** – Participants were presented with three Java programs and were asked to annotate them with LiquidJava specifications. This task aims to answer **Q3** by asking participants how difficult it was to annotate variables, fields, methods, and classes. Because this was a relatively short task, its success also answers **Q1**.
- **Final Comments** – Finally, participants were asked about their overall opinion on using LiquidJava. They were also asked if they would like to use LiquidJava in their future projects to answer the last research question, **Q4**.

The study sessions were all conducted through the Zoom video platform, and participants used their own environments to complete the study tasks. To ensure that these environments fulfilled the requirements to complete all tasks, participants installed Visual Studio Code with JDK11 and the Language Support for Java(TM) by Red Hat extension. During the study, participants had access to the GitHub repository with the study files, a webpage with information on LiquidJava, and all IDE features provided by the plugins.

4.3.2 Background of Participants

We designed the study for 30 participants familiar with Java and recruited them through social media channels, such as Twitter and Instagram, and through direct contact via email.

Figure 4.7 shows that more than 90% of the participants considered themselves *Familiar* or *Very Familiar* with Java. The remaining participants, who considered themselves only *Vaguely Familiar* with Java, were only accepted into the study because they selected they were familiar with testing frameworks (e.g., JUnit). Of all the participants, 80% were *Vaguely Familiar* or *Not Familiar* with Refinement Types, which shows that despite their utility, Refinement Types are not widely known and used. The three participants familiar with LiquidJava had attended a talk about it but had not used it in any capacity. As for the participant's occupations, around 50% are university students, 26% work in the

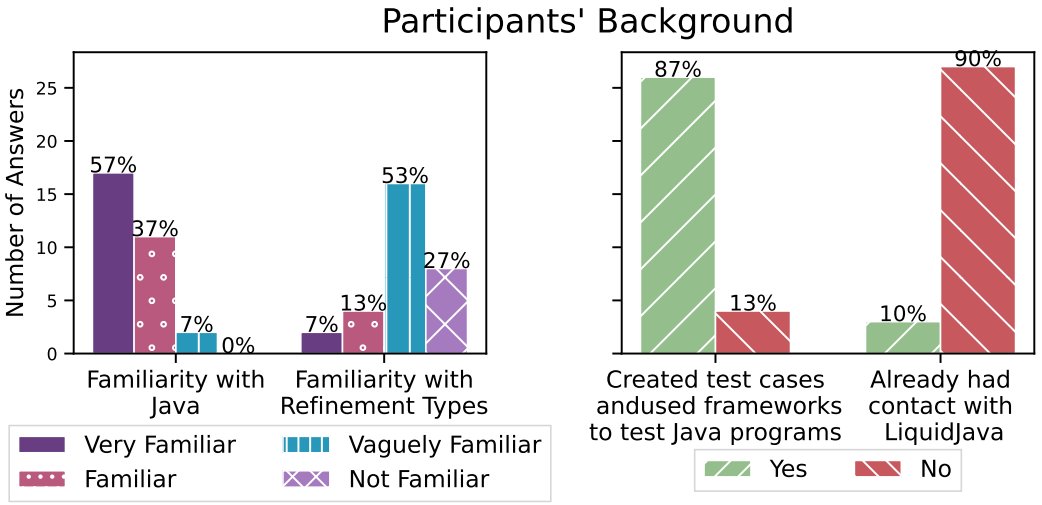


Figure 4.7: Answers to participants' background.

Table 4.1: Occupations of study participants.

Occupation/Job	# Participants
Business	8 (26.7%)
Faculty	6 (20.0%)
PhD Students	5 (16.7%)
Masters Students	7 (23.3%)
Final-Year Bachelor Students	4 (13.3%)

industry, and the remaining work as faculty in academia, as described in the table 4.1.

All participants completed the study, and the gathered data results are analyzed in the next section.

4.3.3 Exercises and Results

This section presents the exercises used in each of the tasks of the study and the results obtained.

Interpreting Refinements without prior explanation

Since 90% of the participants had no previous contact with LiquidJava, and more than 80% were not familiar with Refinement Types, we wanted to understand if, without a prior explanation, the added specifications were intuitive to use. Thus, the study included a task with refinements examples that the participants needed to interpret and use. Specifically, we presented three code snippets with LiquidJava refinements with an increasing difficulty level (as showed in listing 4.4) and asked the participants to implement a correct and incorrect usage for each of the represented features.

```java
// 1 - Variable Refinement
@Refinement("-25 <= x && x <= 45") int x;

// 2 - Function/Method Refinement
@Refinement("_ >= 0")
public static double function1(@Refinement("a >= 0") double a,
   @Refinement("b >= a") double b)
  { return (a + b)/2; }

// 3 - Class Protocol Refinement
@StateSet({"sX", "sY", "sZ"})
public class MyObj {
    @StateRefinement(to="sY(this)")
    public MyObj() {}

    @StateRefinement(from="sY(this)", to="sX(this)")
    public void select(int number) {}

    @StateRefinement(from="sX(this)", to="sZ(this)")
    public void pay(int account) {}

    @StateRefinement(from="sY(this)", to="sX(this)")
    @StateRefinement(from="sZ(this)", to="sX(this)")
    public void show() {}
}
```

Listing 4.4: Exercises to interpret refinements.

In the first exercise, participants had to assign a correct and incorrect value to the variable `x`, which was restricted by the limits of Earth's surface temperature. The second task had participants implement correct and incorrect invocations of `function1`, where the second parameter depends on the first. The last task presented a class protocol with three states and methods that model the object state. Here, the participants were asked to create a *MyObj* object and implement a correct and incorrect sequence of at least three invocations. The *MyObj* class aimed to represent a Vending Machine with the three states *sX, sY* and *sZ* as *Show Items, Item Selected* and *Paid*, respectively. The anonymization of the states and the class name were intentional to make the participants try to understand the refinements instead of calling the methods according to their mental idea of how a vending machine works.

Figure 4.8 shows the evaluation of the answers given by the participants. Each answer was classified as *Correct* if both the correct and incorrect usage of the specification were correct, *Incorrect* if at least one of the usages was incorrect, or *Unanswered* if the placeholder for answering was left blank. In the variable assignment, 86.7% of the participants answered correctly. The remaining participants understood the error when the examples were explained and claimed that the error was a pure distraction and misread the logical

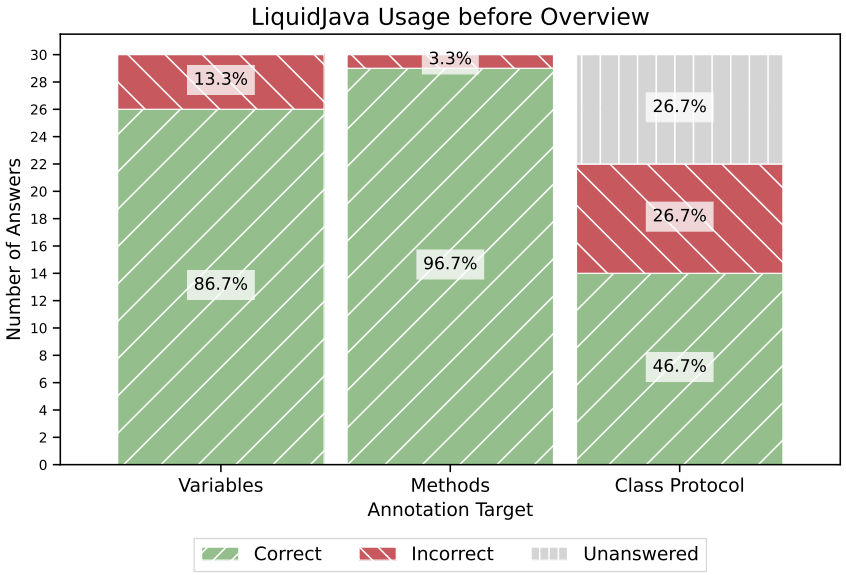


Figure 4.8: Answers on interpreting LiquidJava refinements.

operators. The invocation of the annotated method had only one incorrect answer (3.3%). For the sequential methods' invocations, that depended on the class protocol described using the `@StateRefinements`, 46.7% of the answers were correct, and the remaining amount was split into incorrect and blank answers, hinting that this example is less intuitive and harder to understand without a prior explanation, but still comprehensible by almost half of the participants.

Overall, refinement annotations in variables and methods seem intuitive and easy to understand. However, the annotation of classes and their methods with protocols is less intuitive. In half of the cases, the participants would need a previous explanation to use these annotations correctly.

Using LiquidJava to Detect Bugs

Participants had similar exercises in the first and third tasks to compare the effort of detecting and fixing bugs while using Java and LiquidJava. The four exercises used in the study are versions of *Fibonacci*, a *Sum* of all numbers until a given input, and invocations to the *ArrayDeque* and *Socket* libraries. The exercises are simple, with a maximum of 15 lines of meaningful code (e.g., disregarding empty lines), to ensure that the participants can reason about the programs with their Java background and the introduced information about Liquid Types. Since all participants are already used to working in Java, they all start with the plain Java exercises, and after a brief introduction to LiquidJava, they move to detect bugs with the custom plugin activated.

Each exercise has two versions, a plain Java and a LiquidJava version, with the same implementation errors that allow us to compare the number of participants that found and fixed the bug and the time taken by the participants to complete the exercises. A group of participants started with the exercises *Sum and Socket* while the second group started with

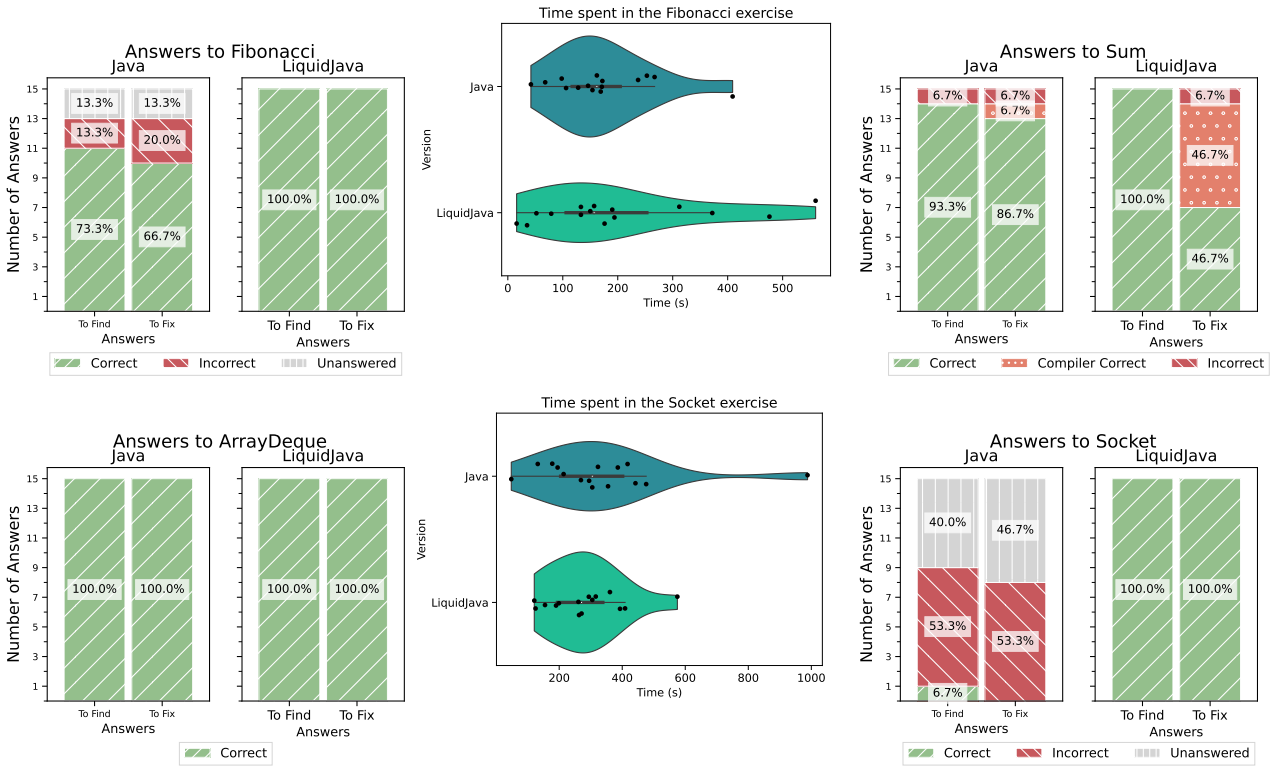


Figure 4.9: Results on finding and fixing errors using Java and LiquidJava.

the plain Java versions of the exercises *Fibonacci and ArrayDeque*. When moving to detect the errors using LiquidJava, the first group had the exercises *Fibonacci and ArrayDeque* whereas the second group got the exercises *Sum and Socket*. Therefore, one participant never used the same exercise in both tasks, avoiding tainting the second task with previous knowledge of the solution and allowing us to obtain plain Java baselines for every exercise. With this split, the maximum number of answers to each version is 15 since only half the participants viewed each exercise version.

For each exercise, we gathered the time spent and the written answers for the incorrect line(s) and the proposed fixes. The answers were then evaluated into one of four possible categories: *Correct*, *Incorrect*, *Unanswered*, and *Compiler Correct*. The last category represents the answers that silenced the compiler error despite not being utterly correct according to what the exercise asked.

Exercise	Average Time Java	Average Time LiquidJava
Fibonacci	2 mins 52 secs	3 mins 22 secs
Sum	4 mins 30 secs	3 mins 32 secs
ArrayDeque	3 mins 41 secs	2 mins 56 secs
Socket	5 mins 35 secs	4 mins 42 secs

Table 4.2: Average times to complete the exercises.

The results of finding and fixing bugs in the four exercises are displayed in fig. 4.9. Moreover, the figure includes two plots on the distribution of time taken by participants in each exercise, and the average times for each exercise are in table 4.2. The code given to participants and their answers is available online at https://doi.org/10.5281/zenodo.7545674.

All exercises have in common that 100% of participants found the error location while using LiquidJava, which was expected since the plugin underlines the error found. However, the plugin does not inform about a possible fix, and in most cases, participants outperform the LiquidJava version for fixing the error. Additionally, in all but one exercise (*Fibonacci*), participants were faster on average in the LiquidJava versions.

Both ***Fibonacci*** and ***Sum*** exercises are implemented using recursion with an error in the base case. Inspecting each exercise individually, we can see that in the ***Fibonacci*** exercise, all participants fixed the error using LiquidJava, while only 66.7% fixed the error in the plain Java version. Regarding the time spent on both versions, on average, participants were faster in the plain Java exercise when compared to the LiquidJava. This result suggests that participants might already be used to Fibonacci's plain implementation, given the algorithm's popularity in introductory programming classes. When the new refinements were added, participants spent more time understanding the different code sections.

For the ***Sum*** exercise, 7 participants correctly fixed the program, and the other 7 provided a *Compiler Correct* answer. These last answers silenced the compiler error since they complied with the liquid specification but did not comply with the informal specification of the method. After reviewing these answers, it is possible to see that 6 out of the 7 participants fixed the error with the same code as they used in the first exercise in plain Java (*Fibonacci*), and while it was the correct fix before, it was not the required fixed in this exercise. This line of thought might indicate that participants were biased by the previous sections of the study and opted for the same answer as they used in the beginning.

In the ***ArrayDeque*** exercise, all participants fixed the error in both versions by fixing the order of invocations of popular methods to add, remove, and retrieve elements from an ArrayDeque object depending on its number of elements.

The ***Socket*** exercise is the one with the largest difference in the number of correct answers while using Java and LiquidJava. In the Java version, none of the participants could fix the error, giving a wrong fix or leaving the answer blank. However, while using LiquidJava, every participant understood that there was a missing invocation in the order of methods to make the invocations valid. In both cases, the participants had access to the informal documentation of the library through the Java plugin active in the IDE. The time spent in this exercise shows that participants were faster by 52 seconds in LiquidJava, on average, with a higher rate of correct answers.

Adding LiquidJava Annotations

In Task 4, participants were asked to add LiquidJava annotations to the implemented code according to the informal documentation written in the program as comments. In this step, participants could use the website and the video to help write the refinements.

Participants had to annotate programs with increasing order of difficulty. The first program relied only on the annotation of a variable with its bounds. The second program

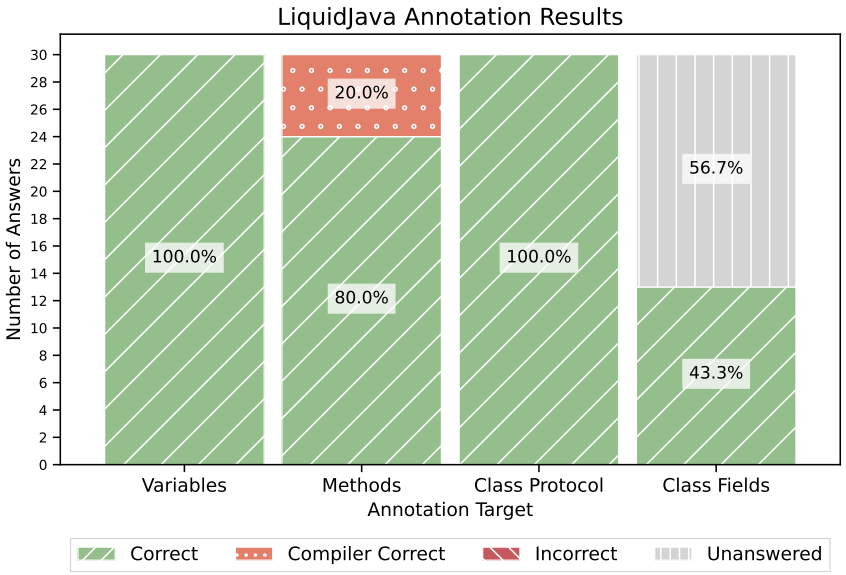


Figure 4.10: Results of the annotations with LiquidJava.

expected the annotation of a method by specifying the parameters and return refinements. Finally, the third program required the annotation of a class protocol and the class fields. For each program, we presented an example of a correct usage of the refinement and another example of its incorrect usage to help the developers test their refinements.

