CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Enhancing Proof Development in Liquid-Haskell: Implementation and Evaluation of Typed Holes

Master's thesis in Computer Science - Algorithms, Languages, and Logic

MATHEUS DE SOUSA BERNARDO

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Master's thesis 2025

Enhancing Proof Development in LiquidHaskell: Implementation and Evaluation of Typed Holes

MATHEUS DE SOUSA BERNARDO

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Enhancing Proof Development in LiquidHaskell: Implementation and Evaluation of Typed Holes
MATHEUS DE SOUSA BERNARDO

Supervisor: András Kovács, Computer Science and Engineering
Advisor: Facundo Domínguez, Modus Create
Examiner: Carl-Johan Seger, Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Enhancing Proof Development in LiquidHaskell: Implementation and Evaluation of Typed Holes

MATHEUS DE SOUSA BERNARDO
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Refinement type systems provide mechanisms for specifying and verifying programs beyond what mainstream type systems can express. LiquidHaskell extends Haskell with refinement types, and beyond that, has evolved to be used as a theorem prover, akin to Agda or Idris. Unfortunately, it lacks a fundamental feature present in most proof assistants: the ability to inspect goals during proof development, for example, with typed holes. In this thesis, I implement typed hole support for LiquidHaskell addressing this limitation and taking an initial step towards more interactivity in LiquidHaskell.

Keywords: Refinement types, LiquidHaskell, typed holes, proof assistants, interactive development, type-driven development, program verification.

Acknowledgements

First of all, I would like to thank my advisor András Kovács, for all the feedback and suggestions. Also very importantly, I would like to thank my mentor Facundo Domínguez and Tweag for allowing him to give some of his time and help me understand LiquidHaskell.

Matheus De Sousa Bernardo, Gothenburg, 2025-06-24

Contents

List of Figures

List of Tables

1

Introduction

Type systems provide a way for programmers to guarantee some correctness of their code by restricting the set of values allowed in various operations. Furthermore, these restrictions can be checked at compile-time, eliminating a number of runtime errors. However, the type systems of mainstream languages still allow for other classes of errors to occur, for example, division by zero and buffer overflows.

Refinement types allow for the specification and verification of semantic properties of programs, like the previous mentioned ones. This is accomplished by augmenting the type with a logical predicate that effectively restricts the allowed values of that type [1, 2].

For example, we can encode a precondition on a division function ensuring at compile time that division by zero does not happen.

```
div :: Int -> {v : Int | v /= 0} -> Int
```

A known implementation of a refinement type system is LiquidHaskell [3]. As the name suggests, it was designed for the Haskell programming language. With further development, LiquidHaskell received support for proof writing, allowing the specification and verification of arbitrary properties. This feature has been used in the verification of programs in a variety of domains [4, 5, 6].

1.1 Problem Statement

Despite advances, a lack of interactivity was observed when doing large-scale proofs, which reduced the productivity of users given the slower feedback loop [5, 6].

Furthermore, a basic feature of every proof assistant is the ability to inspect a goal/hole type. Currently, LiquidHaskell does not support this seemingly basic functionality. The status quo is that when writing proofs, if you insert a hole, the error message contains only Haskell information and does not include any information from refinement types.

As an example of this gap between proof assistants and LiquidHaskell, consider a basic proposition, $x + 0 = x$. To construct this proof in Agda [7], a well-known proof assistant and programming language, we can use a feature called holes (represented as a question mark). Agda reports the hole goal `Goal: suc (x + 0) == suc x`.

```
+-identity : (x : Nat) -> x + 0 == x
+-identity zero = refl
+-identity (suc x) = ?

+-identity : (x : Nat) -> x + 0 == x
+-identity zero = refl
+-identity (suc x) =
    begin
        suc (x + 0) ==<cong suc (+-identity x) >
        suc x       .
```

Figure 1.1: Proof of $x + 0 = x$ in Agda.

With help from editor shortcuts, we can easily continue the proof. In Figure 1.1, we show the initial step with the hole and the final proof.

Currently, LiquidHaskell does not support inspecting intermediate steps of a proof, and interactivity is quite limited.

The problem this thesis addresses is: **"How to create a more interactive experience when doing proof-development in LiquidHaskell?"**.

As a sub-problem, the thesis focuses on: **"Can typed holes be integrated into LiquidHaskell and provide a foundational step toward interactivity?"**

1.2 Contributions

This thesis makes the following contributions.

- An extension to LiquidHaskell that allows users to insert holes in programs and have a more interactive experience when doing proof development.
- A comparison between this extension and well-known proof assistants and programming languages that also implement holes.

1.3 Motivating Example

The following is an example to demonstrate how the proposed extension improves the LiquidHaskell proof development experience.

```
{-@ measure sum1 :: Int -> Int @-}
{-@ sum1 :: x:Int -> {v:Int | v == x + 1} @-}
sum1 :: Int -> Int
sum1 x = x + 1

{-@ foo :: {v : Int | v == sum1 2} @-}
foo :: Int
foo = _
```

Without the extension, the user only gets the default GHC output for typed holes, nothing is reported from LiquidHaskell.

```
typed-holes/Example0.hs:11:11: error: [GHC-88464]
- Found hole: _ :: Int
-  In an equation for 'foo': foo = _
- Relevant bindings include
...
```

With our extension enabled (`--warn-on-term-holes`), and some minor modification (substituting the underscore for 'hole'). It outputs a message that contains information about the hole and relevant constraints.

```
**** LIQUID: UNSAFE ************************************
typed-holes/Example0.hs:15:5: error:
    Hole Found
Example0.hole :: {v : GHC.Types.Int | v == Example0.sum1 2}
in the context
  Example0.hole : forall a . a
   |
15 |     foo = hole
   |     ^^^^^^^^^
```

2

Background

This chapter provides the necessary background for the reader to understand what refinement types and LiquidHaskell are. It also gives the necessary theory to understand the implementation of the thesis.

2.1 Introduction to Haskell and Type Systems

Haskell is a functional programming language with a strong type system [8]. We are going to introduce Haskell and its type system with an example.

