Usability Barriers for Liquid Types

Catarina Gamboa[1,2], Abigail Reese[2], Alcides Fonseca[1], Jonathan Aldrich[2]

[1] LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
[2] School of Computer Science, Carnegie Mellon University, USA

LASIGE reliable software systems

Liquid Types have been introduced in 2008, have been implemented across many languages (Haskell, Java, C, JavaScript, Rust) to statically detect bugs ranging from simple errors to security issues and protocol violations.
They add logical predicates to the type system, e.g.:

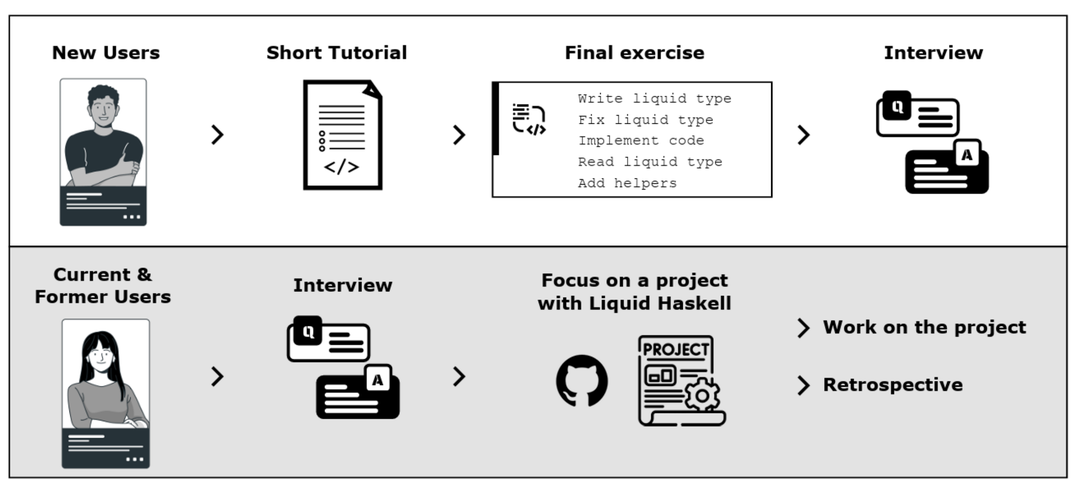
```
{-@ abs :: Int -> {v: Int | v >= 0} @-}   <- Liquid Type
abs :: Int -> Int                         <- Implementation
abs n = if n > 0 then n else (-n)
```

These guarantees can be checked before your program runs, and they have been used to find bugs in MVC frameworks, object protocols, and database queries.
However, they have yet to gain widespread adoption in mainstream programming.

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

Developed a study focused on: LiquidHaskell

Sign-up Form



Verification Challenges

- Unclear divide between Haskell and LiquidHaskell — 11 New, 5 Exp
- Confusing verification features — 11 New, 4 Exp
- Unfamiliarity with Proof Engineering — 7 New, 5 Exp

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

Example from the tutorial for New Users

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

Think about

Where are `f` and `b` in the code?