The participants shared their proposals for the annotation of each exercise, and we evaluated them with the four categories used in the previous section, of *Correct*, *Incorrect*, *Unanswered* and *Compiler Correct*. The results of the annotations are in fig. 4.10 and are analyzed along with each exercise below.

The first and more straightforward exercise just included a variable named `currentMonth` that should be restricted with the lower and upper bound of month values, resulting in a code similar to `@Refinement("currentMonth >= 1 && currentMonth <= 12")`. In the first exercise, all the participants used the website as a resource to look for the right syntax, and 100% of them annotated the variable correctly. The second exercise presented the method `inRange` where participants should add a refinement to the second parameter, changing the signature of the method to `public static int inRange(int a, @Refinement("b > a")int b)`, and refine the return type of the method, adding the refinement `@Refinement("_ >= a && _ <= b")` above the method's signature.

24 out of 30 participants were able to add the expected annotations leading to 80% of `Correct` answers. However, 20% only added the annotations to the parameter, silencing the example error but not completing the exercise in its totality, which led to the *Compiler Correct* answers.

For the last exercise, we asked participants to "Annotate the class TrafficLight, that uses RGB values (between 0 and 255) to define the color of the light, and follows the protocol defined by the image [in fig. 4.11]", as announced in the study guide. The evaluation of this exercise was split into two: the addition of the refinements to model the class protocol

```
1  public class TrafficLight {
2    private int r;
3    private int g;
4    private int b;
5
6    public TrafficLight() { r = 76; g = 187; b = 23; }
7    public void transitionToGreen() { r = 76; g = 187; b = 23; }
8    public void transitionToAmber() { r = 255; g = 120; b = 0; }
9    public void transitionToRed() { r = 230; g = 0; b = -1; }
10 }
11
12 //Correct Test - different file
13 TrafficLight tl = new TrafficLight();
14 tl.transitionToAmber(); tl.transitionToRed();
15 tl.transitionToGreen(); tl.transitionToAmber();
16
17 //Incorrect Test - different file
18 TrafficLight tl = new TrafficLight();
19 tl.transitionToAmber();
20 tl.transitionToRed();
21 tl.transitionToAmber();
22 tl.transitionToGreen();
```

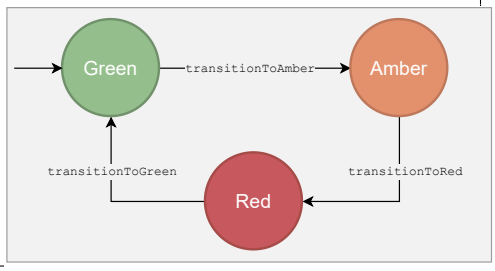


Figure 4.11: Program to annotate with the protocol showed in the diagram, and with the limit ranges on fields.

and the specification on the class private fields.

All participants correctly modeled the class by declaring the starting states and the state transitions of each method. This percentage constitutes a significant increase in the understanding of class protocols compared to the first time participants tried to understand the protocol in Task 1, where half of the participants could not interpret the meaning of the annotations.

However, only 43.3% of participants annotated the class fields. The remaining ones did not add any refinement to fields, leaving an incorrect assignment inside a method's body. There might be several reasons for this occurrence. One might be that they misinterpreted the exercise, not realizing the need for these annotations. Another possible explanation is that participants did not consider it important to add these annotations to the code.

After being introduced to the annotations, the participants had to evaluate the ease of adding annotations from *0 - Very Difficult* to *5 - Very Easy*. 60% of the participants considered that adding the annotations was *Very Easy*, while the remaining 40% considered the task *Easy*, leading us to conclude that refinements are simple to add to implemented code.

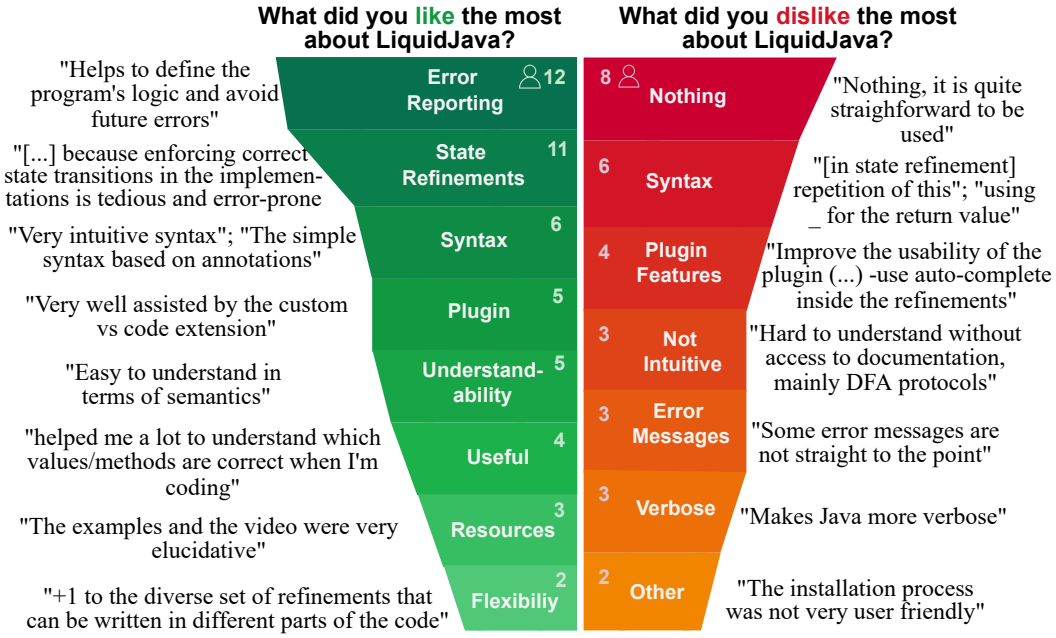


Figure 4.12: Comments on what participants liked and disliked.

Final Comments

At the end of the tasks, we asked the participants about the overall experience of using LiquidJava using three questions:

- What did you enjoy the most while using LiquidJava?
- What did you dislike the most while using LiquidJava?
- Would you use LiquidJava in your projects?

Figure 4.12 summarizes the answers to the first two questions with the codes obtained through a qualitative coding [129] approach and quotes from participants. We used inductive coding to create the codes, which were then used to review all passages and to identify the main topics of each answer, leading to a cohesive and systematic view of the results. We found that participants mostly appreciated the error reporting, state refinements to model objects, and the intuitive and non-intrusive syntax. As for the disliked topics, only 26 participants answered the question, from which eight explicitly mentioned there was nothing they did not like. The remaining negative aspects are related to some syntax options and plugin features to improve.

The last question of the study asked if participants would use LiquidJava in their projects, to which **all the participants answered affirmatively**. However, in the final suggestions, one participant declared that they would only use it in critical parts of the project, and two other participants stated that they are not currently using Java in any project but would like to have similar verifications in other programming languages.

4.3.4 Study Conclusions

The major takeaways of the study can be summarized in the following points.

- **Interpretation of refinements (Q1)** – Refinements in variables and methods are easy to understand without a prior explanation, and even though the features to model classes are not very intuitive at first, they are easy to understand with few resources.
- **Detecting and fixing implementation errors in Java and in LiquidJava (Q2)** – From Task 1 (section 4.3.1) and 3 (section 4.3.1), it is possible to assess that LiquidJava helped developers find the error present in the code. For fixing the bugs, LiquidJava helped in all but one case since developers focused on silencing compiler errors disregarding the reasoning behind the changes applied. As for the time taken in each exercise, participants generally finished the LiquidJava exercises faster.
- **Best result for detecting and fixing error in LiquidJava** – From all exercises, the one with the most improvements while using LiquidJava was the `Socket` client, where no participant could fix the error in plain Java. However, all participants were able to detect and fix the error using LiquidJava. This result might show that LiquidJava is more useful when applied to lesser-known classes and protocols than to mainstream classes or simple code.
- **Annotate Programs (Q3)** – All participants were able to add refinements to variables and to model class protocols. 80% were able to add refinements to methods correctly (the remaining silenced the errors), and 43% were able to introduce refinements in class fields (the remaining participants left the answer blank). Participants also classified the annotation process as *Easy* or *Very Easy*. Thus, we can conclude that refinements are easy to add to the code to model the desired behavior of programs.
- **Compiler Correct answers** – Having partial specifications on the code led to some participants changing the code to respect the specification, disregarding the program's intent, silencing the compiler error, and passing the verification without fully fixing the program. For example, in the *Sum* exercise and the annotation of methods, participants gave answers that were correct according to the refinements but incorrect according to the informal documentation, producing the *Compiler Correct* category of classification. This result might show that partial specifications can mislead developers if they do not capture the specification's meaning and the program's expected behavior.
- **Would participants use LiquidJava (Q4)**– Achieving the desired number of participants in the study shows that developers are open to participating in studies to discover new approaches to improve their code quality. Moreover, the affirmative answers on developers' willingness to use LiquidJava give us confidence that participants are open to using this approach. Therefore, we are confident that participants find LiquidJava accessible for its gains and are ready to use new useful verification tools.

4.3.5 Threats to Validity

This study shares threats to validity with other empirical studies (e.g., Glacier [41]). The first threat is the limited number of participants and the fact that they may not represent the population of Java developers. Their occupation in table 4.1 identifies the population for which this study could be generalizable. The exercises used for the tasks may also not be demonstrative of real-world tasks since they are small and simple to allow the sessions to be under 1h30min. However, these tasks were designed based on problems that also occur in larger projects. The study sessions themselves may not represent the development environment that developers are used to, and they might have less interest in fulfilling all tasks correctly. However, the environment was the same for Java and LiquidJava, and while VSCode is not on par with other IDEs, it has the necessary features for the small tasks presented.

There is also a potential learning effect between Tasks 1 and 3. This threat was addressed by using different problems, as described in the previous section.

4.4 Key Takeaways and Research Directions

Our participatory design and evaluation of LiquidJava demonstrated that refinements can be added to a mainstream, imperative, and object-oriented language. The formative study identified that Java developers prefer having refinements closer to the code units they refer to (e.g., annotations near method parameters), departing from approaches like JML [92]. These design decisions contribute to addressing two barriers identified in our broader study on Liquid Types adoption (Chapter 3): the *unclear divide between base language and verification*, which we mitigate by keeping refinements as familiar Java annotations; and *confusing verification features*, which we address through explicit annotations without implicit lifting of functions into the specification logic.

In the summative study, we found that refinements on variables and methods are intuitive (86% correct without prior explanation), while object state refinements proved harder to interpret without context (46% correct on first exposure). However, after a 4-minute video, 100% of participants correctly introduced protocol specifications. More importantly, LiquidJava helped participants detect and fix more bugs than plain Java—particularly for the Socket client exercise, where no participant fixed the error in plain Java, but all succeeded with LiquidJava. Participants valued that LiquidJava "helps to define the program's logic and avoid future errors" and that "enforcing correct state transitions in the implementations is tedious and error-prone" without such tooling.

The study also revealed areas for improvement: error messages that are "not straight to the point" and state refinements are "hard to understand without access to documentation." These findings motivate the subsequent contributions:

Automated Specification Synthesis. While state refinements ranked as the second most appreciated feature (11 participants), they remain difficult to interpret without documentation. This tension between developers valuing object protocols but struggling

with their complexity reveals an opportunity for automation. Chapter 5 presents an agentic approach that automatically generates object state protocols from class documentation using LLM-based techniques, reducing both the interpretation burden and the specification effort.

Improved Error Reporting. Although error reporting was the most appreciated feature (12 participants), participants noted that "some error messages are not straight to the point." This aligns with the *unhelpful error messages* barrier from our usability study (Chapter 3). Chapter 7 addresses this by introducing improved diagnostic presentation, expression simplification, and interactive debugging tools that bridge the gap between the verifier's reasoning and actionable feedback.

Aliasing Tracking. All participants expressed interest in using LiquidJava in their projects. However, our prototype does not yet support aliasing of mutable objects, which is a pervasive pattern in real-world Java programs. To enable broader adoption, Chapter 6 extends the type system with lightweight aliasing tracking that enforces unique references in the heap while precisely tracking aliasing in the stack, requiring minimal additional annotations.

5 Agentic Synthesis of LiquidJava Annotations

This chapter presents ongoing work on automated synthesis of LiquidJava annotations using LLM-based agentic techniques. Building on the usability barriers identified in Chapter 3 and the developer-centric design of LiquidJava presented in Chapter 4, we explore how automation can reduce the specification burden that hinders Liquid Types adoption. Specifically, we focus on generating object state protocols directly from class documentation, given that it was one of the most valued yet challenging features of LiquidJava. The work described in this chapter is currently in progress. We present our motivation grounded in the findings from our developer studies, describe the proposed agentic workflow, and outline the expected contributions.

5.1 Motivation

Writing formal specifications remains a significant barrier to adopting verification tools in practice. Our study of LiquidHaskell users (Chapter 3) revealed that developers struggle not because they lack motivation, given that they see clear value in verification, but because translating their understanding of code behavior into formal properties demands specialized expertise. Participants reported difficulty identifying the right pre- and post-conditions, structuring proofs, and reasoning about how the SMT solver interprets their specifications. This unfamiliarity with proof engineering creates a fundamental gap between developers' intuitions about correctness and how to encode the formal specifications to be verified.

Object state protocols exemplify this tension particularly well. In our evaluation of LiquidJava (Chapter 4), state refinements ranked as the second most appreciated feature among participants, with 11 developers highlighting their value for catching API misuse bugs. However, participants also reported that these specifications were difficult to interpret without documentation, and complex to write correctly. For example, developers understand that you cannot send data from a closed `Socket`, and this knowledge is often encoded informally in Javadoc comments, but it's more difficult to recognize it as a formal constraint and to know how to write it formally in LiquidJava. This disconnect between conceptual understanding and formal specification represents an opportunity that automation can address.

Recent advances in Large Language Models (LLMs) offer a promising path forward. LLMs excel at interpreting natural language documentation and have shown success in

generating function-level specifications such as pre- and post-conditions [57, 124]. However, object state protocols present unique challenges that existing approaches do not address. Unlike local method properties, protocols encode valid interaction sequences across an object's lifetime, as constraints that span multiple methods and evolve through state transitions. A `Socket` protocol must capture not just that `connect()` requires certain arguments, but that it must precede `sendUrgentData()` and that `close()` invalidates all subsequent operations. Generating these cross-cutting specifications requires reasoning about the class as a whole rather than individual methods in isolation.

We propose an agentic workflow that bridges this gap by transforming existing class documentation into verifiable LiquidJava specifications. Our key idea is that documentation already encodes the protocols that could be verified, particularly in the Java standard library, whose Javadoc forms a rich but often verbose source of usage directives and behavioral information [50], which we aim to extract and formalize. By generating a Deterministic Finite Automaton (DFA) as an intermediate representation, we structure the protocol extraction process and enable more reliable synthesis of state transitions. This approach directly addresses the barriers our studies identified as it reduces the proof engineering burden by automating specification generation, and it leverages documentation that developers already produce and understand. Rather than requiring developers to learn formal specification languages, we work from the documentation is already present generating the formal annotations on their behalf.

5.2 Related Work

Early approaches to automated specification synthesis relied on dynamic analysis or template-based generation. Daikon [60] dynamically infers likely invariants by instrumenting programs and observing variable values across test executions. While effective for discovering simple properties, it requires extensive test suites and cannot capture properties that tests do not exercise. Houdini [63] takes a static approach, using an iterative algorithm to generate candidate annotations from heuristic templates. However, it remains fundamentally limited by predefined patterns that support only simple logical expressions. Neither approach handles the cross-method dependencies that characterize object state protocols.

Recent work has leveraged Large Language Models (LLMs) to generate formal specifications from code and documentation, moving beyond the limitations of template-based methods. Endres et al. [57] demonstrate that LLMs can transform natural language documentation into formal postconditions, successfully catching real-world bugs. Richter and Wehrheim [124] extend this approach to generate both preconditions and postconditions, addressing the limitation of ignoring input assumptions. Beyond single-pass generation, researchers have explored iterative refinement strategies that leverage verification feedback. Wen et al. [146] combine LLMs with static analysis and program verification, iteratively refining specifications for C programs until they pass verification. Similarly, Ma et al. [100] introduce a two-phase approach for Java that combines conversational LLM generation with mutation-based refinement, achieving verifiable specifications for 72.5% of programs. These approaches generate function-level specifications effectively, but they do not address the

challenge of encoding valid method sequences across an object's lifetime. Closer to our focus on class-level properties, Sun et al. [133] propose ClassInvGen, which co-generates class invariants and test inputs for C++ classes. However, class invariants capture properties that hold at stable points rather than the state transitions that define object protocols. Related work on state machine extraction [136, 152] focuses on inferring protocols from implementations, but these approaches extract behavioral models rather than generating verifiable specifications.

LLMs have also been applied to proof generation in formal verification. Carrott et al. [29] present CoqPilot, a VS Code extension that automates Coq proof generation by combining LLMs with traditional tactics. Reducing the cost of these approaches, Chakraborty et al. [31] investigate fine-tuning smaller models on F* programs, demonstrating that fine-tuned models can match larger models like GPT-4 at lower cost. Closer to our domain, Li et al. [97] introduce a neurosymbolic approach specifically targeting Liquid Haskell, automatically generating refinement type annotations. While these approaches focus on proof completion or annotation inference, our work targets the upstream problem of synthesizing protocol specifications from documentation.

The effectiveness of multi-agent workflows for verification tasks motivates our architectural decisions. Mukherjee and Delaware [112] present AutoVerus, an agentic framework that orchestrates multiple specialized LLM agents for automated proof generation in Verus. Their work demonstrates that decomposing verification tasks across specialized agents outperforms single-model approaches. We adopt a similar philosophy, using dedicated agents for DFA generation, test synthesis, and refinement annotation—each with focused context and iterative feedback loops.

Although prior work addresses LLM-based specification generation, class-level properties, and agentic verification workflows individually, no existing approach combines these aspects to generate object state protocols. Our work fills this gap by transforming class documentation into verifiable LiquidJava specifications that encode both state transitions and method-level constraints.