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x / y)
```

We see that Haskell has base types like `Int`, higher-level types like `Maybe`, and function types to combine those.

One step of a Haskell compiler, like the Glasgow Haskell Compiler (GHC), is to prove that the program is type correct. In particular, GHC elaborates Haskell surface syntax to an intermediate language (Core). This elaboration process consists of both type inference and checking, as well as a desugaring step to Core.

The use of an intermediate language is a common strategy to implement compilers for statically typed languages. Core is a typed language, it has a much smaller syntax than the surface syntax and it is backed by a formal theory (System Fc) [9].

Core expressions (CoreExpr) are defined by the `Expr b` data type, where `b` ranges over binders:

```
data Expr b
  = Var b                          -- Variable reference
  | Lit Literal                    -- Literal (e.g. Int#, Char#)
  | App (Expr b) (Arg b)           -- Function application
  | Lam b (Expr b)                 -- Lambda abstraction
  | Let (Bind b) (Expr b)          -- Non-recursive or recursive
  ↪ bindings
  | Case (Expr b) b Type [Alt b]   -- Pattern match
  | Cast (Expr b) Coercion         -- Type coercion
```

```
  | Tick (Tickish b) (Expr b)      -- Annotations for
  ↪ profiling/debugging
  | Type Type                      -- Explicit type (no runtime
  ↪ code)
  | Coercion Coercion              -- Explicit coercion (no runtime
  ↪ code)
```

2.1.1 GHC Holes and type-driven development

Typed holes are a feature of GHC that allows for the insertion of placeholders in expressions. The syntax uses leading underscores (`f x = _foo`) to represent these holes. When compiling, GHC will report an error that explains which type is expected at the hole.

Holes are quite useful in a technique called type-driven development where the programmer inserts holes and observes the reported type to be able to continue the development. In this style, the stronger the type system, the more useful holes are to the user.

2.2 Refinement Types and LiquidHaskell

Refinement types allow the specification and verification of semantic properties of programs. This is accomplished by augmenting the type with a logical predicate that effectively restricts the allowed values of that type [2].

As an example, we can encode a precondition on a division function ensuring at compile time that division by zero does not happen. Thus, we do not need to have as return type `Maybe Int` like previously shown.

```
div :: Int -> {v : Int | v /= 0} -> Int
```

A refinement type checker will only allow calls to the `div` function if it can guarantee that the second argument is never zero. One technique to check and infer these types is Liquid Types [10]. It works by checking that at each call-site, the types of the arguments are subtypes of the function inputs.

In this specific example, it would be checked in the second argument that the input is a subtype of `{v : Int | v /= 0 }`. This subtyping query later is discharged to a logical implication that is solved by an off-the-shelf SMT (Satisfiability Modulo Theories) solver.

LiquidHaskell is an implementation of a refinement type system for Haskell [3]. It uses Liquid Types [10] and also handles the lazy semantics of Haskell.

Initially, LiquidHaskell predicates on types could only come from decidable logic, but in [4], it received support to specify and verify properties outside the decidability realm. Additionally, an equational reasoning library was implemented and more interesting proofs like in Figure 2.1 could be expressed using LiquidHaskell.

```
{-@ involutionP :: xs:[a] -> {reverse (reverse xs) == xs} @-}
involutionP :: [a] -> Proof
involutionP [] = reverse (reverse [])
    -- applying inner reverse
    ==. reverse []
    -- applying reverse
    ==. []
    *** QED
involutionP (x:xs) = reverse (reverse (x:xs))
      -- applying inner reverse
      ==. reverse (reverse xs ++ [x])
          ? distributivityP (reverse xs) [x]
      ==. reverse [x] ++ reverse (reverse xs)
          ? involutionP xs
      ==. reverse [x] ++ xs
          ? singletonP x
      ==. [x] ++ xs
      -- applying append on []
      ==. x:([]++ xs)
      -- applying ++
      ==. (x:xs)
      *** QED
```

Figure 2.1: Proof by induction of involution [11].

2.3 LiquidHaskell Architecture

As mentioned above, LiquidHaskell implements a refinement type system on top of Haskell. The initial versions were a standalone executable (using GHC as a library) that would run on individual Haskell files and report the output from the refinement type checker. This strategy had a couple of drawbacks. For example, there was no integration with IDE tools and one would run only one file at a time. Fortunately, those issues were addressed when LiquidHaskell was converted to a GHC plugin [12].

2.3.1 GHC Plugins

A GHC Compiler Plugin is a feature that enables modifications to different phases and behaviors of the compiler. For example, it can inspect and transform GHC's intermediate language (Core) [13].

A plugin is implemented as a module and exports at least one identifier of type `GHC.Plugins.Plugin`. It has a function which effectively installs a pass in the compiler pipeline.

Furthermore, there are different kinds of plugins that attach at different phases of the compiler pipeline,

- **Core Plugins:** Operate on Core allowing for custom analyses.
- **Typechecker Plugins:** Operate on the constraint solver, allowing for modifications on it.
- **Source Plugins:** Can access different representations of the source code and its purpose is to facilitate the implementation of development tools.
- **Frontend Plugins:** Operate on parsing flags and administrative tasks, facilitating users to add new major modes to GHC.

2.3.2 High Level View

LiquidHaskell works by intercepting the Haskell code that needs to be type checked, generating the corresponding constraint problem, and sending it to *liquid-fixpoint*[1]. If a solution is found then a report is generated and the compiler pipeline proceeds, if no solution is found, an error report is generated and the pipeline is stopped. Figure 2.2 is a high-level view of this architecture.

First, LiquidHaskell hooks into the phase immediately after parsing, where it collects comments containing refinement types (specifications) and constructs a corresponding data structure.

Next, it operates after GHC's type-checking phase. At this stage, it proceeds to desugar each module using the GHC API, transforming them into unoptimized Core, which is needed for LiquidHaskell to operate correctly [14]. With both the

[1] `https://github.com/ucsd-progsys/liquid-fixpoint`

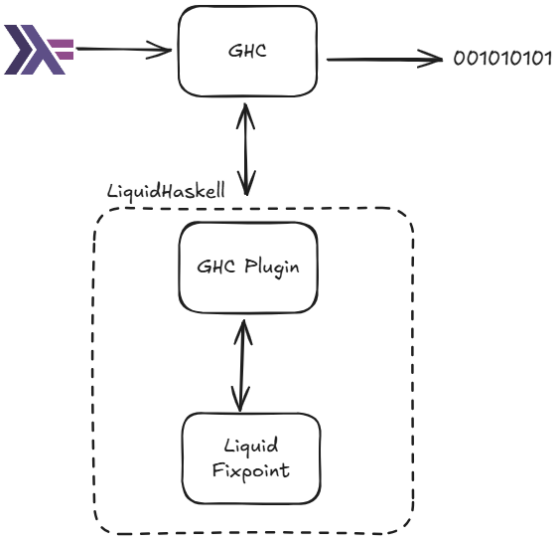

Figure 2.2: High Level architecture of LiquidHaskell

Core representation and LiquidHaskell specification, it prepares the code for further analysis.

2.3.3 Constraint Generation and Solving

After the desugaring to Core, LiquidHaskell traverses the Core syntax tree to generate a set of constraints.

The constraint generation top level function is