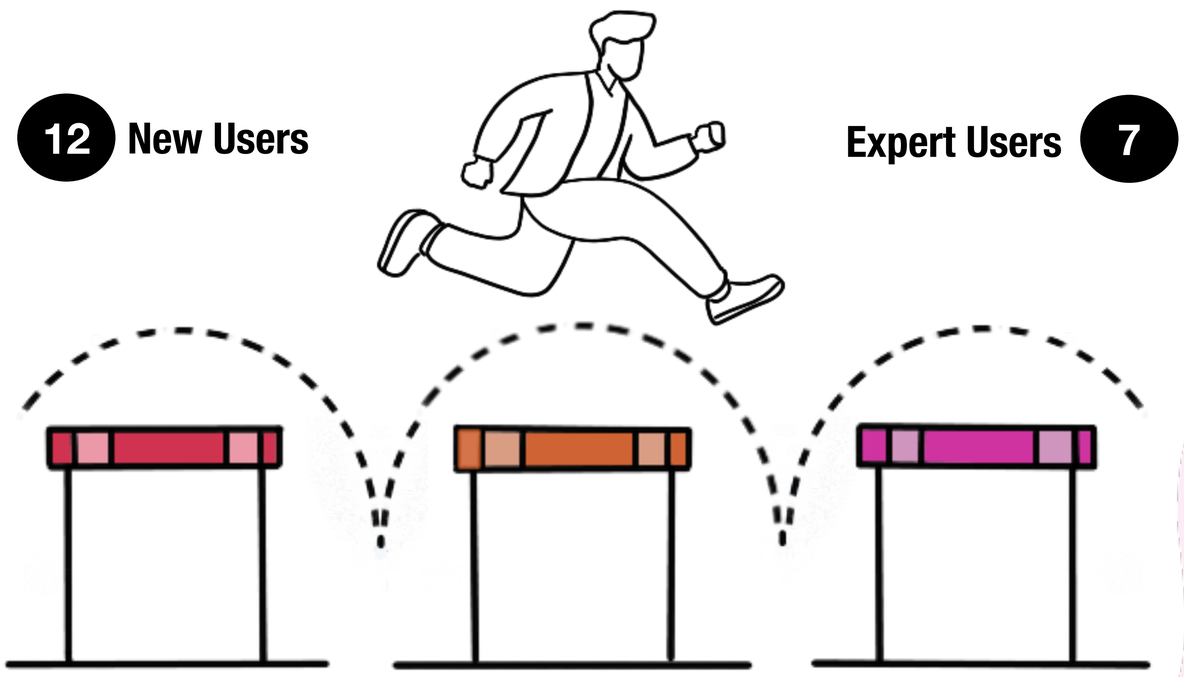
Why don't we need to check for an empty `back`?

Is this an idiomatic way to write code?

How do you create these requirements for your code?

What can you write in the specifications?

> *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. (NC6)*



Scalability Challenges

- Limitations of Automation and Manual Proof Flexibility — 7 Exp
- Scalability and Solver limitations — 6 Exp

5+ hours to verify the full project (FM4)

When automation fails, what do you need to add, and how do you get to a point where it works?

The solver itself has limitations:
- Function translation;
- Bugs…

Developer Experience Challenges

- Unhelpful Error Messages — 9 New, 4 Exp
- Limited IDE Support — 11 New, 4 Exp
- Insufficient learning and reference resources — 7 New, 5 Exp
- Complex Installation and Setup — 7 New, 5 Exp

> *I tried to install Liquid Haskell (...) 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)*

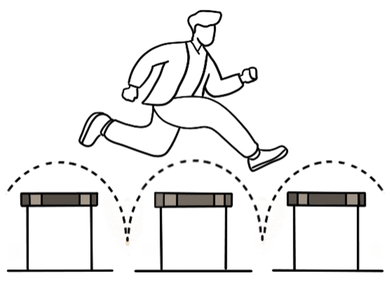
Example from the tutorial for New Users

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

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

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

{-@ example2Q :: QueueN _ 2 @-}
example2Q = Q (1 `cons` (2 `cons` nil)) nil
```

- Error indirection & code mismatch
- No syntax highlighting
- Using comments feels "fake"

Error message

```
Error: Illegal type specification for
Tutorial_09_Case_Study_Lazy_Queues.emp
  Tutorial_09_Case_Study_Lazy_Queues.emp :: forall a .
  {VV : (Tutorial_09_Case_Study_Lazy_Queues.Queue a) | qsize == 0}

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



Are these barriers unique to LiquidHaskell?

In Liquid Types: LiquidJava, Flux

In other verification tools: JML, Dafny, Agda, KeY, Frama-C, Why3

Expert survey on formal methods in 2020

Together, we can cross these barriers and empower developers to build better quality software.

Read our Paper
Accepted at **PLDI '25**
In Seoul, South Korea, in June 2025

Acknowledgements: This work is supported by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under the fellowship PRT/BD/154254/2021, the LASIGE Research Unit, ref. UID/00408/2025, and the RAP project (https://doi.org/10.54499/EXPL/CCI-COM/1306/2021)