5.3 Approach

Figure 5.1 presents our agentic workflow for transforming Java class documentation into verifiable specifications of constraints on both the class protocol and method arguments. The workflow employs three main agents that: 1) generate a state machine representing the class typestate; 2) synthesize passing and failing tests for the object according to the typestate; and 3) generate typestate and refinements as constraints over function arguments in LiquidJava. These agents produce a test suite and a specification-annotated class that enable detection of errors in the test suite.

This workflow draws on techniques shown effective in prior work. We integrate verification tools in-the-loop to leverage error message feedback for guided generation [100, 146]. We provide task-relevant context to each agent [31, 135], such as DFA construction rules and LiquidJava syntax documentation. Additionally, we also decompose the workflow into specialized agents that focus on distinct subtasks, following the philosophy that mimicking

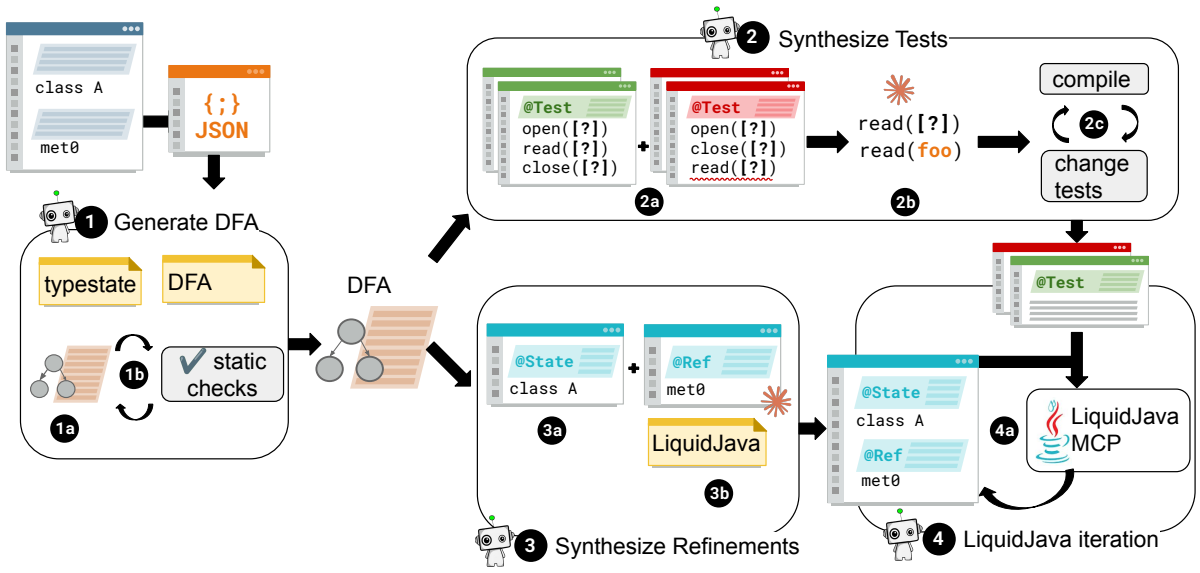


Figure 5.1: Agentic workflow to generate refinements. The approach starts with class documentation transformed into JSON format. Step 1 generates a DFA representing the class: the agent interprets documentation into typestate and applies DFA construction rules for structured output (1a), then validates the state machine iteratively until it passes all checks (1b). The DFA is then used for two parallel synthesis tasks: tests (2) and refinements (3). In step 2, we statically enumerate passing and failing test templates (2a), complete the templates (2b), and iterate until they compile as valid Java code (2c), producing a test suite. In step 3, we statically generate state refinements from the DFA (3a), use the LLM to generate parameter and return type refinements (3b), and validate them using the LiquidJava engine (3c), outputting annotated classes. Finally, we use the generated test cases to verify that the LiquidJava annotations detect all errors in the test suite.

expert behavior improves generation quality [112].

To illustrate each step of the workflow, we use the `java.net.Socket`[1] class as a running example throughout this section. This class appeared in our previous evaluation of LiquidJava [68], where participants identified typestate and refinement annotations as particularly helpful for finding bugs in client code.

5.3.1 DFA generation

The first agent receives the class documentation as structured JSON containing the class description, method signatures, and field descriptions. From this information, it generates a state machine that captures the class protocol and outputs the result as a Mermaid diagram[2]. To guide this generation, we provide the agent with documentation explaining typestate and its usage. This document draws on prior work on typestate [14], including a summary of the concept, steps for analyzing class protocols, and common protocol categories identified in empirical studies. We also include a classic example of a `File` class with `open` and `closed` states, illustrating both valid and invalid method sequences. An additional document describes DFA construction rules and the expected output format.

After generating an initial DFA, the agent validates it using a custom MCP tool that performs a series of well-formedness checks. These checks verify that all public methods and constructors are present with correct formatting, all states are reachable and have outgoing transitions, and the automaton is deterministic. If any check fails, the agent iterates on the DFA using the tool's feedback until it produces a version that passes all validation criteria.

Figure 5.2 shows a simplified version of the DFA generated by the agent for the `Socket` class. We retain the expected arguments for all constructors and methods to differentiate transitions that depend on parameter values. For example, initializing the object with no parameters (`Socket()`) transitions it to an *Unconnected* state, whereas providing the address and port (`Socket(InetAddress a, int port)`) transitions it directly to *Connected*. Some methods can be called from multiple states: `connect` is available in both *Bound* and *Unconnected*, while `close` can be called from any state. The protocol also includes self-loops, such as `sendUrgentData` in the *Connected* state, and methods like `toString` that are available in all states without changing them, but we omit these from the diagram for clarity.

With the generated DFA, we can proceed to the test synthesis and refinement phases.

5.3.2 Synthesize tests

Using the DFA, we generate test cases that exercise the class protocol by testing object creation and sequences of method calls. We first statically generate test templates for both valid and invalid usage patterns (2a). This generation explores paths through the DFA while applying constraints and heuristics to create a diverse test suite. Specifically, we employ a depth-first search with limits on maximum depth, consecutive method calls, and

[1] https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html
[2] https://mermaid.js.org/intro/

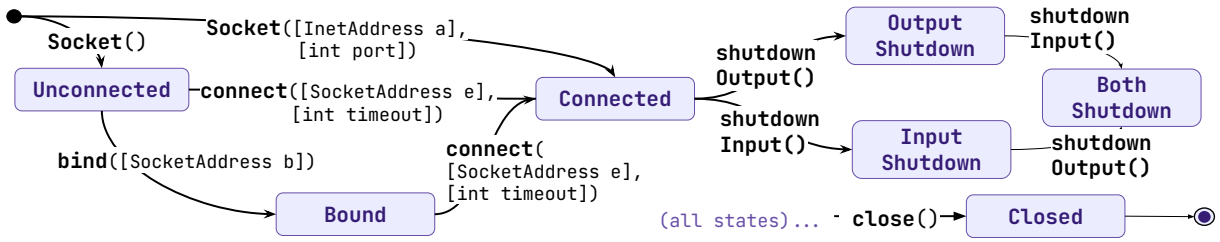


Figure 5.2: Simplified version of the DFA generated by Agent 1 for the `Socket` class. There are seven states present, and methods that change the object state.

self-loops, along with a minimum path length to ensure sufficiently complex examples. From this initial set of paths, we remove duplicates and rank the remaining tests to maximize coverage of state changes and distinct states visited.

At this stage, the tests remain templates as arguments and imports are not yet filled in. To complete them, the agent reasons about the expected types of each method call and generates appropriate argument values (2b). The agent then compiles the tests and iteratively refines them until compilation succeeds, producing a complete test suite (2c).

These tests serve as concrete usage patterns to validate LiquidJava's static type checking. They must be syntactically correct and typecheck according to Java's basic type system; however, we do not require them to be executable since runtime behavior falls outside the scope of the intended verification.

5.3.3 Refinement Synthesis

The first step in generating LiquidJava annotations is to produce the typestate refinements statically from the Mermaid DFA. These annotations describe valid state transitions, ensuring that methods are called in the correct sequence. Since typestate refinements directly mirror the structure of the state machine, we created a custom MCP tool to generate them automatically. The resulting class contains a skeleton with all state transitions but does not yet include refinements for method arguments.

In the next step, the agent annotates method arguments with logical constraints and adds any necessary imports to ensure compilation. To generate these refinements, the agent can consult the JSON documentation along with a guide describing how liquid types can constrain inputs and outputs. It produces candidate predicates for primitive types (e.g., `int`, `long`) and identifies repeated patterns to generate reusable refinement aliases.

Listing 5.1 presents the annotated `Socket` class. The highlighted portions show the refinements added by the agent, including imports for compilation and constraints on arguments with defined ranges—for instance, `port` values must fall within 0 to 65535. Since this predicate appears in multiple methods, the agent introduces an alias to avoid repetition. We show only one use in this condensed example; the full output contains additional applications of the same pattern.

With both the test suite and the annotated class in place, we can now evaluate whether LiquidJava correctly detects all protocol violations in the test suite.

```java
import liquidjava.specification.ExternalRefinementsFor;
import liquidjava.specification.StateRefinement;

import java.net.InetAddress;
import java.net.SocketAddress;

@StateSet({"bothshutdown", "bound", "closed", "connected", "inputshutdown", "outputshutdown"
    , "unconnected"})
@RefinementAlias("ValidPort(int port) { port >= 0 && port <= 65535 }")
@RefinementAlias("NonNegativeTimeout(int timeout) { timeout >= 0 }")
@RefinementAlias("ValidUrgentData(int data) { data >= 0 && data <= 255 }")

public interface SocketRefinements {
    // Constructors
    @StateRefinement(to = "unconnected(this)")
    public void Socket();

    @StateRefinement(to = "connected(this)")
    public void Socket(InetAddress address, @Refinement("ValidPort(_)") int port);

    // Connection methods
    @StateRefinement(from = "unconnected(this)", to = "bound(this)")
    public void bind(SocketAddress bindpoint);

    @StateRefinement(from = "unconnected(this)", to = "connected(this)")
    @StateRefinement(from = "bound(this)", to = "connected(this)")
    public void connect(SocketAddress endpoint, @Refinement("NonNegativeTimeout(_)") int
        timeout);

    @StateRefinement(from = "connected(this)")
    @StateRefinement(from = "inputshutdown(this)")
    @StateRefinement(from = "outputshutdown(this)")
    public void sendUrgentData(@Refinement("ValidUrgentData(_)") int data);

    // Shutdown methods
    @StateRefinement(from = "connected(this)", to = "inputshutdown(this)")
    @StateRefinement(from = "outputshutdown(this)", to = "bothshutdown(this)")
    public void shutdownInput();

    @StateRefinement(from = "connected(this)", to = "outputshutdown(this)")
    @StateRefinement(from = "inputshutdown(this)", to = "bothshutdown(this)")
    public void shutdownOutput();

    @StateRefinement(to = "closed(this)")
    public void close();
}
```

Listing 5.1: Generated LiquidJava annotations for the `Socket` class with typestate and argument refinements.

5.3.4 Implementation Details

We implemented this approach using Claude Code[3] with the Sonnet 4.5 model, which ranked as the top model for software engineering tasks on the SWE-Bench leaderboard [134] at the time of writing. We provide the task descriptions as prompts for the agents and describe which tools are available. Each agent receives a task description as a prompt along with a specification of available tools. All static tools follow the Model Context Protocol (MCP), allowing agents to invoke them as needed. We provide documents with specialized information, such as typestate definitions and LiquidJava syntax guides, as skills[4], which agents can access whenever relevant context is required.

To orchestrate the complete workflow, we defined a custom command[5] that invokes all agents in sequence. This makes the approach easier to use for generating specifications for a class requires only a single command such as `claude -p "\synthesize-refinements for the class Socket with documentation at PATH"`.

5.4 Evaluation Methodology

To evaluate our agentic approach, we compare automatically generated specifications against expert-written ones across a diverse set of Java classes known to exhibit typestate protocols. For our evaluation, we focus on two research questions:

RQ1 To what extent are the generated specifications effective in correctly capturing object protocols?

RQ1.1 To what extent do the generated specifications correctly distinguish between **valid and invalid protocol** usages in the held-out test suite?

RQ1.2 How do the automatically generated specifications compare to **expert-written specifications** in terms of DFA correctness, state coverage, transition accuracy, and refinement correctness?

RQ2 What is the impact of **key design choices** such as DFA-guided generation, test-driven validation, and verification-driven refinement, on specification generation success?

This section describes our dataset selection, the manual annotation process, the automated generation procedure, and the metrics we use for comparison.

5.4.1 Dataset Selection

We selected 35 classes from a prior study on object protocols in Java by the Beckman et al. [14] with available online data.[6] We selected classes available in JDK17, the version currently supported by LiquidJava, that are public classes with public constructors, and for which a concrete implementation exists. The final list includes classes with different protocol types, domains, and sizes, as presented in Table 5.1. We included classes from

[3] https://www.claude.com/product/claude-code
[4] https://code.claude.com/docs/en/skills
[5] https://code.claude.com/docs/en/slash-commands
[6] https://www.nelsbeckman.com/research/esopw/

Table 5.1: Dataset of 35 Java classes with typestate protocols, grouped by domain. Links point to JDK17 Javadoc documentation. For each domain, we report the number of classes of that domain and the ranges of number of methods across classes.

Domain	Classes	#	Sizes
Graphics	PopupMenu, Window, Clipboard, DragSourceContext, DropTarget, FlatteningPathIterator, ImageReadParam, ImageWriteParam, DefaultMutableTreeNode, AbstractUndoableEdit, UndoManager	11	5–89
I/O	BufferedInputStream, BufferedReader, PipedOutputStream, PipedWriter, DeflaterOutputStream, ZipFile, ZipOutputStream	7	7–14
System	ThreadGroup, Throwable, UUID, Vector, MLet, RequiredModelMBean	6	15–53
Network	DatagramSocket, ServerSocket, Socket, RMIConnector	4	13–54
Security	ChoiceCallback, KerberosKey, KerberosTicket, LoginContext	4	8–26
Data	SQLException, TransformerException, XMLFilterImpl	3	13–35
	Total	**35**	**5–89**

all different *protocol types* identified in the original study, such as Initialization, Liveness Check, Deactivation Check, Boundary Check, Redundant Request, and Marker State. We covered multiple *domains* including I/O, Networking, Graphics, Security, System, and Data. And, finally, classes of varying *size*, ranging from 5 to 89 public methods, to assess how class size affects generation quality.

5.4.2 Manual Expert Annotation

To establish a ground truth for comparison, two authors independently annotated each class following a standardized protocol. We developed an annotation guide specifying the expected outputs and decision criteria. For each class, annotators produced a LiquidJava specification file containing `@StateSet`, `@StateRefinement`, and `@RefinementAlias` annotations, along with an optional DFA diagram and tests with example usages. Beyond the specification itself, annotators recorded metadata to support later analysis: the time spent on the annotation, the number of states and transitions identified, the number of methods annotated (and how many included argument refinements), and any refinement aliases created. Annotators also documented their decision rationale for ambiguous cases such as unclear state transitions, methods callable in multiple states, or uncertain self-loops, and noted any tool limitations where properties could not be expressed in LiquidJava. After independent annotation, each author pair met to reconcile their specifications for each class. They compared their DFAs, discussed disagreements, and agreed on a single canonical specification. This reconciliation process ensures that our ground truth reflects careful consideration of ambiguous documentation rather than a single annotator's interpretation.

5.4.3 Automated Specification Generation

We plan to run our agentic workflow on each of the 35 classes using their official Javadoc documentation as input. For each class, the system produces a DFA representing the typestate protocol, a LiquidJava specification file with state and argument refinements, and a test suite exercising the protocol. To answer RQ2 regarding component contributions, we also plan to run ablation variants of the workflow. Specifically, we evaluated configurations with and without DFA-guided generation, test-driven validation, and verification-driven refinement to isolate the impact of each design choice.

5.4.4 Comparison Metrics

We evaluate the generated specifications along two dimensions: bug detection effectiveness (RQ1.1) and structural similarity to expert specifications (RQ1.2). For bug detection, we assess whether the generated specifications correctly distinguish valid from invalid protocol usages. We use the test suites produced by our system, which include both passing tests (valid method sequences) and failing tests (protocol violations). A specification is effective if LiquidJava reports errors for invalid tests and accepts valid ones.

For structural comparison, we treat DFAs as labeled graphs and compute their similarity using Graph Edit Distance (GED) [25, 71] with a method-focused cost model. This approach measures the minimum cost to transform one DFA into another, considering node insertions, deletions, and substitutions as well as edge operations. This metric allows us to quantify how closely automatically generated protocols match expert-written ones in terms of states identified, transitions captured, and methods covered.

5.5 Progress and Expected Contributions

Currently, we are implementing the agentic workflow and finishing the expert manual annotation of the classes. The examples presented in this chapter (i.e., `Socket` class), were already generated using the implemented system, and are, thus, showing promising preliminary results. Once the manual annotations are complete, we will run the full evaluation as described in the previous section. We expect to complete this evaluation and submit the results for publication within the next few months.

We expect this work to contribute:

- An agentic workflow that generates LiquidJava specifications from class documentation, combining DFA generation, test synthesis and verification-driven refinement.
- A benchmark of 35 Java classes with expert-written LiquidJava specifications covering diverse domains and protocol types.
- An empirical evaluation quantifying the effectiveness of automatically generated specifications in bug detection and their structural similarity to expert annotations, along with an analysis of key design choices.

Additionally, for LiquidJava we also have a list of tool improvements identified during the expert annotation process, which we plan to implement and release as part of the

LiquidJava open-source project.[7] Some of these limitations, from preliminary results, also emphasize the need for aliasing tracking and more expressive refinements, which we plan to address in the next chapter Chapter 6.

[7] https://github.com/liquid-java/liquidjava

6 Liquid Types and Alias Tracking in LiquidJava

In this chapter, we present an approach to extend LiquidJava's refinement types and typestate with support for alias tracking to handle object mutation. To this end, we introduce *Latte*, a type system that tracks aliasing and uniqueness while aiming to minimize both the annotation burden and the complexity of invariants that developers must reason about in an object-oriented language with mutation. An initial version of *Latte* was presented at the WITS and HATRA workshops [156]. The work in this chapter joins the ideas presented in those workshops and extends them by combining alias tracking with liquid types and typestate, as well as providing a formalization of the unified system.