```
generateConstraints :: TargetInfo -> CGInfo
```

where `TargetInfo` is a data structure that was built by LiquidHaskell, that holds the specification and Core binds (a bind is an association between a variable and its definition in Core). And `CGInfo` holds the constraint problem generated.

The algorithm for constraint generation is syntax-directed on the GHC's Core syntax tree. Furthermore, it is bidirectional, so there is both a 'checking' and a 'synthesis' function. In the checking phase an expression and a constraint are verified against a known constraint, in the synthesis phase the constraint is inferred. The output of the constraint generation is a collection of subtyping queries. A subtyping query has the following shape

$$\Gamma \vdash \{\nu : b \mid e_1\} <: \{\nu : b \mid e_2\}$$

where Γ is an *environment*, which is a sequence of type bindings, e_1 and e_2 are boolean expressions that restrict the allowed values of the refined type to those that make the expression true. Last, $<:$ is a subtyping relation that hold if and only if e_1 implies e_2 .

The subtyping relation is used to capture the requirements the specification imposes. For example,

$$\cdots \vdash \{\nu : \text{Int} \mid \kappa\} <: \{\nu : \text{Int} \mid \nu \neq 0\}$$

claims that values of the refinement type $\{\nu : \text{Int} \mid \kappa\}$ must be non-zero.

The way the constraint problem is solved is by translating the subtype queries to logical formulas. The rough translation intuitively is that the environment becomes an assumption and the refinement in the subtype implies the refinement in the supertype [3].

To illustrate this process, consider the following code:

```
{-@ safeDiv :: x:Int -> {v:Int | v /= 0} -> Int @-}
safeDiv :: Int -> Int -> Int
safeDiv x y = x `div` y

good = safeDiv 42 one
bad  = safeDiv 42 zero

one  :: { v:Int | 0 < v  }
one  = 1

zero :: { v:Int | 0 == v }
zero = 0
```

LiquidHaskell checks that the second argument y is never zero before the division. The Core translation of the function roughly looks like:

```
safeDiv = \x y -> div x y
```

From this, the constraint generation phase will produce subtyping queries at the application site of div. For example, the call in *good* will generate the following subtyping query:

$$\Gamma \vdash \{\nu : \text{Int} \mid 0 < \nu\} <: \{\nu : \text{Int} \mid \nu \neq 0\}$$

Here Γ is a type environment containing the variable y bound to a refinement, the left-hand side of the query comes from the argument passed to y and the right-hand side is imposed by the function's specification.

The translation to logical formula transform convert this subtyping query into:

$$\forall \nu.\Gamma \implies 0 < \nu \implies \nu \neq 0$$

The verification succeeds since $0 < \nu \implies \nu \neq 0$ holds. On the other hand, at the call site of *bad* the query fails since $0 = \nu \implies \nu \neq 0$ does not hold.

2.4 Proof Assistants

A proof assistant is a piece of software that helps the development of formal proofs. It normally involves a form of interactive interface where a proof is developed with help from the proof assistant.

The interactive experience is such a fundamental feature of those programs, they are even called interactive theorem provers. As hinted in the introduction, theorem provers have features that allows the users to construct proofs in a step by step observing the intermediate goals and proceding interactively.

One way to categorize proof assistants is to differentiate between tactic-based proof development and hole-driven development. With tactics, the user develops with a set of predefined tactics or commands manipulating the proof state using this meta-language. The user applies tactics in a sequence to construct the proof. Coq and Isabelle are well-known examples of tactic-based proof assistants. With holes, the proof assistant identifies gaps in the proof that need to be filled. The user can then interact with these holes, through an IDE or editor, to refine the proof. The system provides feedback such as current constraints and expected type of goals. Agda and Idris are well-known examples of hole-driven development.

Here is a non-exhaustive list of proof assistants and a brief description of them.

- **Agda** is a functional programming language and also an interactive proof assistant [7]. One of its key features is typed holes, which inspired GHC's typed holes. Agda uses the Curry-Howard isomorphism, where propositions can be represented as types and vice versa. It supports inductive data types and an advanced pattern matching to construct proofs.

- **Idris**, similarly to Agda, is a functional programming language with dependent types [15]. It also has typed holes and an interactive editing experience both with the Idris REPL and also with *Vim* and *Emacs*. Idris has less focus on proofs than Agda.

- **Lean** is both a functional programming language and an interactive theorem prover [16]. It is designed with dependent types and supports the Curry-Howard correspondence, where propositions are types and proofs are programs. Lean provides powerful tactic-based proof construction as well as direct term-style proofs. It features an interactive development environment through tools like *VS Code*, and supports features like typed holes to guide proof development. The Lean ecosystem includes *mathlib*, a large and growing library of formalized mathematics.

- **Rocq (formerly Coq)** is an interactive theorem prover [17]. It supports both declarative and tactic-based proof development, enabling users to write machine-checked formal proofs. Rocq is widely used in academia and industry for verifying mathematical theorems and software correctness. It integrates well with development environments like CoqIDE and VS Code and in proof mode it shows goals and current proof state.

As previously introduced, LiquidHaskell lacks interactivity during the development of proofs. This thesis attempts to bridge this gap between LiquidHaskell and other proof assistants. More specifically, and inspired by Agda, we implement typed holes with LiquidhHaskell as a step forward in bridging this gap.

3

Implementation

3.1 Context

In [18], the authors propose an integration between LiquidHaskell and GHC Holes, but this work consisted of only a proposal, and there was no implementation.

In LiquidHaskell, the status quo is that when using a GHC hole in an expression, the error message is not useful at all. As we saw in the introduction chapter, even a fatal error can be thrown when using a hole.

There was some work in LiquidHaskell to detect holes, but the end goal was program synthesis [19]. This work had some limitations on the detection. For example, it only worked with bindings that followed this form `test = _`. Furthermore, the hole detection from this previous work was no longer compatible with latest GHC as it was developed targeting a relative old version of GHC. Nonetheless, it was a head start on how to detect holes and explore the code base.

3.1.1 First Iteration

The first iteration consisted of detecting a GHC Hole based on how it was desugared to GHC Core. For example, the following code:

```
foo :: a -> a
foo = _
```

was desugared to the Core representation shown in Figure 3.1

It required the implementation to detect those empty cases and save the refinement types information about the hole somewhere.