6.1 Introduction

Java programs rely heavily on mutable objects that interact through shared references. A method may modify a field in one object that another object depends on, and without tracking who points to what, a verifier cannot determine which guarantees still hold after a mutation. Liquid types can catch many classes of errors at compile time, from simple range violations to protocol misuse, but to verify programs that mutate objects and pass references between methods, a liquid type system also needs to reason about aliasing. Without this information, the verifier cannot determine which objects a method call may affect, making it impossible to maintain sound refinements across mutations.

The previous version of LiquidJava, presented in Chapter 4, focused on adding liquid types and typestate to Java but only reasons about the current object (`this`). It cannot express constraints that span multiple objects or track how references flow through assignments and method calls. Listing 6.1 illustrates this limitation with a common pattern from the `java.awt` library.

```
1 Frame frame = new Frame(false);
2 Window w = new Window(frame);
3 Color c = new Color(0.5f);
4 w.setBackground(c);  // Runtime exception!
```

Listing 6.1: Example of creating an java.awt.Window that produces a runtime exception.

Here, the precondition of `setBackground` requires that either the window's underlying frame is undecorated or the color's alpha value equals 1.0, a constraint that spans three

objects. Verifying this statically requires knowing that `w.frame` has the contents of `frame` and tracking the state of `frame.undecorated` and `c.alpha` through prior method calls. The previous version of LiquidJava has no mechanism to express or verify such cross-object relationships, and this limitation became concrete during the manual annotation effort in Chapter 5, where we repeatedly encountered specifications that required multi-object pre- and post-conditions.

This chapter aims to fill that gap. We introduce *Latte*, a lightweight type system for uniqueness and aliasing tracking in Java. *Latte* focuses exclusively on tracking which references point to which objects and what operations each reference allows; it does not, by itself, reason about value constraints. LiquidJava (Chapter 4), on the other end, provides refinement types and typestate but cannot track how references flow through assignments and method calls. Neither system alone can verify cross-object constraints like those in Listing 6.1, but together they can express and verify them. To bridge aliasing information with refinement checking, our approach relies on three core mechanisms: symbolic values to represent objects and track their evolving state, uniqueness permissions to determine which objects a method may mutate, and havoc operations with state updates to model the effects of method calls on the heap. *Latte* requires only a few annotations: two (`unique` and `shared`) for object fields and return types, and three (adding `borrowed`) for method parameters; we infer the remaining information for local variables.

We plan to formalize this combined type system on an extended Featherweight Java [81] core language, with an initial formalization already underway. We intend to implement the system as an extension of the existing LiquidJava prototype and evaluate it on real-world Java libraries, including those identified in Chapter 5, to assess whether the added expressiveness enables the verification of programs that were previously out of reach.

6.2 Approach

To prevent the runtime exception in Listing 6.1, the verifier needs to know what the `setBackground` method actually requires. When we look at the documentation of `java.awt.Window`, we find pre-conditions that involve the state of multiple objects. In particular, focusing only on the conditions related to the `Frame` object for simplicity:

> All the following conditions must be met to enable the per-pixel transparency mode for this window:
>
> - The window must be undecorated (see Frame.setUndecorated(boolean) and Dialog.setUndecorated(boolean))
>
> IllegalComponentStateException - if the alpha value of the given background color is less than 1.0f and the window is decorated

Java's type system cannot catch this error, and LiquidJava without alias tracking cannot either, since the precondition spans multiple objects. However, with the right annotations and aliasing information, we can verify it statically. Listing 6.2 shows how we annotate the relevant `java.awt` classes using the combined LiquidJava and *Latte* system. We use

the Featherweight Java syntax for simplicity, with additions for the *Latte*'s uniqueness annotations (`unique`, `borrowed`, `shared`), and for LiquidJava's state transitions with refinement types (`@{ pre- » post-condition}`).

We define that a `Frame` has a boolean field `undecorated` that tracks whether the frame is decorated, and a `Color` has a float field `alpha` constrained between 0.0 and 1.0. Since the fields of these classes are primitive values (`boolean` and `float`), they do not require ownership annotations, since they do not carry object references and cannot be aliased. The `Window` class holds a `unique` reference to a `Frame`, meaning the window owns the frame. In the constructor, we require receiving a `unique Frame` reference, which the constructor then stores in its field and uses in its post-condition saying that the state of the frame remains unchanged. When the constructor is called, the caller must give up ownership of the `Frame` object, so it cannot be used after the call, but the constructor can refer to its state in the post-condition.

Now, to verify the client code, the key annotation is the pre-condition of `setBackground`, as it requires that either the *frame is undecorated or the color's alpha equals 1.0* directly encoding the documented constraint. The post-conditions use `old(...)` to state that fields not modified by the method retain their previous values.

Given these annotations, the client code in Listing 6.1 fails verification because neither condition holds: the frame is decorated (`undecorated == false`) and the alpha is 0.5. However, to reach this conclusion, the verifier must track that `w.frame` has the same values as `frame` had when the constructor was called, follow chains of field accesses like `w.frame.undecorated`, determine which objects a method call may modify, and update the known state accordingly. To do so, we focus on three ideas to complement the refinement type checking: using symbolic values to track objects and their evolving state, defining uniqueness permissions to determine which objects can be mutated when calling a method, and using havoc operations with state updates to reflect the changes in object state after method calls.

Symbolic values to track objects and their state. In Java, an object's fields change over time as methods mutate its state. To reason about these changes, we introduce symbolic values, as new identifiers that represent an object or field at a specific point in the program. Each time a field is assigned, we create a new symbolic value and record the corresponding refinement, preserving older symbolic values, so the verifier can refer to both current and previous states.

Listing 6.3 illustrates this idea for the `undecorated` field of the `Frame` class. Each assignment to the field creates a fresh symbolic value that captures the new state. After line 1, the symbolic environment maps `f.undecorated` to ν_1 with the refinement $\nu_1 == false$. After line 2, it maps the same field to a new symbolic value ν_2 with $\nu_2 == true$. Crucially, the old symbolic value ν_1 remains available, so we can reason about the state of the object at different points in the program. Beyond tracking field state, symbolic values also capture relationships between objects when the uniqueness permissions allow for it.

```
class Frame {
    boolean undecorated;

    @({true} » {this.undecorated == undecorated})
    Frame(boolean undecorated) {
        this.undecorated = undecorated;
    }

    @({true} » {this.undecorated == undecorated})
    void setUndecorated(boolean undecorated) {...}
}

class Color {
    @Refinement(alpha >= 0.0 && alpha <= 1.0)
    float alpha;

    @({alpha >= 0.0 && alpha <= 1.0} » {this.alpha == alpha})
    Color(float alpha)  { this.alpha = alpha; }
}

class Window {
    unique Frame frame;

    @({true}
      »
      {this.frame.undecorated == old(frame.undecorated)})
    Window(unique Frame frame) {
        this.frame = frame;
    }

    @({this.frame.undecorated == true || c.alpha == 1.0}
      »
      {this.frame == old(this.frame) &&
       this.frame.undecorated == old(this.frame.undecorated) &&
       c.alpha == old(c.alpha)})
    void setBackground(borrowed Color c) {...}
}
```

Listing 6.2: Annotating java.awt.Window with LiquidJava and *Latte*.

```
frame.undecorated = false; // ν_f.undecorated ↦ ν_1;  ν_1 == false
frame.undecorated = true;  // ν_f.undecorated ↦ ν_2;  ν_2 == true
```

Listing 6.3: Symbolic values to capture object state at different points in the program.

Uniqueness Permissions. When a method is called, the verifier needs to know which objects the method may modify. We make this explicit by annotating parameters and fields with one of three uniqueness permissions.

- `unique` reference is the only reference to an object, so it can be safely mutated without affecting other parts of the program.
- `shared` reference may have untracked aliases, so mutations through it are not permitted since other references could observe the object in an inconsistent state.
- `borrowed` reference provides temporary access to an object owned elsewhere the method can read from it and refer to its state in pre- and post-conditions, but does not claim ownership.

These annotations allow the verifier to accurately track the state of unique objects across method calls, and refer to the state of borrowed objects without claiming ownership. For example, in Listing 6.4, the `frame` is passed as a `unique` argument to the `Window` constructor. Therefore, we are able to assign it to the `unique` field `frame`. Additionally, the post-condition refers that `this.frame.undecorated == old(frame.undecorated)`, which we can track combining the symbolic values and the uniqueness permissions.

```
1 Frame frame = new Frame(false);
2 // frame ↦ ν_f, ν_f.undecorated ↦ ν_2; ν_2 == false
3 Window w = new Window(frame);
4 // w ↦ ν_w; ν_w.frame ↦ ν_wf; ν_wf.undecorated ↦ ν_3;
5 // ν_2 == false ∧ ν_3 == ν_2
6 // ν_w.frame: unique, ν_f: inaccessible
```

Listing 6.4: Combining ownership and symbolic values to capture object state in the constructor call.

After the constructor call, the verifier adds new symbolic values to represent the window object and its fields. The predicates will now include the post-condition of the constructor in relation to the symbolic values ($\S\nu_3 == \nu_2\S$), and will record that `frame` is now inaccessible since its ownership has been transferred to the window.

Havoc operations and state updates. When a method mutates an object, some constraints in the verifier's context become stale. The verifier must determine which facts to discard and which new facts to add to the context. We handle this through a forget-then-relearn pattern: first, we *havoc* the symbolic values of fields that might have changed, and then we *update* the context with the post-condition of the method with the available symbolic values.

Listing 6.5 illustrates this pattern. After constructing the `Frame` on line 9, the verifier knows that ν_f.undecorated $\mapsto \nu_1$ with $\nu_1 == false$. When we call `setUndecorated(true)` on line 10, the verifier must reconcile the old state with whatever the method promises.

In the first step, we havoc all fields of objects reachable in the method's context which includes the receiver (`this`), explicitly passed arguments, any symbolic values reachable from these, and all `shared` objects. For each field, we create a fresh symbolic value and record a mapping from the new value to the old one. In this example, the only reachable

```
1  class Frame{
2      @({true} » {this.undecorated == undecorated})
3      Frame(boolean undecorated) {...}
4
5      @({true} » {this.undecorated == u})
6      void setUndecorated(boolean u) {...}
7  }
8  // Client code
9  Frame f = new Frame(false); // νf.undecorated ↦ ν1; ν1 == false
10 f.setUndecorated(true); // What is are preserved? What is updated?
```

Listing 6.5: Example for the need of havoc operations.

mutable field is `undecorated`, so we replace ν_1 with a fresh ν_2 about which we know nothing yet:

`Frame f = new Frame(false);`
ν_f.undecorated $\mapsto \nu_1$
$\nu_1 == false$

`f.setUndecorated(true);`
1. Havoc:
ν_f.undecorated $\mapsto \nu_2$
$\nu_1 == false$
Old mapping: $\nu_2 \mapsto \nu_1$

In the second step, we add the method's post-condition to the refinement path, instantiated with the actual arguments. The post-condition of `setUndecorated` states `this.undecorated == u`, which resolves to $\nu_2 == true$:

`f.setUndecorated(true);`
2. Update with post-condition:
ν_f.undecorated $\mapsto \nu_2$
$\nu_1 == false \wedge \nu_2 == true$
Old mapping: $\nu_2 \mapsto \nu_1$

The verifier now knows the current state of the field ($\nu_2 == true$) while retaining the old state ($\nu_1 == false$). This old mapping is what allows post-conditions with `old(...)` expressions to work: when a post-condition refers to `old(this.undecorated)`, the verifier resolves it through the mapping from ν_2 to ν_1. The pattern ensures we never hold contradictory facts about the same symbolic value as we conservatively forget what might have changed, then relearn exactly what the method guarantees.

Typing Environments. To make these three mechanisms precise, we define three environments that the verifier maintains during type checking, summarized in Table 6.1. The typing environment Γ maps variables to their types, as in a standard type system, and is used for type checking expressions and statements. The symbolic environment Δ maps variables and fields to symbolic values, recording which symbolic value each reference currently points to. The permission environment Σ maps each symbolic value to an ownership permission, determining what operations are allowed on that value. The refinement path φ accumulates the predicates that the verifier has learned, forming a conjunction of facts about symbolic values that grows as the program executes.

Symbol	**Environment**	**Formal Definition**	**E.g.,**
Δ	Symbolic environment	$\Delta ::= \emptyset \mid x{:}\,\nu, \Delta \mid \nu.f{:}\,\nu, \Delta$	ν`.frame` $\mapsto \nu_f$
Σ	Permission environment	$\Sigma ::= \emptyset \mid \nu{:}\,\alpha \mid \nu{:}\,\bot$	ν_f : `unique`
φ	Refinement path	$\varphi = true \mid \rho_1 \wedge \rho_2 \wedge \ldots \wedge \rho_n$	$\nu_2 == true$

Table 6.1: Environments for tracking symbolic values, permissions and refinements.

We use ν to denote symbolic values, α to denote permissions, and ρ to denote individual refinement predicates. The symbols x and f represent variables and fields, respectively. We use $\bot$ in the permission environment to represent inaccessible references, for example, a unique reference that has been consumed by an assignment. Primitive values (e.g., booleans and floats) have immutable values and therefore carry a special `imm` permission.

The core typing judgment has the form $\Gamma; \Delta; \Sigma; \varphi \vdash s \dashv \Gamma'; \Delta'; \Sigma'; \varphi'$, where s is a statement, and the primed environments reflect the updated state after checking s. The system uses auxiliary judgments to handle argument evaluation and permission checking, determining reachable objects and invalidating stale facts (havoc), and incorporating method post-conditions. The full typing rules are under development; here we illustrate the key ideas through an example.

6.3 Illustrative example

If we go back to the client code in Listing 6.1, we can now see how the ideas presented above come together to verify that the code is unsafe. Let's apply these concepts step-by-step to the client code to find the error.

In the constructor of `Frame` we create new symbolic values to represent the new object and its fields, and add the post-condition to the refinement path.

```
Frame frame = new Frame(false);
    Δ : frame ↦ ν_f, ν_f.undecorated ↦ ν_u
    Σ : ν_f : unique, ν_u : imm
    φ : ν_u == false
```

When calling the constructor for `Window`, the frame is passed as `unique`. Since the pre-condition is `true`, we just need to apply havoc for the reachable fields, in this case

only `f.undecorated` (becoming ν_{u1}). Then we can add the new symbolic values for the return object and its fields, and resolve the post-condition to use the symbolic values. Since the post-condition uses old(ν_{u1}) to refer to the previous state of the field, we use the old mapping to resolve it to ν_u, and the refinements' path knows that $\nu_u == false$. Since the constructor takes ownership of the frame, we mark ν_f as inaccessible in the permission environment.

```
Window w = new Window(frame);
```
1. Check pre-condition: `true` ✓
2. Havoc:
   $\Delta : frame \mapsto \nu_f, \nu_f.\text{undecorated} \mapsto \nu_{u1}$
   $\Sigma : \nu_f : \text{unique}, \nu_u : \text{imm}$
   $\varphi : \nu_u == false$
   $Old : \nu_{u1} \mapsto \nu_u$
3. Add symbolic values for the new object:
   $\Delta : frame \mapsto \nu_f, \nu_f.\text{undecorated} \mapsto \nu_{u1},$
   $\text{w} \mapsto \nu_w, \nu_w.\text{frame} \mapsto \nu_{fnew}, \nu_{fnew}.\text{undecorated} \mapsto \nu_{unew}$
   $\Sigma : \nu_f : \text{unique}, \nu_u : \text{imm}, \nu_w : \text{unique}, \nu_{fnew} : \text{unique}, \nu_{unew} : \text{imm}$
4. Resolve post-condition and update refinements' path:
   `this.frame.undecorated == old(frame.undecorated)` is $\nu_{unew} == \nu_u$
   $\varphi : \nu_u == false \wedge \nu_{unew} == \nu_u$
5. Update access permissions:
   $\Delta : frame \mapsto \nu_f, \nu_f.\text{undecorated} \mapsto \nu_{u1},$
   $\text{w} \mapsto \nu_w, \nu_w.\text{frame} \mapsto \nu_{fnew}, \nu_{fnew}.\text{undecorated} \mapsto \nu_{unew}$
   $\Sigma : \nu_f : \bot, \nu_u : \text{imm}, \nu_w : \text{unique}, \nu_{fnew} : \text{unique}, \nu_{unew} : \text{imm}$
   $\varphi : \nu_u == false \wedge \nu_{unew} == \nu_u$

For the `Color` constructor, we create a new symbolic value for the object and its field, and add the post-condition to the refinements' path.

```
Color c = new Color(0.5f);
```
$\Delta : \cdots, \text{c} \mapsto \nu_c, \nu_c.\text{alpha} \mapsto \nu_{c1}$
$\Sigma : \cdots, \nu_c : \text{unique}, \nu_{c1} : \text{imm}$
$\varphi : \nu_u == false \wedge \nu_{unew} == \nu_u \wedge \nu_{c1} == 0.5$

Now we have all the information to check the pre-condition of `setBackground`, and notice that it does not hold since the frame is decorated ($\nu_{unew} == false$) and the alpha is 0.5 ($\nu_{c1} == 0.5$), which leads to a verification failure. We can therefore, statically catch the error that would lead to the runtime exception in the original code.

```
w.setBackground(c); //Runtime exception!
```
1. Check pre-condition:
   `this.frame.undecorated == true || c.alpha == 1.0`

- Resolve pre: this.frame.undecorated is ν_{unew}, $c.alpha$ is ν_{c1}
- SMT call:

$$\nu_u == false \land \nu_{unew} == \nu_u \land \nu_{c1} == 0.5 \implies \nu_{unew} == true \lor \nu_{c1} == 1.0 \times$$

6.4 Related Work

Aliasing and uniqueness systems for OO languages Reasoning about aliasing in object-oriented languages has been explored through several lines of work, including ownership types [37, 38], linear types [144], and reference capabilities [30]. Ownership types restrict access to objects based on their owners, but classic ownership alone does not track aliasing within ownership boundaries, making it difficult for verification tools to reason about the effects of assignments. AliasJava [6] extends Java with annotations (`unique`, `owned`, `lent`, `shared`) and reduces the annotation burden through inference. Our approach is similar in spirit, but achieves greater simplicity by removing ownership parameters and allowing more local aliasing within methods. Alias burying [21] defines uniqueness without destructive reads, but as noted by Boyland and Retert [22], it exposes implementation details such as the fields read by a method, which breaks encapsulation. Reachability Types [10] use reachability sets to reason about ownership and track reachable values through type qualifiers, but target a higher-order functional setting rather than Java. Castegren and Wrigstad [30] combine ownership types, linear types, and regions in their κ language, but rely on reference capabilities not present in Java. In summary, prior work in this space tends toward either high expressiveness at the cost of complex abstractions and developer effort, or reliance on language features that Java does not provide. *Latte* addresses this trade-off by prioritizing a lightweight design with few annotations, no changes to Java semantics, and support for common coding patterns, while still tracking aliasing precisely enough to support more advanced type systems such as liquid types.

Liquid Types and Ownership. Liquid Types were originally designed for pure functional languages, where aliasing and mutation are largely absent. LiquidHaskell [142] exemplifies this setting as refinements constrain values, but the absence of mutable state sidesteps the need for ownership or alias tracking. Extending refinement types to imperative and systems languages requires confronting how mutation and aliasing interact with logical invariants. Low-Level Liquid Types[126] addressed this for a C-like language by introducing abstract locations that summarize possibly-aliased regions; an unfold/fold discipline controls when invariants must hold, effectively treating abstract locations as owned regions. CN [122] takes a different approach for systems C, combining refinements with separation logic assertions so that heap mutation consumes and produces resources, and aliasing is controlled through disjointness. ConSORT [139] integrates refinement types with fractional permissions in

an imperative setting, where full ownership (fraction 1) permits mutation and fractional ownership (less than 1) provides read-only access to aliased references. Rely-Guarantee References [75] allow aliasing under controlled interference: each reference carries rely and guarantee conditions describing what updates it may observe and perform, with uniqueness recovered when the rely is empty. F* and its Low* subset [121] use an effect system with pre/postconditions over heaps, encoding ownership of buffers in types and effects while supporting controlled sharing protocols. Most closely related to our work, Flux [94] brings SMT-based liquid refinements to Rust, leveraging the borrow checker's guarantees to enable strong updates at unique owners while treating shared references as read-only at the refinement level. Our approach draws inspiration from Flux's integration of refinements with ownership, but targets Java's object-oriented setting where Rust's ownership model is unavailable.

6.4.1 Progress and Expected Contributions

We have designed Latte, a lightweight alias tracking system for Java, and presented initial versions at the WITS and HATRA workshops. We have defined the core typing judgment and environments for the combined Latte and LiquidJava system on extended Featherweight Java, and implemented a standalone Latte prototype, available as open source.[1]

This chapter will contribute:

- ***Latte***, a lightweight alias tracking system for Java that minimizes annotation burden while enabling verification of mutable, aliased objects *(designed; prototype implemented)*
- **A unified type system** combining *Latte*'s uniqueness tracking with LiquidJava's refinement types and typestate, enabling verification of cross-object pre- and post-conditions *(core judgment defined; rules in progress)*
- **A formalization** on extended Featherweight Java with a type safety proof *(proposed)*
- **An integrated implementation** extending the LiquidJava prototype *(proposed)*
- **An evaluation** demonstrating that the combined system can verify programs previously out of reach *(proposed)*

The addition of alias tracking will allow us to verify more properties of real-world Java libraries, particularly those involving complex interactions between mutable objects, which were previously out of reach for LiquidJava alone. However, we expect that the added expressiveness will introduce trade-offs in annotation burden and usability. For annotation burden, we leave for future work the development of inference techniques to reduce the need for explicit annotations, or an approach similar to Chapter 5 that automatically suggests annotations for existing code. For usability, we recognize that verification errors become harder to interpret as the system grows more expressive. In Chapter 7, we address this challenge by improving LiquidJava's error diagnostics and conducting a user study that includes tasks with both LiquidJava alone and the combined *Latte* and LiquidJava system.

[1] https://github.com/liquid-java/latte

7 Verification Feedback in LiquidJava

This chapter presents our ongoing work on improving verification feedback for LiquidJava. We describe the design principles we developed for communicating verification state and outcomes to developers, and show how we are currently applying them to LiquidJava. These principles are based on our prior usability studies (Chapters 3 and 4) as well as related work on error messages for advanced type systems. We then outline our planned user study to validate that these improvements to help developers understand and resolve verification failures more effectively. This user study can include different error types, including both refinement failures, typestate violations, and (if possible) aliasing errors from the Latte extension (Chapter 6). Since the Latte integration remains in progress, the aliasing-specific feedback and comprehensive evaluation represent future work; however, the design principles and core implementation for refinement verification are already in place.

7.1 Motivation

Traditional type systems rely on compilers to verify type correctness, catching errors like assigning a string to an integer variable at compile time, so before the program is executed. Compilers communicate verification outcomes to developers primarily through error messages, and modern IDEs enhance this feedback with features like inline highlighting, hover information, and quick fixes. Yet even for traditional type systems, this communication remains challenging, as developers consistently find compiler messages cryptic, verbose, or unhelpful [12, 51]. Languages like Rust [2] and Elm [3] have invested heavily in improving this feedback, introducing structured presentation, code suggestions, and extensible help [1, 34, 88, 99].

Advanced type systems augment these communication challenges, since verification can involve constraint solving, SMT reasoning, or proof obligations, which increase the gap between what the tool checks and what developers understand. Liquid Types present a particular challenge as verification relies on SMT solvers and refinement logic, and the verification process operates as a black box. When verification fails, developers receive an error message as the result of a failed proof but often have no insight into why it failed or what they should change.

Our usability study on Liquid Types (Chapter 3) confirmed these challenges, identifying *unhelpful error messages* as one of nine key barriers to adoption. Participants reported that verification feedback exposes internal representations, like variable names that have no correspondence to their code (e.g., `VV##0`), and displays verification conditions that

require developers to simulate SMT reasoning to understand the failure. Both newcomers and experienced users struggled with these issues, indicating that the challenge does not diminish. In fact, experienced users noted that verification feedback becomes increasingly unmanageable as codebases grow, often displaying full screens of Horn clauses that cannot be mapped back to source code.

The LiquidJava evaluation (Chapter 4) reinforced these findings. While participants appreciated having verification feedback integrated into the IDE, they noted that "some error messages are not straight to the point." This feedback, combined with the systematic barriers we identified, motivates a user-centered redesign of how LiquidJava communicates verification outcomes. We aim to bridge the gap between the verifier's internal reasoning and the developer's comprehension by surfacing verification state through simplified presentations, counterexamples, and interactive exploration tools. Furthermore, as we extend LiquidJava with aliasing tracking (Chapter 6), we should also consider the additional complexity of ownership verification and address new usability challenges that appear from that.

The remainder of this chapter is organized as follows. We begin with related work on verification feedback and debugging tools for advanced type systems (Section 7.2). We then present six design principles that guide our approach to improving verification feedback (Section 7.3), followed by our current implementation plans of these principles in LiquidJava (Section 7.4). Finally, we outline a planned user study to evaluate whether these improvements help developers understand and resolve verification failures (Section 7.5).

7.2 Related Work

Showcasing effective verification feedback is essential for developers to understand and resolve issues in their code. These feedback mechanisms can spam from textual error messages to interactive debuggers that expose verification reasoning.

Research on compiler error messages provides the basis for effective verification feedback. These error messages are a known as a pain point for developers, who often consider them as cryptic, verbose, or unhelpful, leading to many studies on how to design better textual error messages. Becker et al.'s survey [**?**] synthesizes work dating back to 1965, offering guidelines such as providing hints, using positive tone, and relating errors to source code, and Denny et al. [51] identify readability factors including shorter messages, simpler vocabulary, and economy of words. Beyond textual content, Dong et al. [54] show that visual presentation such as color, spacing, and headings, significantly impacts usability. Interactive environments like IDEs expand the design space further, enabling progressive disclosure and interactive exploration of errors [17].

However, as languages evolve to include more expressive type systems, communicating verification outcomes becomes increasingly challenging. Modern languages like Rust [1] and Elm [88] demonstrate that investment in verification feedback pays off, providing structured messages with code suggestions. But even in these languages, research continues on how to improve error messages for advanced features, like Rust's ownership model [45]. For other advanced type systems, such as ownership and typestate, Coblenz et al. [42]

found that the careful design significantly affects developer success. For dependent types, Eremondi et al. [58] propose a framework for error messages using replay graphs and counterfactual solving to improve verification feedback in Idris [23] and Agda [20]. Juhošová et al. [83] identify learning obstacles in interactive theorem provers, finding that unclear feedback and inadequate tooling hinder adoption. In auto-active verifiers that use SMT automation, Mugnier et al.[111] interviewed Dafny expert developers who also find difficulties in interpreting error messages and debugging failed proofs, and found helpful the ability of controlling the visibility of proof facts. ProofPlumber [35], for example, provides a framework to help developers apply proof debugging techniques on the source-level proofs to help them understand and recover from failures. For refinement types specifically, an extension to HayStack targets LiquidHaskell and aims to extract counterexamples from the SMT model and visualizing refinement type refutations, surfacing verification state that would otherwise remain hidden [145].

Interactive tools can go beyond static messages to expose verification reasoning directly. The Lean VS Code infoview [91] displays proof state in real time as developers navigate their code presenting the current context and the proof goals, while Paperproof [137] renders expandable proof trees for step-by-step exploration. ChameleonIDE [67] visualizes type inference through interactive exploration, helping developers untangle complex failures. In the study of the impact of formal verification on software development, Mugnier et al.[111] recommend having incremental and interactive contexts, showing what is relevant at a program point and provide a visualization of the verification state. Targeting Rust's trait errors, Argus [76] defines four principles to guide the presentation of inference failures and providing interactive exploration of the trait resolution process. A user study with 25 participants found that developers using Argus localized faults more accurately and faster than those using standard error messages, demonstrating the value of surfacing verification state.

7.3 Design Principles for Verification Feedback

We derived six design principles from three sources: the usability barriers identified in our developer study (Chapter 3), the feedback from the LiquidJava evaluation (Chapter 4), and previous work on guidelines for compiler error messages and debuggers for advanced type systems. Each principle addresses specific challenges developers face when interpreting verification state in Liquid Types.

P1 Distinguish Error Types. When verification fails, developers should quickly identify the category of error they are encountering. Our barrier study revealed that developers struggles to distinguish between syntax and type errors inside the refinements (e.g., "unexpected '-' expecting bareTyArgP") from those related to verification failures, causing developers to attempt wrong fixes. By classifying errors in different categories —syntax, refinement, state transition, and (in future) aliasing—developers can apply appropriate debugging strategies. This principle aligns with Becker et al.'s [12] identification of categorization as a key factor in effective error messages.

P2 Map Errors to Source Code. Verification feedback should directly relate to source-level constructs used in developers' code. The barriers study revealed that developers were confused by error messages that referred to internal variables (e.g., `VV##0`, `Queue a##a2cb`) with no connection to the source code, and that "don't mean anything to you in that context." The lack of syntax highlighting for refinements also endangered readability, and localization of possible issues. Additionally, developers struggled to identify the root cause of some errors given that the error location often pointed to a different place in code, the place where the error manifested rather than where it originated. To target these issues, this principle targets mapping the errors back to the source code, using developer's familiar language and constructs, and localizing the root cause of the error. This principle aligns with the idea that, when verification errors have multiple valid source locations, tools should localize the root cause rather than where the error manifests [76], and to Becker et al.'s [12] *mapping problem*, from relating transformed code back to source.

P3 Simplify the Presentation. Large queries displaying "a full screen of failed Horn Clauses" overwhelm developers. Participants also noted messages were "not straight to the point" and required "prior knowledge of the inner workings of the tool." One participant even mentioned resorting to manually shrinking the verification conditions to pinpoint the failure. Therefore, we aim to simplify the presentation of verification conditions to focus on the essential information needed to understand the failure. However, simplification involves tradeoffs, as showing "5 is not less than 4" loses context, and therefore we need to balance this simplification. Similarly to Argus's *ShortTys* concept [76], we aim to show simplified forms by default but make full expressions available on demand.

P4 Progressive Disclosure of Verification State. Developers feel the lack feedback about SMT reasoning, as participants resorted to "making proofs on paper" and "executing the function to test edge cases" to understand failures. Rather than hiding verification internals entirely, we expose them progressively, with a brief overview first, and with expandable details for those who want to dive deeper into the verification process. Instead of requiring participants to "have a prior knowledge of the inner working of the tool to understand the error", we can guide them through the verification steps. This strategy allows developers to explore the verification state at their own pace, catering to different expertise levels. This follows the idea of incremental and interactive contexts [111], that is implemented in several tools like Lean's infoview [91], Paperproof [137], and HayStack [145] with effective progressive disclosure for verification state.

P5 Provide Contextual Information. Developers struggled with "lack of context on what functions and type aliases are available" and wanted to know "what type aliases were already defined." This principle focuses on providing relevant context for developers to reason about the state of the program at a certain point, and

```java
void printUnderweight(@Refinement("bmi < 18") int bmi){
    System.out.println("You are underweight.");
    // Additional logic...
}

void messageBMI(){
    int weight = 100; //kgs
    int height = 190; //cms
    int bmi = (weight * 10000) / (height * height);
    printUnderweight(bmi); // Verification error here
}
```

Listing 7.1: Simple example of BMI calculation and print verification.

provide information about related constructs. Therefore, we enable developers to access contextual information on-demand, such as available type aliases, function signatures, and relevant program state. Additionally, we can show the current verification context by showing the current variable refinements, and illustrating the specified state transitions for a class.

P6 Guide Recovery. Move beyond stating *what* failed to suggesting fixes and hint to possible issues. We can reduce friction by guiding developers towards using the tool effectively, with IDE support for auto-complete and setup, as participants noted these were missing in LiquidJava, and by providing hints and examples for users. It is possible to include counterexamples that concretely demonstrate the failing case, similar to how type refutations [145] extracts refutations from the SMT model to show specific values that violate constraints. In prior work, Becker et al. [12] advocate for hints and examples.

7.4 Current Implementation in LiquidJava

The implementation of these design principles is ongoing work, with the implementation details being explored in a master thesis project by Ricardo Costa. Here, we showcase how we are currently applying each design principle in LiquidJava. To illustrate our approach, we use the example in Listing 7.1, which calculates the Body Mass Index (BMI) and prints a message if the user is underweight, with the input values leading to a verification failure. In the old version of LiquidJava, tested in the LiquidJava evaluation (Chapter 4), this code produces the verification error shown in Listing 7.2, with the highlight of the error in the IDE.

This error message exhibits several issues identified in our barrier studies that the design principles are targeting. For example, we can see the overwhelming presentation logic constraints in the section `Refinement Found`, which includes the constraints sent to the SMT solver, but requires the developer to replay the SMT reasoning to understand

```
Failed to check refinement at:
Method invocation printUnderweight(bmi) in:
printUnderweight(bmi)

Type expected:(#bmi_8 < 18)

Refinement found:true && #bmi_8 == #weight_5 * (10000) / #height_7 * #
    height_6 && #weight_5 == #weight_0 && #weight_0 == 100 && #height_7 ==
    #height_1 && #height_1 == 190 && #height_6 == #height_1
Instance translation table:
-------------------------------------------------------------------------
| Variable Name| Created in                        | File
-------------------------------------------------------------------------
| #weight_5    | weight                            | Example.java:12, 20
| #bmi_8       | int bmi = (weight * 10000) / (height * height) | ...
| #height_6    | height                            | Example.java:12, 48
| #height_7    | height                            | Example.java:12, 39
| #height_1    | int height = 190                  | Example.java:11, 13
| #weight_0    | int weight = 100                  | Example.java:10, 13
-------------------------------------------------------------------------
Location: (/Users/cgamboa/git/liquidjava/liquidjava-example/src/main/java/
    test/currentlyTesting/Example.java:13)
```

Listing 7.2: Old version Verification error output

the failure. The additional variables denoted by the `#` symbol (e.g., `#bmi_8`, `#weight_5`) represent internal variables created during verification, which have no correspondence to the source code, making it difficult for developers to map the error back to their code. The message does have a section of `Instance translation table` in the attempt of providing extra information on the mapping, but the large amount of information is not easy to parse and connect to the constraints and the code itself.

Based on the design principles outlined above, we started implementing improvements in the open-source LiquidJava's IDE extension for VS Code, [1] that already provided the error location and showed the old message (Listing 7.2). We have extended this extension with a side panel dedicated to verification feedback, where we render the improved error messages, allowing us to apply richer formatting and interactive features like progressive disclosure.