```
\ _ ->
    case GHC.Internal.Control.Exception.Base.typeError
            @GHC.Types.LiftedRep
            @()
            "OMITTED ERROR MESSAGE"#
    of {}
```

Figure 3.1: Representation of a function with hole in it in Core.

There were two ideas, (i) we could search directly in the GHC surface syntax and save the location of the hole and in a later phase query LiquidHaskell about the constraint related to the location of hole. This idea would have the advantage of being more modular, but querying LiquidHaskell based on expression location did not appear feasible (ii) The other idea was that during constraint generation, if we find an expression that represents the typed hole, we save the checked constraint in some data structure and continue the constraint generation.

We proceeded with the second idea and extended the constraint generation algorithm. Specifically in the checking phase of the algorithm, if an expression matches the core representation of a hole (see Figure 3.1), we saved the associated checked refinement type into a map. To support this, we added a new field, `hsHoles :: Map Var HoleInfo`, to the data structure that carries constraint generation information (later referred to as `CGInfo`). Each hole is stored with a unique `Var` identifier.

During this iteration, we also discovered that LiquidHaskell has a normalization step in which expressions are transformed into A-normal form (ANF). This means that every subexpression gets assigned to its own fresh let binding, and since the hole is a subexpression (in some cases) it also gets assigned to a let binding.

The main issue of this iteration was guaranteeing that the desugaring of holes that GHC performs remains unaltered as GHC evolves, as happened already with the previous work for program synthesis. We could mitigate this problem with a test suite in LiquidHaskell that checks for breaking changes.

An alternative that we considered is to use a Valid Hole Fit plugin [20]. This kind of plugin is called when a typed hole is found and allows inspection of the content of the hole. Unfortunately, when we tried to validate this idea, we realized that valid hole fit plugins runs before the LiquidHaskell plugin (Source Plugin) making it impractical. We decided to leave it as future work since we would have to explore if we could change the order of the plugin calls in GHC itself.

3.1.2 Second Iteration

In the second iteration, we already had some foundation and more knowledge of how the constraint generation algorithm worked. Furthermore, we already had created some data structure to save hole-related information. We were not content with the matching algorithm and its coupling with GHC.

To tackle this and further refine our implementation, we considered a special syntax to represent holes that would be added as an earlier step in LiquidHaskell. Later in the constraint generation, instead of matching in a specific term that depends on GHC implementation, we could simply look for this special symbol.

In the end, we decided to follow a simpler approach by requiring the user to define a binding to `undefined` that has 'hole' as suffix in the name and using this binding instead of an underscore to represent the holes.

This allows us to reap the benefits of having a stable syntax to represent the holes, and also in the future have holes with different names. When this binding gets

```
module Example
    hole = undefined

    {-@ sum1 :: x:Int -> {v: Int | v == x + 1} @-}
    sum1 :: Int -> Int
    sum1 x = x + hole
```

Figure 3.2: Example of sum function with a hole.

```
let lq_anf#1 = Var 'hole'
let lq_anf#2 = Prim.Int 1
in  lq_anf#1 + lq_anf#2
```

Figure 3.3: Example of normalized core with hole

desugared to GHC Core, it becomes of the following form in GHC Core syntax `App (Tick (SourceNote srcLoc) ("Var hole")) _`.

At this moment, we also put the feature under a flag so that users can opt-in to use it. And we also added some test cases to LiquidHaskell in addition to documentation.

Example of the warning with some custom formatting:

```
typed-holes/Example0.hs:10:5: error:
Hole Found
Example0.hole :: x1:[a]
    -> {VV##0 : GHC.Types.Int | VV##0 == len x1
                              && VV##0 >= 0}
in the context
Example0.hole : forall a . a
   |
10 |     listLength = hole
   |     ^^^^^^^^^^^^^^^^
```

During this iteration, one persistent issue was deciding in which phase of the bidirectional constraint generation to detect the holes. In the end, we limited the scope and placed the detection only in the 'checking' phase. Unfortunately, this resulted in some drawbacks, for example, when the hole was used in a function application, the reported warning did not contain any useful information.

We then embarked on the quest to deal with the lack of useful information when a hole appears in the middle of a function application.

Consider the code in Figure 3.2, the output from this warning would just show that `hole :: GHC.Types.Int` without any refinement associated with it. That happens because during synthesis, when the hole is detected, not enough information is available.

Fortunately, later in the constraint generation we observed that due to the normalization step, the hole was actually assigned to a let-binding and this binding was

used in expressions that do have a refinement type. See Figure 3.3.

Then we proceeded with the following plan: During constraint generation, we also save the ANF let-bindings to holes. When we find that a let-binding is being used in an expression, we save the refined type of the expression to be shown to the user, since it might be a useful constraint.

3.2 Final Version

In this context, we now present the final version of the implementation. The code is currently in a fork of LiquidHaskell[1] and a pull-request is open towards the "develop" branch[2]. The relevant implementation is also available in Appendix A.1.

3.2.1 Detecting Holes

To use the hole detection feature, it is necessary to pass a flag to LiquidHaskell.

```
{-@ LIQUID "--warn-on-term-holes" @-}
```

The implementation of a new configuration flag in LiquidHaskell is straightforward, it is just an extension of two records in files *CmdLine.hs* and *Config.hs*, making this configuration available during later steps.

The algorithm for detecting holes has as entry point the function `detectTypedHole` which has the following signature `CoreExpr -> Maybe (RealSrcSpan, Var)`, as seen in Figure 3.4.