Figure 7.1 shows a prototype of the improved verification error output in LiquidJava, applying a set of these design principles. We separate different error types (Item **P1**) by clearly labeling the errors at the top, and allowing reporting several errors at once. We aim to map errors location to source code (Item **P2**) by highlighting the relevant line in the code, and using syntax highlighting inside predicates to improve readability. The presentation is simplified (Item **P3**) when compared to the full predicate shown in the old version, by applying constant propagation and folding to show concrete values rather than SMT expressions. In the initial version, the `Refinement Found` section shows an expression

[1] https://github.com/liquid-java/vscode-liquidjava

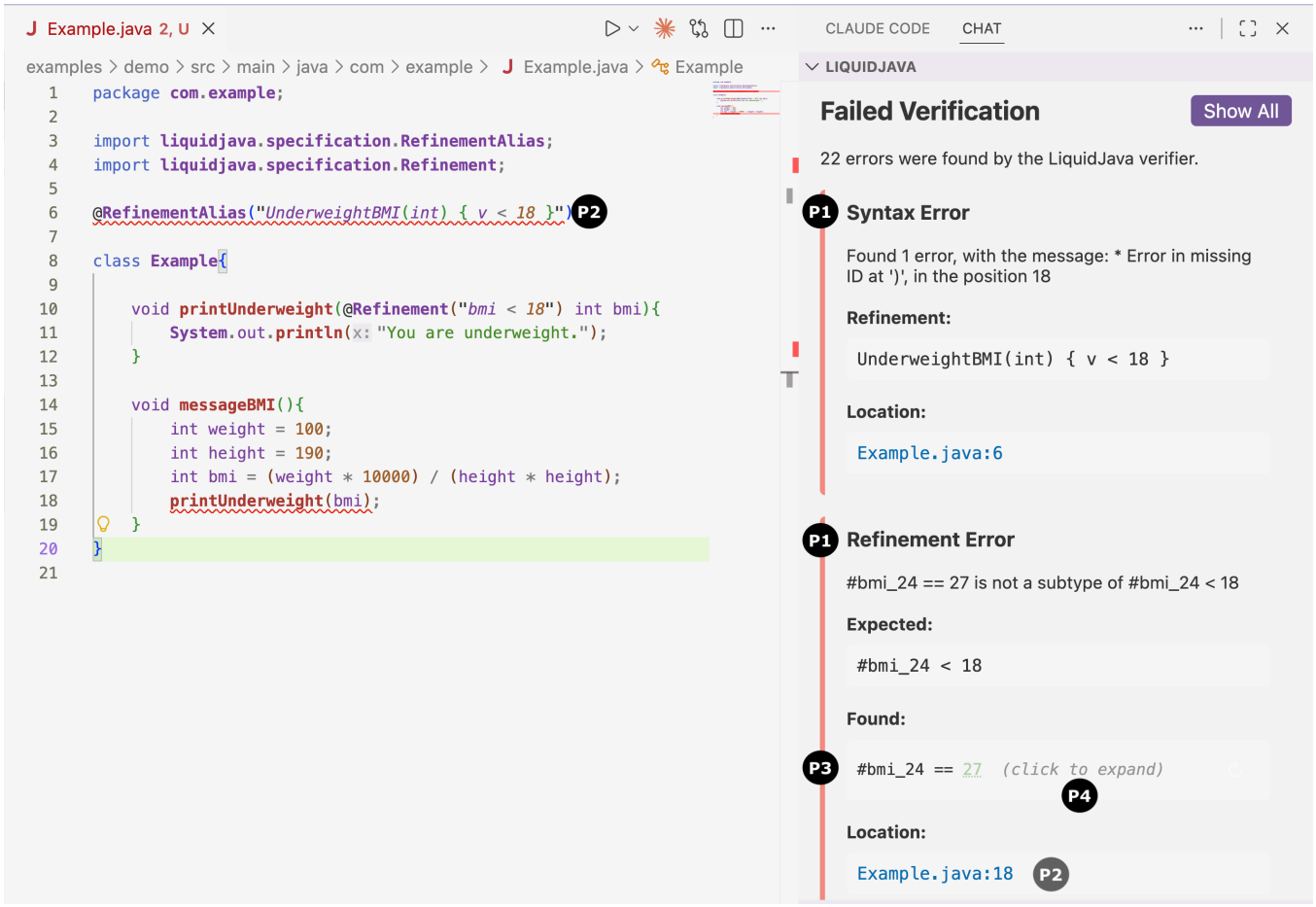


Figure 7.1: Prototype of improved verification error output in LiquidJava, applying the design principles for verification feedback (identified by number).

with eight predicates including the calculation of the `bmi` value (`#bmi_8 ==#weight_5 * (10000) / #height_7`), several transitive equalities (e.g., `#height_7 == #height_1`), and redundant predicates (`true`) which can be overwhelming. The simplified version shows only the core issue: `#bmi_24 == 27`, which is the computed value of `bmi` after constant folding, making it easier to understand that 27 is not less than 18. If the intermediate steps are needed, however, the full expressions remain accessible on demand via expandable clicking on the refinements found (Item **P4**).

We are still working on further applying these principles, and understanding the trade-offs involved. For example, while simplifying the presentation of the `bmi` calculation to `#bmi_24 == 27`, we are now introducing a new value that does not directly map to any one place in the source code, putting in contrast Item **P3** and Item **P2**. Additionally, the internal variable `#bmi_24` is still an internal representation of an intermediate program state

where the source code variable `bmi` had a given value, if it was assigned multiple times, we would need to be able to distinguish these states. The refinement error message text itself presumes the knowledge of how verification occurs internally, as it states `#bmi_24 == 27 is not a subtype of bmi < 18`, requiring the developer to understand what subtyping means in this context. Therefore, we have plans to improve these aspects further.

We are also working on providing contextual information (Item **P5**) by showing relevant type aliases and function signatures on demand, and a graph illustrating the state transitions for classes. Figure 7.2 shows a prototype of the state machine visualization for class states, which can be accessed from the verification feedback side panel, showcasing the state machine directly from the LiquidJava annotations on code. The example class is the one used in the LiquidJava evaluation (Chapter 4) for the `Socket` class, where developers found it challenging to fix the error in the code. Finally, we aim to guide recovery (Item **P6**) by including counterexamples that concretely demonstrate the failing case, and providing hints for possible fixes.

We are also exploring how to extend these improvements to aliasing errors from the Latte extension (Chapter 6), which introduces additional complexity in verification feedback due to ownership and aliasing tracking.

7.5 Planned User Study

To understand the effects of these improvements, we plan to conduct a user study comparing the new verification feedback against the old version tested in the LiquidJava evaluation (Chapter 4). We plan a study to answer the following research questions:

RQ1: How does improved verification feedback affect developers' ability to **resolve** different types of verification errors?

RQ2: How does improved verification feedback affect developers' **understanding** of verification state and outcomes?

RQ3: What aspects of verification feedback do developers find **most valuable** for diagnosing and resolving errors?

To this end, we conduct a randomized controlled experiment with two parallel groups. Participants are assigned either to the treatment group, using the improved IDE feedback (redesign), or to the control group, using the existing feedback (baseline). Following a waitlist (delayed-treatment) control design [8], control participants are given access to the improved IDE feedback only after completing all study tasks, enabling them to directly compare both versions while preserving unbiased primary outcome measures.

Figure 7.3 illustrates the study design. We plan to recruit around 30 participants with prior Java experience, but not necessary experience with verification or liquid types, similarly to how we recruited participants in the LiquidJava evaluation (Chapter 4). We will start with an interactive tutorial introducing LiquidJava with exercises for the participants to get familiar with the tool and the verification feedback they will use, so the redesign group will be able to interact with the new feedback design, and the baseline group with

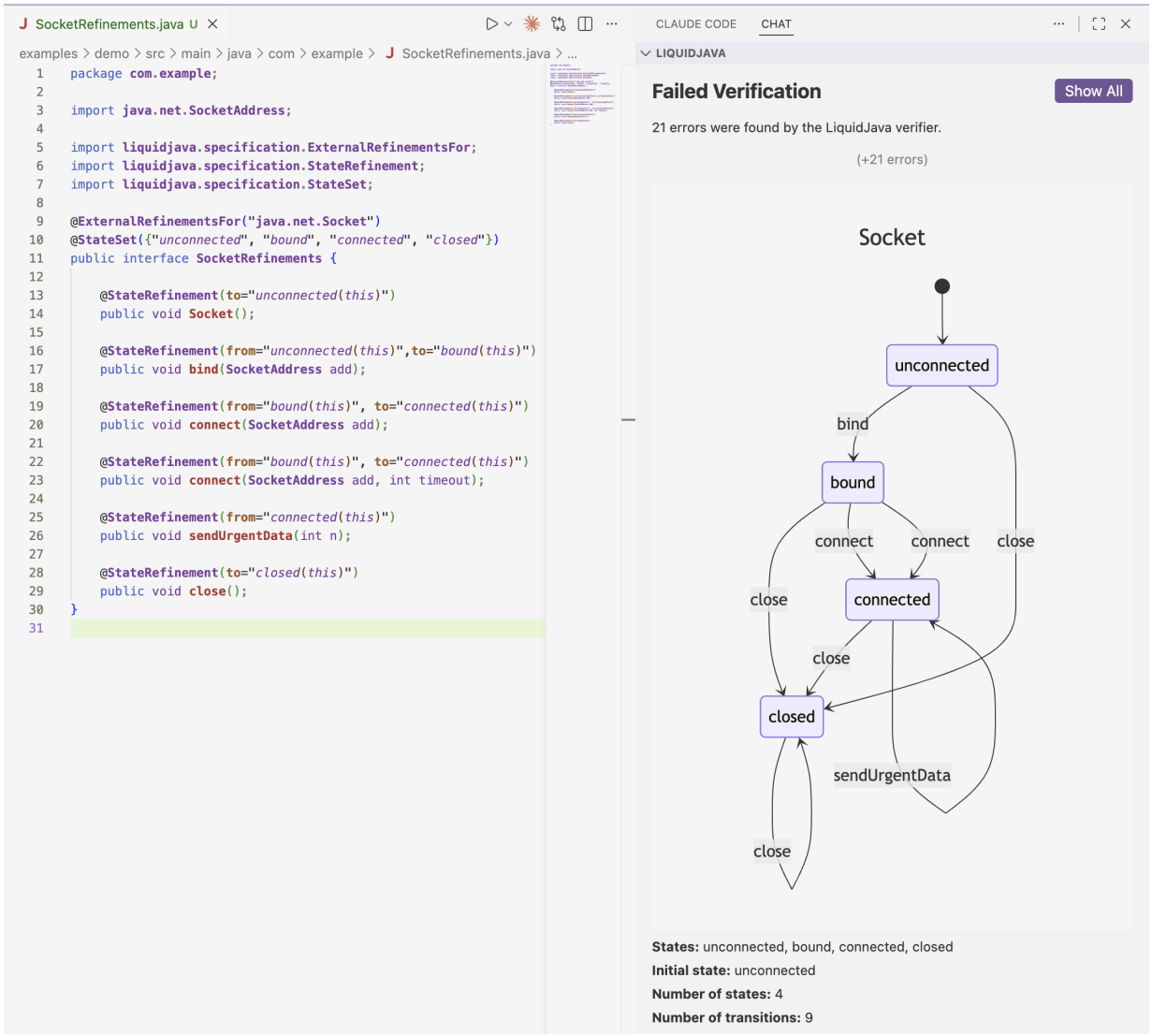


Figure 7.2: Side panel in IDE with the state machine visualization for class states.

the original feedback. The tutorial will cover basic concepts of LiquidJava and will contain the same exercises for both groups, differing only in the verification feedback presented.

After the tutorial, we will present participants with 3-4 of debugging tasks, where they will need to fix verification errors in LiquidJava code snippets (1) Fix errors). We will measure their success rate, time taken, and strategies used to resolve the errors. These exercises will include different error types, from examples that we have been collecting throughout the studies in this thesis, and publicly available in an open-source repository.[2] We aim to have exercises covering syntax errors inside refinements, refinement type errors, typestate violations, and (if possible) aliasing errors from the Latte extension. These tasks will help us answer RQ1 about their ability to resolve errors.

[2] https://github.com/liquid-java/liquidjava-examples

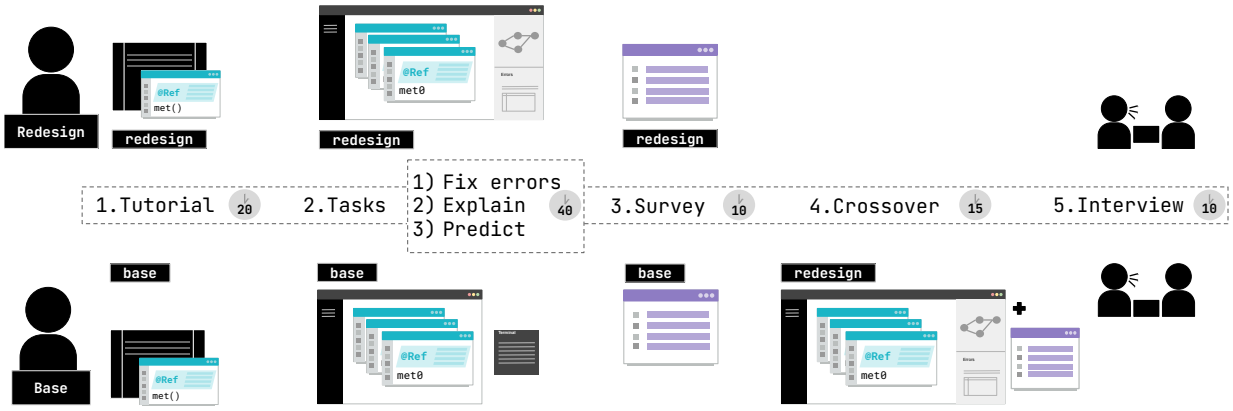


Figure 7.3: Study design

A follow-up exercise will then ask them to use the think-aloud protocol to verbalize their thought process and explain to the interviewer their process for finding and fixing the errors (2) Explain). For a final exercise, we will give them a program with a possible verification error and ask them to predict whether it will verify or not with the current information they have (3) Predict). These last two tasks will help us answer RQ2 about their understanding of verification state and outcomes.

In step 3, we will show both groups a survey to gather their perception on the verification feedback, using open-ended questions and Likert-scale items to assess which aspects they found most valuable for diagnosing and resolving errors, addressing RQ3. Then, for the redesign group, we will conduct a semi-structured interview to gather qualitative feedback on their experience.

For the baseline group, we will then apply the treatment as well by having an extra session to show them the new verification feedback with a task, in order to showcase the improvements we made (4. Crossover). They will be able to directly compare both versions and provide feedback on the differences they noticed, first by answering a survey similar to the previous one, and then by participating in a semi-structured interview.

7.5.1 Anticipated Threats to Validity

Internal validity. Learning effects from the tutorial may influence task performance. We mitigate this by keeping tutorial exercises distinct from measured tasks and using between-subjects comparison.

Construct validity. Our tasks are necessarily smaller and more isolated than production verification scenarios. Effects observed may not transfer to larger codebases with more complex verification failures. However, we will try to mitigate this by including a variety of error types and complexities in our tasks.

External validity. Participants are primarily students, whose experience may differ from professional developers. We mitigate this by screening for Java proficiency and collecting

experience data for subgroup analysis.

Scope limitations. The error types covered depend on implementation progress. We plan to include refinement and typestate errors, with aliasing errors included if the Latte integration (Chapter 6) is completed in time.

7.6 Progress and Expected Contributions

We are currently designing and implementing the features in the LiquidJava VSCode extension,[3] following the design principles outlined in Section 7.3. A Master's student is collaborating on this implementation as part of their thesis project. We are currently planning the user study materials and an IRB submission, with data collection expected until Fall 2026.

The contributions of this work include:

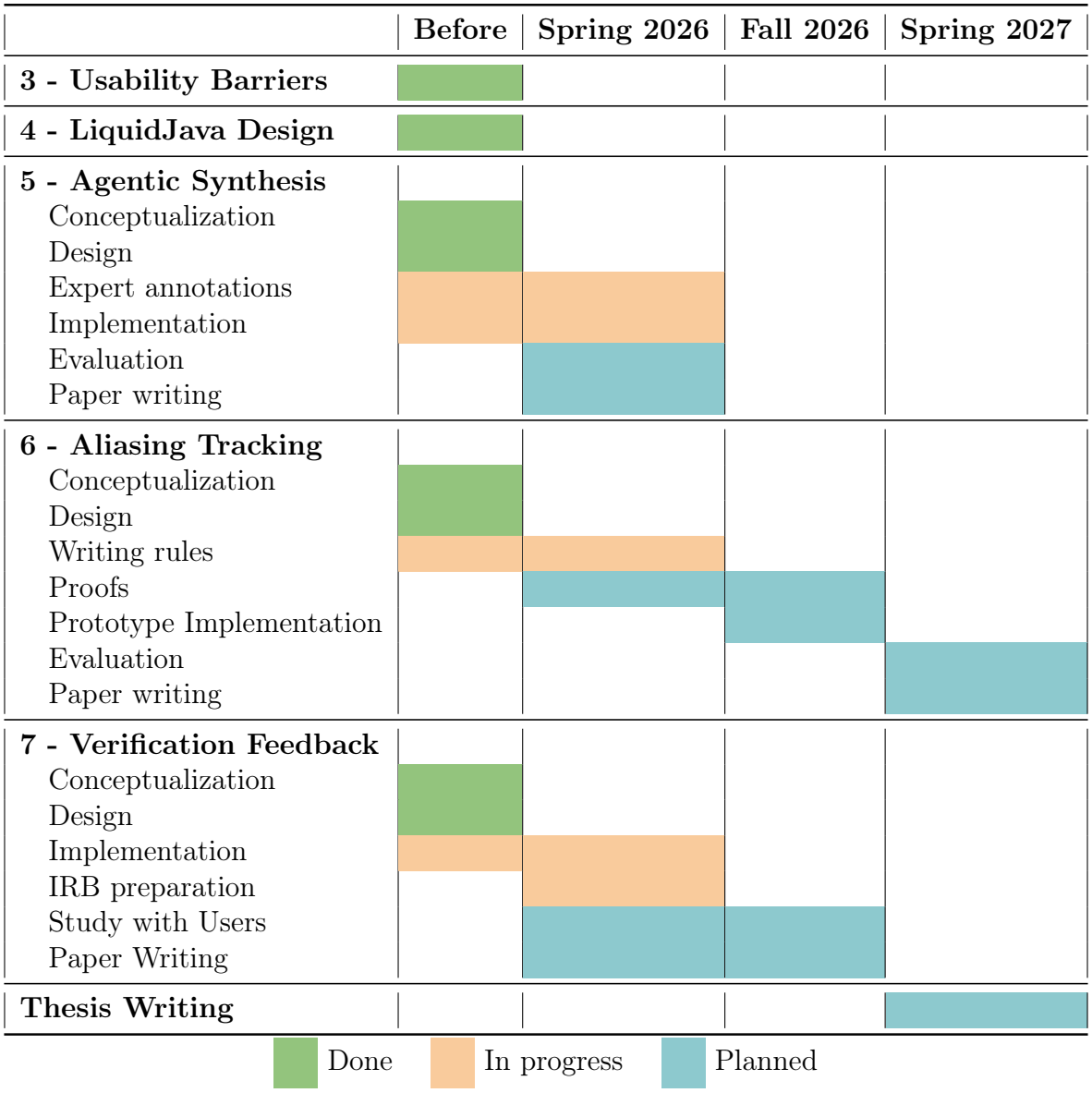
- Six design principles for verification feedback in Liquid Types, grounded in our empirical barrier studies (Chapters 3 and 4) and prior work on error messages for compilers and advanced type systems.
- An open-source implementation of these principles in LiquidJava's VS Code extension, including simplified error presentation, progressive disclosure, and source code mapping.
- A user study evaluating how verification feedback design affects developers' ability to resolve refinement and typestate errors, comparing baseline feedback against our principled redesign.

[3] https://marketplace.visualstudio.com/items?itemName=AlcidesFonseca.liquid-java

8 Timeline

I propose the following timeline for completing the work in this proposal:

Table 8.1: Thesis Timeline



Chapter 3 and Chapter 4 are complete and published, and require no further updates.

Chapter 5 is currently in progress, we are almost finished with the manual annotations and the implementation is mostly complete. This semester, we will finish the implementation and evaluation of the synthesis tool, and write a paper describing the approach and results.

Chapter 6 is in development, we have completed the conceptualization and design of the aliasing tracking mechanism, and are currently writing the formal rules. We plan to finish the rules by the end of this semester, and then write proofs and implement a prototype in the next semester, leaving the evaluation and paper writing for the last semester.

Chapter 7 is currently under implementation, and we are submitting the IRB proposal in the next few weeks. We plan to conduct the user study and write the paper in the next semester.

Finally, we will dedicate the last semester to writing the thesis and preparing the defense.

9 Conclusion

Software verification helps developers catch bugs early in the development process, reducing the cost and effort of fixing errors that would otherwise reach production. Liquid Types offer a promising approach by extending traditional type systems with logical predicates, yet they have not achieved mainstream adoption. To understand why, we conducted empirical studies that identified key usability barriers developers face when using Liquid Types. Building on these insights, we designed LiquidJava, a usability-oriented refinement type system that brings Liquid Types to Java, a mainstream object-oriented language. To reduce the burden of writing specifications, we developed an agentic workflow that automatically synthesizes LiquidJava annotations from class documentation using LLM-based techniques. To verify common Java patterns involving mutable state and object references, we extended the type system with lightweight aliasing tracking. Finally, to improve developers' interaction with the verification process, we redesigned error diagnostics to provide actionable feedback. Together, these contributions support our thesis: by understanding the usability barriers that hinder Liquid Types adoption, we can design a usable, developer-centric refinement type system for Java that, combined with automated synthesis, improved diagnostics, and aliasing tracking, makes Liquid Types expressive and practical for verifying Java programs.

9 Bibliography

[1] Shape of errors to come: Rust blog, 2016. URL https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html. 7.1, 7.2

[2] The rust programming language, 2018. URL https://doc.rust-lang.org/book/ch19-04-advanced-types.html. 7.1

[3] Elm, 2022. URL https://elm-lang.org/. 7.1

[4] Pypl popularity of programming language. http://pypl.github.io/PYPL.html, 2024. 4.1

[5] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The key platform for verification and analysis of java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments - 6th International Conference*, volume 8471 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2014. doi: 10.1007/978-3-319-12154-3_4. 2

[6] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, pages 311–330. ACM, 2002. doi: 10.1145/582419.582448. URL https://doi.org/10.1145/582419.582448. 6.4

[7] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1015–1022. ACM, 2009. doi: 10.1145/1639950.1640073. 4.2.3

[8] American Psychological Association. Wait-list control groups. *APA Dictionary of Psychology*, 2026. https://dictionary.apa.org/wait-list-control-group. 7.5

[9] Pascal W. M. Van Gerven Babu Noushad and Anique B. H. de Bruin. Twelve tips for applying the think-aloud method to capture cognitive processes. *Medical Teacher*, 46(7):892–897, 2024. doi: 10.1080/0142159X.2023.2289847. URL https://doi.org/10.1080/0142159X.2023.2289847. PMID: 38071621. 3.3.1

[10] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark

Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021. doi: 10.1145/3485516. URL https://doi.org/10.1145/3485516. 6.4

[11] Brett A. Becker and Keith Quille. 50 years of CS1 at SIGCSE: A review of the evolution of introductory programming education research. In *Technical Symposium on Computer Science Education*, SIGCSE '19, page 338–344. ACM, 2019. ISBN 9781450358903. doi: 10.1145/3287324.3287432. URL https://doi.org/10.1145/3287324.3287432. 3.4.7

[12] 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. 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*, pages 177–210. ACM, 2019. doi: 10.1145/3344429.3372508. URL https://doi.org/10.1145/3344429.3372508. 3.6, 7.1, **P1**, **P2**, **P6**

[13] Bernhard Beckert and Sarah Grebing. Evaluating the usability of interactive verification systems. In *International Workshop on Comparative Empirical Evaluation of Reasoning Systems*, volume 873, pages 3–17. CEUR-WS.org, 2012. URL https://dblp.org/rec/conf/cade/BeckertG12.bib. 3.6

[14] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2011. doi: 10.1007/978-3-642-22655-7_2. 5.3.1, 5.4.1

[15] Mordechai Ben-Ari. The bug that destroyed a rocket. *ACM SIGCSE Bull.*, 33(2): 58–59, 2001. doi: 10.1145/571922.571958. URL https://doi.org/10.1145/571922.571958. 1

[16] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *ACM Trans. Program. Lang. Syst.*, volume 33, pages 8:1–8:45, 2011. doi: 10.1145/1890028.1890031. URL https://doi.org/10.1145/1890028.1890031. 3.1, 4.1

[17] Lorenzo Bettini. Type errors for the IDE with Xtext and Xsemantics. *Open Computer Science*, 9(1):52–79, 2019. doi: 10.1515/comp-2019-0003. 7.2

[18] Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. Getting into the flow: Towards better type error messages for constraint-based type inference. *Proc. ACM Program. Lang.*, 7(OOPSLA2), 2023. doi: 10.1145/3622812. URL https://doi.org/10.1145/3622812. 3.6

[19] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016. doi: 10.6092/ISSN.1972-5787/4593. URL https://doi.org/10.6092/issn.1972-5787/4593. 3.4.5

[20] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda,a functional language with dependent types. In *International Conference on Theorem Proving in*

Higher Order Logics, pages 73–78. Springer, 2009. 3.5, 7.2

[21] John Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exp.*, 31(6):533–553, 2001. doi: 10.1002/spe.370. URL https://doi.org/10.1002/spe.370. 6.4

[22] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 283–295. ACM, 2005. doi: 10.1145/1040305.1040329. URL https://doi.org/10.1145/1040305.1040329. 6.4

[23] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013. 3.5, 7.2

[24] Chris Brown. Nudging developers to participate in se research, 2022. URL https://api.semanticscholar.org/CorpusID:252539209. 3.3.2

[25] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8):689–694, 1997. ISSN 0167-8655. doi: https://doi.org/10.1016/S0167-8655(97)00060-3. URL https://www.sciencedirect.com/science/article/pii/S0167865597000603. 5.4.4

[26] Nuno Burnay, Antónia Lopes, and Vasco T. Vasconcelos. Statically checking REST API consumers. In Frank S. de Boer and Antonio Cerone, editors, *Software Engineering and Formal Methods*, volume 12310, pages 265–283, 2020. doi: 10.1007/978-3-030-58768-0_15. 4.1

[27] Raymond P. L. Buse, Caitlin Sadowski, and Westley Weimer. Benefits and barriers of user evaluation in software engineering research. In Cristina Videira Lopes and Kathleen Fisher, editors, *26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 643–656. ACM, 2011. doi: 10.1145/2048066.2048117. 4.3

[28] Oscar Callaú, Romain Robbes, Éric Tanter, David Röthlisberger, and Alexandre Bergel. On the use of type predicates in object-oriented software: the case of smalltalk. *SIGPLAN Not.*, 50(2):135–146, oct 2014. ISSN 0362-1340. doi: 10.1145/2775052.2661091. URL https://doi.org/10.1145/2775052.2661091. 3.6

[29] Pedro Carrott, Nuno Saavedra, Kyle Thompson, Sorin Lerner, João F. Ferreira, and Emily First. Coqpilot: Proof navigation in python in the era of LLMs. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14. ACM, 2024. doi: 10.1145/3663529.3663814. 5.2

[30] Elias Castegren and Tobias Wrigstad. Reference capabilities for concurrency control. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 5:1–5:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.ECOOP.2016.5. URL https://doi.org/10.4230/LIPIcs.ECOOP.2016.5. 6.4

[31] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu K. Lahiri, and Nikhil Swamy. Towards neural synthesis for SMT-assisted proof-oriented programming. *arXiv preprint arXiv:2405.01787*, 2024. 5.2, 5.3

[32] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. Exploring regular expression comprehension. In *International Conference on Automated Software Engineering, ASE*, pages 405–416. IEEE Computer Society, 2017. doi: 10.1109/ASE.2017.8115653. URL https://doi.org/10.1109/ASE.2017.8115653. 3.6

[33] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. PL and HCI: better together. *Commun. ACM*, 64(8):98–106, 2021. doi: 10.1145/3469279. URL https://doi.org/10.1145/3469279. 2, 3.6

[34] Sean Chen. The anatomy of error messages in rust — rustfest global 2020, 2020. URL https://www.youtube.com/watch?v=oMskswu1SxM&list=PL85XCvVPmGQiudPknCxiSpybc5RTfkXe6. 7.1

[35] Chanhee Cho, Yi Zhou, Jay Bosamiya, and Bryan Parno. A framework for debugging automated program verification proofs via proof actions. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 348–361. Springer, 2024. doi: 10.1007/978-3-031-65627-9_17. URL https://doi.org/10.1007/978-3-031-65627-9_17. 7.2

[36] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 587–606. ACM, 2012. doi: 10.1145/2384616.2384659. URL https://doi.org/10.1145/2384616.2384659. 1, 3.1, 4.2.1

[37] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013. doi: 10.1007/978-3-642-36946-9_3. URL https://doi.org/10.1007/978-3-642-36946-9_3. 6.4

[38] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*, pages 48–64. ACM, 1998. doi: 10.1145/286936.286947. URL https://doi.org/10.1145/286936.286947. 6.4

[39] Michael Coblenz, Michelle Mazurek, and Michael Hicks. Does the bronze garbage collector make rust easier to use? a controlled experiment. *arXiv preprint arXiv:2110.01098*, 2021. 4.3

[40] Michael Coblenz, April Porter, Varun Das, Teja Nallagorla, and Michael Hicks. A Multimodal Study of Challenges Using Rust, 2023. URL https://kilthub.cmu.edu/articles/conference_contribution/A_Multimodal_Study_of_Challenges_Using_Rust/22277326. 3.6

[41] Michael J. Coblenz, Whitney Nelson, Jonathan Aldrich, Brad A. Myers, and Joshua

Sunshine. Glacier: transitive class immutability for java. In *International Conference on Software Engineering (ICSE)*, pages 496–506. IEEE / ACM, 2017. doi: 10.1109/ICSE.2017.52. URL https://dblp.org/rec/conf/icse/CoblenzNAMS17.bib. 2, 3.6, 4.2, 4.3.5

[42] Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Can advanced type systems be usable? an empirical study of ownership, assets, and typestate in obsidian. *Proc. ACM Program. Lang.*, 4(OOPSLA):132:1–132:28, 2020. doi: 10.1145/3428200. URL https://dblp.org/rec/journals/pacmpl/CoblenzAMS20.bib. 2, 3.6, 4.2, 7.2

[43] Michael J. Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. PLIERS: A process that integrates user-centered methods into programming language design. *ACM Trans. Comput. Hum. Interact.*, 28(4):28:1–28:53, 2021. doi: 10.1145/3452379. URL https://dblp.org/rec/journals/tochi/CoblenzKKWBSAM21.bib. 2, 3.6, 4.2, 4.3

[44] David R. Cok. Openjml: Software verification for java 7 using jml, openjdk, and eclipse. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment*, volume 149 of *EPTCS*, pages 79–92, 2014. doi: 10.4204/EPTCS.149.8. 2

[45] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. A grounded conceptual model for ownership types in rust. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi: 10.1145/3622841. URL https://doi.org/10.1145/3622841. 3.6, 7.2

[46] Critical Software. The role of software in medical device failures, 2018. URL https://www.criticalsoftware.com/multimedia/critical/pt/xvhdnSb3o-CSW_White_Paper_The_Role_of_Software_in_Medical_Device_Failures.pdf. 1

[47] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *J. Object Technol.*, 5(5):59–85, 2006. doi: 10.5381/JOT.2006.5.5.A3. URL https://doi.org/10.5381/jot.2006.5.5.a3. 3.5

[48] Matthew C. Davis, Emad Aghayi, Thomas D. LaToza, Xiaoyin Wang, Brad A. Myers, and Joshua Sunshine. What's (not) working in programmer user studies? *ACM Trans. Softw. Eng. Methodol.*, 32(5):120:1–120:32, 2023. doi: 10.1145/3587157. URL https://doi.org/10.1145/3587157. 3.6

[49] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24. 3.2

[50] Uri Dekel and James D. Herbsleb. Improving api documentation usability with knowledge pushing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 320–330, 2009. doi: 10.1109/ICSE.2009.5070532. 5.1

[51] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C. Albrecht, and Garrett B. Powell. On designing programming error messages for novices: Readability and its constituent factors. In Yoshifumi Kitamura,

Aaron Quigley, Katherine Isbister, Takeo Igarashi, Pernille Bjørn, and Steven Mark Drucker, editors, *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*, pages 55:1–55:15. ACM, 2021. doi: 10.1145/3411764.3445696. URL https://doi.org/10.1145/3411764.3445696. 7.1, 7.2

[52] Alan Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction (3rd Edition)*. Prentice-Hall, Inc., 2003. ISBN 0130461091. 4.2

[53] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, 48(3):835–847, 2022. doi: 10.1109/TSE.2020.3004525. 4.2.1, **R.3**

[54] Tao Dong and Kandarp Khandwala. The Impact of "Cosmetic" Changes on the Usability of Error Messages. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–6, 2019. ISBN 9781450359719. doi: 10.1145/3290607.3312978. 7.2

[55] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chetan Murthy, Catherin Parent, Christine Paulin-Mohring, and Benjamin Werner. *The COQ Proof Assistant: User's Guide: Version 5.6*. INRIA, 1992. 3.5

[56] Brian Ellis, Jeffrey Stylos, and Brad A. Myers. The factory pattern in API design: A usability evaluation. In *29th International Conference on Software Engineering*, pages 302–312. IEEE Computer Society, 2007. doi: 10.1109/ICSE.2007.85. 4.3

[57] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1–12. ACM, 2023. 5.1, 5.2

[58] Joseph Eremondi, Wouter Swierstra, and Jurriaan Hage. A framework for improving error messages in dependently-typed languages. *Open Comput. Sci.*, 9(1):1–32, 2019. doi: 10.1515/comp-2019-0001. 3.5, 7.2

[59] K. Anders Ericsson and Herbert A. Simon. *Protocol Analysis: Verbal Reports as Data*. The MIT Press, 04 1993. ISBN 9780262272391. doi: 10.7551/mitpress/5657.001.0001. URL https://doi.org/10.7551/mitpress/5657.001.0001. 3.3.1

[60] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007. doi: 10.1016/j.scico.2007.01.015. 5.2

[61] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Ron Crocker and Guy L. Steele Jr., editors, *2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003*, pages 302–312. ACM, 2003. doi: 10.1145/949305.949332. 4.2.1

[62] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *Programming Languages and Systems - 22nd European Symposium on Programming ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013. doi: 10.1007/978-3-642-37036-6_8. URL https://doi.org/10.1007/978-3-642-37036-6_8. 3.5

[63] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001. doi: 10.1007/3-540-45251-6_29. 5.2

[64] Alcides Fonseca, Paulo Santos, and Sara Silva. The usability argument for refinement typed genetic programming. In *Parallel Problem Solving from Nature - PPSN*, volume 12270 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2020. doi: 10.1007/978-3-030-58115-2_2. URL https://doi.org/10.1007/978-3-030-58115-2_2. 3.5

[65] Manuel J. Fonseca, Pedro Campos, and Daniel Gonçalves. *Introdução ao Design de Interfaces*. 10 2012. 4.2

[66] Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277. ACM, 1991. doi: 10.1145/113445.113468. 2

[67] Shuai Fu, Tim Dwyer, Peter J. Stuckey, Jackson Wain, and Jesse Linossier. ChameleonIDE: Untangling type errors through interactive visualization and exploration. In *International Conference on Program Comprehension, ICPC*, pages 146–156. IEEE, 2023. doi: 10.1109/ICPC58990.2023.00029. URL https://doi.org/10.1109/ICPC58990.2023.00029. 3.6, 7.2

[68] Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. Usability-oriented design of liquid types for java. In *International Conference on Software Engineering (ICSE)*, pages 1520–1532. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00132. URL https://doi.org/10.1109/ICSE48619.2023.00132. 3.1, 3.6, 4, 5.3

[69] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. Usability barriers for liquid types. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi: 10.1145/3729327. URL https://doi.org/10.1145/3729327. 3

[70] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. Artifact for "usability barriers for liquid types". Zenodo repository, 2025. URL https://doi.org/10.5281/zenodo.15044759. Artifact containing research materials for the qualitative study on developer experiences with LiquidHaskell. 3.3

[71] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Anal. Appl.*, 13(1):113–129, February 2010. ISSN 1433-7541. 5.4.4

[72] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 Expert Survey on Formal Methods. In *Formal Methods for Industrial Critical Systems, FMICS*, page 3–69. Springer-Verlag, 2020. ISBN 978-3-030-58297-5. doi: 10.1007/978-3-030-58298-2_1. URL https://doi.org/10.1007/978-3-030-58298-2_1. 3.6

[73] Jad Elkhaleq Ghalayini and Neel Krishnaswami. Explicit refinement types. *Proc. ACM Program. Lang.*, 7(ICFP):187–214, 2023. doi: 10.1145/3607837. URL https://doi.org/10.1145/3607837. 3.5