```
detectTypedHole :: CoreExpr -> Maybe (RealSrcSpan, Var)
detectTypedHole e =
  case stripTicks e of
    Var x | isVarHole x ->
      case lastTick e of
        Just (SourceNote src _) -> Just (src, x)
        _                       -> Nothing
    _ -> Nothing
```

Figure 3.4: Function to detect typed holes added to constraint generation.

It receives a core expression as input and tries to return a `Maybe` value containing a tuple `(RealSrcSpan, Var)` where the first element is the location of the hole and the second is the variable identifying the hole. If no hole is found, the value is `Nothing`.

It works by using the helper function `stripTicks`, which recursively removes the initial `App` and `Tick` nodes from the expression. This is necessary when the hole is used inside nested expressions, removing unnecessary layers to correctly find the variable.

[1] https://github.com/matheussbernardo/liquidhaskell
[2] https://github.com/ucsd-progsys/liquidhaskell/pull/2486

```
when (warnOnTermHoles (getConfig )) maybeAddHole
...
maybeAddHole = do
    let isItHole = detectTypedHole e
    case isItHole of
    Just (srcSpan, x) -> do
      addHole (RealSrcSpan srcSpan Strict.Nothing) x t
    _ -> return ()
```

Figure 3.5: Excerpt of function where hole detection was added.

When the stripped expression is a `Var` the function checks if the variable represents a hole using the helper `isVarHole`. It decides if a variable's name indicates a hole by checking if the string representation of the variable ends with '.hole', and it ignores the module name.

Lastly, it uses the helper function `lastTick` to retrieve its source span. It traverses the expression to find the last `Tick` node and extract its `SourceNote`, which contains the source span information.

We detect holes in three places of the constraint generation algorithm:

- **In the checking phase of the algorithm**, if the expression we are checking is a hole then we save the refinement type of the hole.
- **In the synthesis phase**, if the expression is a hole we synthesize the refinement type of the expression and save this information.
- **In the top-level core binds**, in this case the `detectTypedHole` function is used to detecting let-binding that are holes due to the ANF normalization.

The constraint generation algorithm was the chosen place to detect holes primarily because constraints are readily available at this point of the type checker.

In the checking phase, it works by checking the configured flag and, if enabled, calling the helper function `maybeAddHole` (function in Figure 3.5). If the expression being checked is a hole, we call `addHole` with the source span information, the variable, the refined type for the hole, and the constraint generation environment.

The `addHole` function saves the information in the `CGInfo` record specifically at the `hsHoles` field. It will be used later when we emit the necessary warning message.

Detection of holes during synthesis works very similarly as in checking, the difference is that the refined type saved is the synthesized one.

The third case where we need to detect holes is when processing a NonRec let binding in the `consCB` function. It checks if the expression being bound contains a typed hole by calling `detectTypedHole`. If a hole is detected, it links the hole to its ANF representation using `linkANFToHole`.

This detection of holes in ANF variables is necessary since in some cases there are relevant constraints where the hole actually appears as the ANF variable.

```
checkANFHoleInExpr :: CoreExpr -> SpecType -> CG ()
checkANFHoleInExpr e t = do
  let vars = collectVars e
  forM_ vars $ \var -> do
    isANF <- isANFInHole var
    case isANF of
      Just uniqueVar -> addHoleANF uniqueVar var e t
      _ -> return ()
```

Figure 3.6: Function to check if ANF Hole is in expression.

As we said, when we find that a let binding is actually an ANF variable bound to a hole, the ANF variable is linked to the unique hole name. This linking happens in a new field, `hsANFHoles`, that we added to the `CGInfo` record. This field is a map where the key is the ANF variable and the value is a tuple with the hole variable plus the source code location of it.

With this information saved, we need to find expressions where the ANF variable is present. The `checkANFHoleInExpr` implements this detection (see Figure 3.6), it functions by calling first `collectVars` which collect all the vars in the expression to a list, then we iterate over it checking if they are present in the `hsANFHoles` map. If a variable is found we call another helper function `addHoleANF` which adds an entry to another map in `CGInfo`. Finally, `checkANFHoleInExpr` is also called both in the checking and synthesis of the constraint generation.

To illustrate the algorithm, consider the code in Figure 3.2.

- During constraint generation, in the synthesis phase, since we are analyzing a function application, `App (+) (Var x) (Var hole)`. The algorithm recurses and finds the hole outputting `Just (SrcSpan, Var hole)` and saves it in the `hsHoles` field.
- Furthermore, since the hole is being used as an ANF variable, we also detect the ANF let binding corresponding to a hole and save it in the map linking the ANF and the hole. We have as input a `NonRec (Var anf) (Var hole)` and we modify the monad state linking these two variables.
- Lastly, we find expressions where the ANF variable is present, using the function `checkANFHoleInExpr` collecting all the vars and finding `Var anf` that links to a hole. In this specific case, we detect its use in the function application expression and add the following constraints to the map:

```
{v : Int | v == x + hole}
{v : Int | v == x + 1}
```

3.2.2 Consolidating Warning Messages

With all this information, the logical next step was to emit a warning message to the user, showing both the refinement type of the hole, the LiquidHaskell environment

and the constraints related to the hole found.

It functions by fetching the necessary fields from `CGInfo`, those that contain the holes and its source location, the expressions where ANF holes appear, and the mapping of holes and its related ANF variables. The second step combines the information of holes with the expressions where ANF holes appear, then we iterate over this combined list of holes and construct a warning using the existing function from LiquidHaskell, `addWarning`.

3.2.2.1 Prettify Warnings

The initial warning message contained too many references to ANF vars, making it hard for users to understand the content of the message. For example, refinement types that refer to ANF:

```
{v : GHC.Types.Int | v == lq_anf#123  + lq_anf#321}
```

As an ad-hoc solution, for now, we are using part of an implementation already available in LiquidHaskell so we can undo the ANF expressions and improve the output.

To be able to use the `inlineInExpr` function, two maps were used that related ANF variables to the original expression. First, constraint generation already has a map `binds` available at `CGInfo`, but for the hole this map was not usable because of the lack of source location information that is essential to make each hole unique, so we constructed a second map from `hsANFHoles`.

Finally, a helper function is called that takes those two maps as arguments and iterates over the refinement types at the warning message substituting the values to the desired ones.

Furthermore, we simplified the ANF variables and removed the qualified name of some modules. The `prettifySpecType` function centralized all these prettification. We refer to the Appendix A.1 for the full implementation.

3.2.3 Test Suite and Examples

During development, a set of examples was created. Mostly, it contains extrinsic proofs of [11]. It also has a simple intrinsic proof. Those examples were also added to LiquidHaskell as a test suite. The full set of examples is in Appendix A.2.

4

Evaluation

In this chapter, we present a comparison of the extension with other proof assistants. Furthermore, we show a benchmark on the performance impact of our extension on LiquidHaskell.

4.1 Comparison with other proof assistants

The comparison consisted of a proof in Agda, Idris, Lean and LiquidHaskell (with the extension). We compare the editor support, the quality of the error messages, and the overall feedback loop.

The proof checks that an implementation of scanl returns a list with one element longer than its input, i.e.,

$$\texttt{length (scanl } f\ q\ xs\texttt{)} \equiv 1 + \texttt{length } xs$$

Here, `length` refers to the standard list length function, and `scanl` is either the standard Haskell implementation or a structurally equivalent, custom-defined version in the other proof assistants. The type signature of `scanl` is:

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```

4.1.1 Proof in LiquidHaskell

We start the comparison with an intrinsic proof in LiquidHaskell about the length property of `scanl`, using our extension.

```
{-@ scanlLength :: (b -> a -> b) -> b -> xs:[a]
                -> {v : Nat | v == 1 + len xs} @-}
scanlLength :: (b -> a -> b) -> b -> [a] -> Int
scanlLength f q xs = hole
```

The generated output contains:

```
Hole Found
 hole
    :: f:(b -> a -> b)
     -> q:b
```

```
        -> x1:[a]
        -> {VV : Int | VV == 1 + len x1 && VV >= 1}
```

We can see that the hole is a function from a list to a `Nat`, and more importantly, this `Int` must equal `1 + len xs`. We continue by pattern matching on the list:

```
scanlLength f q []     = hole
scanlLength f q (x:xs) = hole
```

In the base case, the hole context shows that the result must equal 1, since the list is empty:

```
  Hole Found
   hole :: {VV : Int | VV == 1}
```

So we fill it in with 1. In the recursive case, the refinement contains:

```
{VV : Int | VV == 1 + (1 + len xs) }
```

which clearly suggests the right-hand side is:

```
1 + scanlLength f (f q x) xs
```

But now we move to a full extrinsic proof, first we define the proposition with Haskell function:

```
  {-@ scanlLength :: f:(b -> a -> b) -> q:b -> xs:[a]
                  -> { length (scanl f q xs) == 1 + length xs } @-}
  scanlLength :: (b -> a -> b) -> b -> [a] -> Proof
  scanlLength f q [] = hole
  scanlLength f q (x:xs) = hole
```

And when we use our extension, we see the expected type for the base (with some indirections from the ANF):

```
  Hole Found
    Example0.hole :: {VV : () |
      length (scanl f q anf) == 1 + length anf}
  in the context:
      q : b

      f : b -> a -> b

      anf : {VV : [a] | VV == []
                    && length VV == 0
                    && len VV == 0
                    && len VV >= 0}
```

On the inductive case, the output looks like:

```
  Hole Found
    Example0.hole :: {VV : () |
      length (scanl f q anf) == 1 + length anf}
```

```
  in the context
    x : a

    q : b

    xs : {VV : [a] | len VV >= 0}

    f : a -> b -> a

    anf : {VV : [a] | VV == : x xs
                      && length VV == 1 + length xs
                      && head VV == x
                      && len VV == 1 + len xs
                      && lqdc VV == x
                      && lqdc VV == xs
                      && tail VV == xs
                      && len VV >= 0}
```

To construct the proof, we use LiquidHaskell equational reasoning library.

```
scanlLength f q [] =
  length (scanl f q [])
  === hole
  === 1 + length []
  *** QED
```

We need to unfold all the definitions and the typed hole for example helps when it sees that.

```
Hole Found
  Example0.hole :: { VV : Int | VV == length (scanl f q []) }
```

We can unfold the definitions with the proof:

```
scanlLength f q [] =
  length (scanl f q [])
  === length [q]
  === 1
  === 1 + length []
  *** QED
```

Now we go for the inductive case, recall that LiquidHaskell output was:

```
  Hole Found
    Example0.hole :: {VV : () |
      length (scanl f q anf) == 1 + length anf}
```

So we start the proof with our initial and final goal:

```
scanlLength f q (x:xs)
  =   length (scanl f q (x:xs))
```

```
  === hole
  === 1 + length (x:xs)
  *** QED
```

On this case, is valid to note that the messages from the hole gets less helpful because of the noise from ANF. Nonetheless, by continuing to unfold definitions, we get to the full proof as:

```
scanlLength f q (x:xs)
  =   length (scanl f q (x:xs))
  === length (q : scanl f (f q x) xs)
  === 1 + length (scanl f (f q x) xs)
      ? scanlLength f (f q x) xs
  === 1 + (1 + length xs)
  === 1 + length (x:xs)
  *** QED
```

4.1.2 Proof in Agda

In Agda, we encode the same property by stating that the result of `scanl` has a length equal to `suc (length xs)`.

We define `scanl`:

```
scanl' : (b -> a -> b) -> b -> List a -> List b
scanl' f q []       = [q]
scanl' f q (x::xs) = q :: scanl' f (f q x) xs
```

We start the proof by defining the proposition and with a hole:

```
scanlLength : {A B : Set} (f : B -> A -> B) (q : B) (xs : List A)
            -> length (scanl' f q xs) == suc (length xs)
scanlLength = ?
```

We load this in *Emacs* with *agda-mode*, Agda responds with:

```
?0 : {A B : Set} (f : B -> A -> B) (q : B) (xs : List A) ->
       length (scanl' f q xs) == suc (length xs)
```

We add arguments and after case splitting:

```
scanlLength f q [] = ?
scanlLength f q (x :: xs) = ?
```

Looking at the base case shows the goal:

```
?0 : length (scanl' f q []) == suc (length [])
```

This is solved with `refl` because of normalization at the type level means that Agda can tell that both sides are equal.

```
scanlLength f q [] = refl
```

In the inductive case, we start with a high level hole that shows:

```
scanlLength f q (x :: xs) = ?