[74] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-based testing in practice. In *International Conference on Software Engineering (ICSE)*, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639581. URL https://doi.org/10.1145/3597503.3639581. 3.5

[75] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Rely-guarantee references for refinement types over aliased mutable data. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 73–84. ACM, 2013. doi: 10.1145/2491956.2462160. URL https://doi.org/10.1145/2491956.2462160. 6.4

[76] Gavin Gray, Will Crichton, and Shriram Krishnamurthi. An interactive debugger for rust trait errors. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi: 10.1145/3729302. URL https://doi.org/10.1145/3729302. 7.2, **P2**, **P3**

[77] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Vis. Lang. Comput.*, 7(2): 131–174, 1996. doi: 10.1006/JVLC.1996.0009. URL https://doi.org/10.1006/jvlc.1996.0009. 3.6

[78] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empir. Softw. Eng.*, 19(5):1335–1382, 2014. doi: 10.1007/s10664-013-9289-1. 4.3

[79] Felienne Hermans. Hedy: A gradual language for programming education. In Anthony V. Robins, Adon Moskal, Amy J. Ko, and Renée McCauley, editors, *ICER 2020: International Computing Education Research Conference*, pages 259–270. ACM, 2020. doi: 10.1145/3372782.3406262. 4.2

[80] Karen Holtzblatt and Hugh Beyer. *Contextual Design, Second Edition: Design for Life*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2016. ISBN 0128008946. 3.3.1

[81] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3): 396–450, 2001. doi: 10.1145/503502.503505. URL https://doi.org/10.1145/503502.503505. 6.1

[82] Ranjit Jhala and Niki Vazou. Refinement Types: A Tutorial. *Found. Trends Program. Lang.*, 6(3-4):159–317, 2021. doi: 10.1561/2500000032. URL https://doi.org/10.1561/2500000032. 2, 3.2, 3.2, 3.4.6, 4.1, 4.2.1

[83] Sára Juhošová, Andy Zaidman, and Jesper Cockx. Pinpointing the learning obstacles of an interactive theorem prover. In *2025 IEEE/ACM 33rd International Conference*

on Program Comprehension (ICPC), pages 159–170, 2025. doi: 10.1109/ICPC66645.2025.00024. 3.6, 7.2

[84] Kazerounian M. , Vazou N. , Bourgerie A. , Foster J., , Torlak E. Refinement Types for Ruby. https://nikivazou.github.io/static/VMCAI18/paper.pdf. 2

[85] David Holmes Ken Arnold, James Gosling. *THE Java™ Programming Language, Fourth Edition*. Addison Wesley Professional, 2005. ISBN 0-321-34980-6. 4.2.3

[86] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empir. Softw. Eng.*, 20(1):110–141, 2015. doi: 10.1007/s10664-013-9279-3. 4.3

[87] Nikolai Kosmatov and Julien Signoles. Frama-C, A collaborative framework for C code verification: Tutorial synopsis. In *Runtime Verification (RV)*, volume 10012 of *Lecture Notes in Computer Science*, pages 92–115. Springer, 2016. doi: 10.1007/978-3-319-46982-9_7. URL https://doi.org/10.1007/978-3-319-46982-9_7. 3.5

[88] Charlie Koster. Advanced types in elm - opaque types, Sep 2017. URL https://ckoster22.medium.com/advanced-types-in-elm-opaque-types-ec5ec3b84ed2. 7.1, 7.2

[89] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for java. *Sci. Comput. Program.*, 155:52–75, 2018. doi: 10.1016/j.scico.2017.10.006. URL https://doi.org/10.1016/j.scico.2017.10.006. 2

[90] Herb Krasner. The cost of poor software quality in the us: A 2022 report. Technical report, Consortium for Information & Software Quality (CISQ), December 2022. URL https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf. Technical report. 1

[91] Lean for VS Code. Lean for vs code, 2021. URL https://github.com/leanprover/vscode-lean. 7.2, **P4**

[92] Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420. Citeseer, 1998. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=397cb3c2ad6569aef081c282671d17937df483d0. 2, 3.5, 4.4

[93] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. STORM: Refinement Types for Secure Web Applications. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 441–459. USENIX Association, 2021. URL https://www.usenix.org/conference/osdi21/presentation/lehmann. 3.1

[94] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.*, 7(PLDI):1533–1557, 2023. doi: 10.1145/3591283. URL https://doi.org/10.1145/3591283. 1, 2, 3.1, 6.4

[95] Nico Lehmann, Cole Kurashige, Nikhil Akiti, Niroop Krishnakumar, and Ranjit Jhala. Generic Refinement Types. *Proc. ACM Program. Lang.*, 9(POPL):1446–1474, 2025. doi: 10.1145/3704885. URL https://doi.org/10.1145/3704885. 3.1

[96] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20. URL https://doi.org/10.1007/978-3-642-17511-4_20. 3.5

[97] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Neurosymbolic modular refinement type inference. In *Proceedings of the 2025 International Conference on Software Engineering*, pages 280–291. IEEE, 2025. 5.2

[98] Justin Lubin and Sarah E. Chasins. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi: 10.1145/3485532. URL https://doi.org/10.1145/3485532. 3.6, 4.3

[99] Jane Lusby. Rustconf 2020 - error handling isn't all about errors, Aug 2020. URL https://www.youtube.com/watch?v=rAF8mLI0naQ. 7.1

[100] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. SpecGen: Automated generation of formal program specifications via large language models. In *Proceedings of the 2025 International Conference on Software Engineering*. IEEE, 2025. 5.2, 5.3

[101] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind your language: on novices' interactions with error messages. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, page 3–18. ACM, 2011. ISBN 9781450309417. doi: 10.1145/2048237.2048241. URL https://doi.org/10.1145/2048237.2048241. 3.5

[102] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In Gary T. Leavens and Matthew B. Dwyer, editors, *27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 683–702. ACM, 2012. doi: 10.1145/2384616.2384666. 4.3

[103] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992. doi: 10.1109/2.161279. URL https://doi.org/10.1109/2.161279. 2

[104] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA*, pages 1–18. ACM, 2013. doi: 10.1145/2509136.2509515. URL https://doi.org/10.1145/2509136.2509515. 3.5

[105] Microsoft. Language server protocol, 2022. URL https://microsoft.github.io/language-server-protocol/. 4.2.4

[106] Ran Mo, Dongyu Wang, Wenjing Zhan, Yingjie Jiang, Yepeng Wang, Yuqi Zhao, Zengyang Li, and Yutao Ma. Assessing and analyzing the correctness of github copilot's code suggestions. *ACM Trans. Softw. Eng. Methodol.*, 34(7), August 2025. ISSN 1049-331X. doi: 10.1145/3715108. URL https://doi.org/10.1145/3715108. 1

[107] Rebecca Morelle. Beresheet spacecraft: 'technical glitch' led to moon crash. https:

//www.bbc.com/news/science-environment-47914100, Apr 2019. URL https://www.bbc.com/news/science-environment-47914100. 1

[108] Gunnar Morling. Jakarta bean validation specification., 2019. URL https://jakarta.ee/specifications/bean-validation/2.0/bean-validation_2.0.pdf. 4.2.1

[109] João Mota, Marco Giunti, and António Ravara. Java typestate checker. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2021. doi: 10.1007/978-3-030-78142-2_8. URL https://doi.org/10.1007/978-3-030-78142-2_8. 2

[110] Eric Mugnier, Yuanyuan Zhou, Ranjit Jhala, and Michael Coblenz. On the impact of formal verification on software development. *Proc. ACM Program. Lang.*, 9 (OOPSLA2), October 2025. doi: 10.1145/3763181. URL https://doi.org/10.1145/3763181. 2

[111] Eric Mugnier, Yuanyuan Zhou, Ranjit Jhala, and Michael Coblenz. On the impact of formal verification on software development. *Proc. ACM Program. Lang.*, 9 (OOPSLA2), October 2025. doi: 10.1145/3763181. URL https://doi.org/10.1145/3763181. 7.2, **P4**

[112] Prithwish Mukherjee and Benjamin Delaware. AutoVerus: Automated proof generation for rust code. In *Proceedings of the 2024 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2024. 5.2, 5.3

[113] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. Programmers are users too: Human-centered methods for improving programming tools. *Computer*, 49(7):44–52, 2016. doi: 10.1109/MC.2016.200. URL https://doi.org/10.1109/MC.2016.200. 2, 3.6

[114] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in rosetta code. In *International Conference on Software Engineering, ICSE*, pages 778–788. IEEE Computer Society, 2015. doi: 10.1109/ICSE.2015.90. URL https://doi.org/10.1109/ICSE.2015.90. 3.6

[115] Chris Okasaki. Simple and efficient purely functional queues and deques. *J. Funct. Program.*, 5(4):583–592, 1995. doi: 10.1017/S0956796800001489. URL https://doi.org/10.1017/S0956796800001489. 3.3.1

[116] Francisco Oliveira, Alexandra Mendes, and Carolina Carreira. What challenges do developers face when using verification-aware programming languages? In *36th IEEE International Symposium on Software Reliability Engineering, ISSRE 2025, São Paulo, Brazil, October 21-24, 2025*, pages 203–214. IEEE, 2025. doi: 10.1109/ISSRE66568.2025.00031. URL https://doi.org/10.1109/ISSRE66568.2025.00031. 2

[117] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In Barbara G. Ryder and Andreas Zeller, editors, *ACM/SIGSOFT International Symposium on Software Testing*

and Analysis, pages 201–212. ACM, 2008. doi: 10.1145/1390630.1390656. 4.2.1

[118] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. SPOON: A library for implementing analyses and transformations of java source code. *Softw. Pract. Exp.*, 46(9):1155–1179, 2016. doi: 10.1002/spe.2346. 4.2.1

[119] Bartosz Piotrowski, Ramon Fernández Mir, and Edward W. Ayers. Machine-learned premise selection for lean. In *Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2023*, volume 14278 of *Lecture Notes in Computer Science*, pages 175–186. Springer, 2023. doi: 10.1007/978-3-031-43513-3_10. URL https://doi.org/10.1007/978-3-031-43513-3_10. 3.5

[120] Lorraine E. Prokop. Software error incident categorizations in aerospace. *J. Aerosp. Inf. Syst.*, 21(10):775–789, 2024. doi: 10.2514/1.I011240. URL https://doi.org/10.2514/1.i011240. 1

[121] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in f*. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi: 10.1145/3110261. URL https://doi.org/10.1145/3110261. 6.4

[122] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. Cn: Verifying systems c code with separation-logic refinement types. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi: 10.1145/3571194. URL https://doi.org/10.1145/3571194. 6.4

[123] Mary Elizabeth Raven and Alicia Flanders. Using contextual inquiry to learn about your audiences. *SIGDOC Asterisk J. Comput. Doc.*, 20(1):1–13, feb 1996. ISSN 0731-1001. doi: 10.1145/227614.227615. URL https://doi.org/10.1145/227614.227615. 3.3.1

[124] Cedric Richter and Heike Wehrheim. Beyond postconditions: Can large language models infer formal contracts for automatic software verification? *arXiv preprint arXiv:2510.12702*, 2025. 5.1, 5.2

[125] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI)*, pages 159–169. ACM, 2008. doi: 10.1145/1375581.1375602. URL https://doi.org/10.1145/1375581.1375602. 1, 2, 3.1, 4.1

[126] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 131–144, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605584799. doi: 10.1145/1706299.1706316. URL https://doi.org/10.1145/1706299.1706316. 6.4

[127] Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. CSolve: Verifying C with Liquid Types. In *Computer Aided Verification (CAV)*, volume 7358 of *Lecture Notes in Computer Science*, pages 744–750. Springer, 2012. doi: 10.1007/978-3-642-31424-7_59. URL https://doi.org/10.1007/978-3-642-31424-7_59. 3.1

[128] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4): 58–66, 2018. doi: 10.1145/3188720. 4.2.1, **R.3**

[129] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009. ISBN 978-1-84787-548-8. 3.3.3, 4.3.3

[130] Georg Stefan Schmid and Viktor Kuncak. Smt-based checking of predicate-qualified types for scala. In Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche, editors, *7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH*, pages 31–40. ACM, 2016. doi: 10.1145/2998392.2998398. 2

[131] Benno Stein, Lazaro Clapp, Manu Sridharan, and Bor-Yuh Evan Chang. Safe stream-based programming with refinement types. In *33rd ACM/IEEE International Conference on Automated Software Engineering*, page 565–576, 2018. ISBN 9781450359375. 2

[132] Jeffrey Stylos and Brad A. Myers. The implications of method placement on API learnability. In Mary Jean Harrold and Gail C. Murphy, editors, *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 105–112. ACM, 2008. doi: 10.1145/1453101.1453117. 4.3

[133] Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark Barrett, David Dill, Xiaokang Qiu, and Shuvendu K. Lahiri. ClassInvGen: Class invariant synthesis using large language models. *arXiv preprint arXiv:2502.18917*, 2025. 5.2

[134] SWE-bench Team. Swe-bench. https://www.swebench.com/, 2024. Accessed: November 7, 2025. 5.3.4

[135] TBD. LEGO-Prover: Neural theorem proving with growing libraries, TBD. Publication details to be confirmed. 5.3

[136] TBD. Inferring state machine from the protocol implementation via large language model, TBD. Publication details to be confirmed. 5.2

[137] Paper-Proof Team. Paperproof: A new proof interface for lean 4. https://github.com/Paper-Proof/paperproof, 2025. Accessed: 2026-01-19. 7.2, **P4**

[138] Kyle Thayer, Sarah E. Chasins, and Amy J. Ko. A theory of robust API knowledge. *ACM Trans. Comput. Educ.*, 21(1):8:1–8:32, 2021. doi: 10.1145/3444945. URL https://doi.org/10.1145/3444945. 4.3

[139] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. Consort: Context- and flow-sensitive ownership refinement types for imperative programs. In *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*, page 684–714, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-44913-1. doi: 10.1007/978-3-030-44914-8_25. URL https://doi.org/10.1007/978-3-030-44914-8_25. 6.4

[140] V. Javier Traver. On compiler error messages: What they *Say* and what they *Mean*.

Adv. Hum. Comput. Interact., 2010:602570:1–602570:26, 2010. doi: 10.1155/2010/602570. URL https://doi.org/10.1155/2010/602570. 3.5

[141] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study on the impact of C++ lambdas and programmer experience. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22*, pages 760–771. ACM, 2016. doi: 10.1145/2884781.2884849. 4.3

[142] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: experience with refinement types in the real world. In *ACM SIGPLAN symposium on Haskell*, pages 39–51. ACM, 2014. doi: 10.1145/2633357.2633366. URL https://doi.org/10.1145/2633357.2633366. 1, 2, 3.1, 6.4

[143] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for TypeScript. In Chandra Krintz and Emery Berger, editors, *37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, pages 310–325. ACM, 2016. doi: 10.1145/2908080.2908110. 2

[144] Philip Wadler. Linear types can change the world! In Manfred Broy and Cliff B. Jones, editors, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990. 6.4

[145] Robin Webbers, Klaus von Gleissenthall, and Ranjit Jhala. Refinement type refutations. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi: 10.1145/3689745. URL https://doi.org/10.1145/3689745. 3.5, 7.2, **P4**, **P6**

[146] Cheng Wen, Jian Cao, Jia Su, Zhi Xu, Shengchao Qin, Ming He, Hongyu Li, Shing-Chi Cheung, and Chao Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *Computer Aided Verification: 36th International Conference, CAV 2024*, volume 14228 of *Lecture Notes in Computer Science*, pages 302–328. Springer, 2024. doi: 10.1007/978-3-031-65630-9_16. 5.2, 5.3

[147] Christopher D. Wickens, John Lee, Yili D. Liu, and Sallie Gordon-Becker. *Introduction to Human Factors Engineering (2nd Edition)*. Prentice-Hall, Inc., USA, 2003. ISBN 0131837362. 3.3.1, 3.3.1

[148] Hyrum Wright, Titus Delafayette Winters, and Tom Manshreck. *Software Engineering at Google*. O'Reilly Media, Inc., 2020. ISBN 9781492082798. 1

[149] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 249–257. ACM, 1998. doi: 10.1145/277650.277732. 4.1

[150] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, Haoye Tian, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F. Bissyandé. Where were the repair ingredients for defects4j bugs? *Empir. Softw. Eng.*, 26(6):122, 2021. doi: 10.1007/S10664-021-10003-7. URL https://doi.org/10.1007/s10664-021-10003-7. 3.5

[151] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. Assessing the quality of github copilot's

code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2022, page 62–71, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450398602. doi: 10.1145/3558489.3559072. URL https://doi.org/10.1145/3558489.3559072. 1

[152] Miao Zhang, Runhan Feng, Hongbo Tang, Yu Zhao, Jie Yang, Hang Qiu, and Qi Liu. Automated extraction of protocol state machines from 3GPP specifications with domain-informed prompts and LLM ensembles. *arXiv preprint arXiv:2510.14348*, 2025. 5.2

[153] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. Total type error localization and recovery with holes. *Proc. ACM Program. Lang.*, 8 (POPL), January 2024. doi: 10.1145/3632910. URL https://doi.org/10.1145/3632910. 3.5

[154] Jianguo Zhao, Yuqiang Sun, Cheng Huang, Chengwei Liu, YaoHui Guan, Yutong Zeng, and Yang Liu. Towards Secure Code Generation With LLMs: A Study on Common Weakness Enumeration . *IEEE Transactions on Software Engineering*, 51 (12):3507–3523, December 2025. ISSN 1939-3520. doi: 10.1109/TSE.2025.3619281. URL https://doi.ieeecomputersociety.org/10.1109/TSE.2025.3619281. 1

[155] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. Learning and programming challenges of rust: a mixed-methods study. In *International Conference on Software Engineering, ICSE*, page 1269–1281. ACM, 2022. ISBN 9781450392211. doi: 10.1145/3510003.3510164. URL https://doi.org/10.1145/3510003.3510164. 3.6

[156] Conrad Zimmerman, Catarina Gamboa, Alcides Fonseca, and Jonathan Aldrich. Latte: Lightweight aliasing tracking for java, 2023. URL https://arxiv.org/abs/2309.05637. 6