?1 : length (scanl' f q (x :: xs)) == suc (length (x :: xs))
```

Using equational reasoning and by definition of `scanl'` and `length`, we get to this intermediate step:

```
scanlLength f q (x :: xs) =
  begin
    length (scanl' f q (x :: xs))
  ==()
    length (q :: scanl' f (f q x) xs)
  ==()
    suc (length (scanl' f (f q x) xs))
  ==( {!!} )
    suc (length (x :: xs))
  QED
```

Here the hole has the following goal:

```
Goal: suc (length (scanl' f (f q x) xs)) == suc (length (x :: xs))
```

And we can apply the inductive hypothesis and by definition of length we finish the proof:

```
scanlLength f q (x :: xs) =
  begin
    length (scanl' f q (x :: xs))
  ==()
    length (q :: scanl' f (f q x) xs)
  ==()
    suc (length (scanl' f (f q x) xs))
  ==(  cong suc (scanl-length f (f q x) xs) )
    suc (suc (length xs))
  ==()
    suc (length (x :: xs))
  QED
```

4.1.3 Proof in Idris

Idris, similarly to Agda, is a functional programming language with dependent types. It also has typed holes and an interactive editing experience both with the Idris REPL and also with *Vim* and *Emacs*.

To compare the experiences, we prove the same proposition. Here, `scanl'` is user-defined:

```
scanl' : (b -> a -> b) -> b -> List a -> List b
scanl' f q [] = [q]
scanl' f q (x :: xs) = q :: scanl' f (f q x) xs
```

The goal is to prove:

```
scanlLength : (f : b -> a -> b) -> (q : b) -> (xs : List a)
            -> length (scanl' f q xs) = 1 + length xs
scanlLength f q xs = ?hole
```

We can inspect the `?hole` hole in the REPL with the following command, `:t hole` and in Emacs with *C-c C-t*.

With the cursor in 'xs', we can case split (in Emacs C-c C-c).

```
scanlLength f q [] = ?hole_0
scanlLength f q (x :: xs) = ?hole_1
```

We see the hole types in a new buffer:

```
- + "Main.hole_0" [P]
 `--          0  a : Type
              0  b : Type
                 q : b
                 f : b -> a -> b
     ------------------------------
      "Main.hole_0" : 1 = 1

- + "Main.hole_1" [P]
 `--          0  a : Type
              0  b : Type
                 q : b
                 x : a
                xs : List a
                 f : b -> a -> b

     ↪ --------------------------------------------------------------------------
      "Main.hole_1" : S (length (scanl' f (f q x) xs)) = S (S (length
       ↪  xs))
```

We can complete the first hole with proof search (in Emacs C-c C-a).

```
scanlLength f q [] = Refl
```

For the inductive case, the hole type is:

```
- + "Main.hole_1" [P]
 `--          0  a : Type
              0  b : Type
                 q : b
                 x : a
                xs : List a
                 f : b -> a -> b

     ↪ --------------------------------------------------------------------------
```

```
      "Main.hole_1" : S (length (scanl' f (f q x) xs)) = S (S (length
      ↪  xs))
```

We have to tell Idris to apply the inductive case, the final proof is:

```
scanlLength : (f : b -> a -> b) -> (q : b) -> (xs : List a) ->
              length (scanl' f q xs) = 1 + length xs
scanlLength f q [] = Refl
scanlLength f q (x :: xs) = rewrite scanlLength f (f q x) xs in Refl
```

Furthermore, Idris also has a command to automatically separate a lemma from the hole (*Emacs: C-c C-l*). We refer to the Idris documentation[1] for the full set of commands.

4.1.4 Proof in Lean

The development experience in Lean with *VSCode* is quite polished. Lean also uses the Curry-Howard Isomorphism but, differently than Agda, it has support for tactics (which can be seen as macros for generating proof terms). Nonetheless, it also has term holes. The interaction with the language in *VSCode* is mainly based on a side tab called *Lean InfoView*, see Figure 4.1.

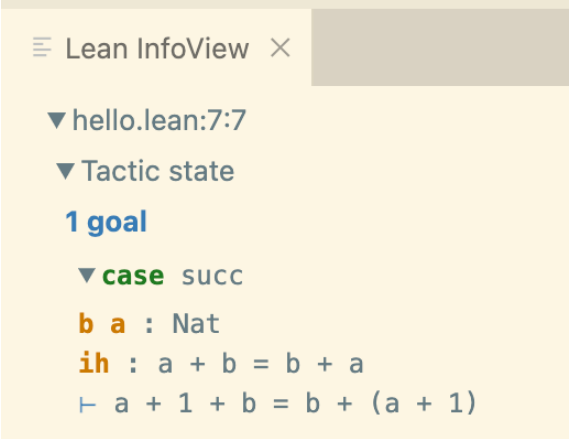

Figure 4.1: Lean InfoView in *VSCode*.

We start by defining `scanl`.

```
def scanl' {A B : Type} (f : B → A → B) (q : B) : List A → List B
| []       =>  [q]
| x :: xs => q :: scanl' f (f q x) xs
```

Next, we define the theorem.

```
theorem scanlLength {A B : Type} (f : B → A → B) (q : B) (xs : List
↪  A) :
  (scanl' f q xs).length = 1 + xs.length :=
  sorry
```

[1] https://idris2.readthedocs.io/en/latest/tutorial/interactive.html

Example	Status	Allocation (GB)	Time (ms)
1	Disabled	0.603	1012
1	Enabled	0.606	1010
2	Disabled	0.937	1059
2	Enabled	0.974	1094
3	Disabled	2.604	1649
3	Enabled	2.773	1769

Table 4.1: LiquidHaskell Performance with Different Configurations

and *Lean InfoView* shows the expected goal:

```
(scanl' f q xs).length = 1 + xs.length
```

We case split and complete the base case:

```
theorem scanlLength {A B : Type} (f : B → A → B) (q : B) (xs : List
↪  A) :
  (scanl' f q xs).length = 1 + xs.length :=
  match xs with
  | []       => rfl
  | x :: xs => sorry
```

For the inductive case we use tactics to unfold definitions and apply inductive hypothesis:

```
theorem scanlLength {A B : Type} (f : B → A → B) (q : B) (xs : List
↪  A) :
  (scanl' f q xs).length = 1 + xs.length :=
  match xs with
  | []       => rfl
  | x :: xs => by
    simp [scanl']
    rw [scanlLength f (f q x) xs]
    rfl
```

4.2 Performance Impact

In this section, we analyze whether the extension has any impact on the performance of LiquidHaskell. The analysis uses GHC's timing capability with the `-ddump-timing` flag.

The benchmark was executed in 2021 MacBook Pro M1 Pro with 32Gb of RAM. We executed the files with the extension enabled (`--warn-on-term-hole`) and disabled.

The results are in table 4.1.

When the extension is enabled, there is a small amount of overhead, but nothing perceptible to the user.

4.3 Discussion and Limitations

When we compare LiquidHaskell error messages with other theorem provers, it is clear that LiquidHaskell needs improvements in that aspect. We suspect that the ANF pass in LiquidHaskell plus the pragmatic approach of integrating directly with an existing language make the error messaging system more complex and with more indirection than when building a proof assistant from the ground up.

On the flip side, LiquidHaskell is well integrated with GHC tooling, and the warning message from hole appear, for example, when using IDEs that integrate with the Haskell Language Server.

The overall feedback loop in Agda, Idris and Lean allows for a more interactive experience than what the extension current enables, nevertheless, the work on this thesis provides a step towards a more interactive experience, we expand on this in the Future Work in Chapter 6.

Another limitation is, how reliable an SMT solver actually is. For example, if we want to add more interactivity to LiquidHaskell, specially, when the proofs start to get bigger, will the SMT solver take too long and is the feedback in the correct granularity. There is a balance between automation and manual proofing to be found.

5

Related Work

5.1 Proof By Logical Evaluation

```
{-@ rightIdP :: xs:[a] -> { xs ++ [] == xs } @-}
rightIdP :: [a] -> Proof
rightIdP []
  =.  [] ++ []
  ==. []
  *** QED
rightIdP (x:xs)
  =.  (x:xs) ++ []
  ==. x : (xs ++ [])
      ? rightIdP xs
  ==. x : xs
  *** QED
```

```
{-@ ple rightIdP @-}
{-@ rightIdP :: xs:[a] -> { xs ++ [] == xs } @-}
rightIdP :: [a] -> Proof
rightIdP [] = ()
rightIdP (x:xs) = rightIdP xs
```

Figure 5.1: Proof on the top without PLE while on the bottom one automation is enabled

A proof automation built by the authors of LiquidHaskell is a technique called proof by logical evaluation (PLE). It works with two mechanisms; first, with refinement reflection arbitrary functions can be reflected to the logic language allowing reasoning about them; second, when PLE is enabled, LiquidHaskell unfolds function applications until the logical formula is proved or a fixpoint is reached [4].

In Figure 5.1, we have an example of how much PLE can save on proof lines. This is clearly a great advance, but when writing proofs, automation can hide important steps and also make proofs too concise, reducing readability of them.

In contrast, in this thesis we focus on helping users understand the proof they are writing and do not apply automation.

5.2 Liquid Proof Macros

```
{-@ assocMin :: a:N → b:N → c:N →
{min (min a b) c == min a (min b c)} @-}
[tactic|
assocMin :: N -> N -> N -> Proof
assocMin a b c = induct a; induct b; induct c
|]
```

Figure 5.2: Associativity of min using Liquid Proof Macros

This work, like ours, detected the lack of interactivity and the cumbersome process of writing proofs in LiquidHaskell. However, the proposed solution explicitly circumvented this problem. Instead, it focused on creating a DSL to automate the development of proofs [21].

As an example of the DSL the paper presents a proof of associativity, as in 5.2.

This DSL was implemented as a macro using Template Haskell and benchmarked against a set of proofs showing a reduction of code in average.

Despite the advances, it does not try to solve the problem of interactivity in LiquidHaskell, while our main focus is exactly on it. We can envision some future work to integrate IDE automation and emit the DSL language for some proof if the user opts in.

5.3 Hazel Language

This work is a live functional programming created around typed holes. In their model, they are able to typecheck, manipulate and even run incomplete programs.

```
# Empty holes stand for missing expressions, patterns,
let empty_hole = ⦇⦈ in

# Non-empty holes are the red boxes around type errors
# (you can still run programs with non-empty holes) #
let non_empty_hole : Int = ⦇⦈ in
```

Figure 5.3: Example of Hazel Live environment with holes.

We can see in Figure 5.3, an example of an expression with holes. In Hazel, the editor is different than traditional text editors, it follows a principle where there are no meaningless editor states, so code is always syntactic correct.

In contrast with our thesis, they designed the language from scratch and have editor services well integrated with holes to assists users.

5.4 Developer Experience in interactive theorem provers

As a more general related work, studies of students learning Agda found that unclear error messages and weak tooling are a big barrier to enter [22]. A similar study [23], note that the VSCode extension for Rocq lacks many interactive features and create similar barriers.

This is in line with our assessment that LiquidHaskell can benefit from improving error messages and integrating interactive features to its tooling. We suggest some next steps in the Future Work in Chapter 6.

6

Conclusion

The main problem guiding this project was: "How to create a more interactive experience when doing proof-development in LiquidHaskell?". This question emerged from observing that LiquidHaskell was lacking features to make proof development more interactive.

We focused on the sub-problem of "How can typed holes be integrated into LiquidHaskell and provide a foundational step toward interactivity?". It was natural to explore typed holes as the essential feature to enable interactivity, since LiquidHaskell is a tool built on top of an existing programming language, when doing proofs in LiquidHaskell, you are effectively writing Haskell code. And, as such there is no manipulation using a meta-language.

As we see in the evaluation chapter, the extension provides LiquidHaskell with a feature that is available in all major theorem provers. This is an important step towards more interactivity, given that with hole display functionality, we also enable new interactive features in LiquidHaskell.

6.1 Future Work

- One small improvement to the experience with holes could be extending it to have an options field associated with each hole; this would allow to name the hole and potentially add other attributes to give more clarity at debugging or hints at some future automation. Something like:

  ```
  foo = hole(HoleOptions { holeName = Just "foo" })
  ```

- The error messages in LiquidHaskell are somewhat confusing, especially given the indirections that ANF pass generates. We did an ad hoc improvement to the hole warning messages, but it would be good to investigate if there is a more sound approach to improving it, maybe in a more general way for all warning messages of LiquidHaskell.
- Another step towards a more interactive LiquidHaskell would be to conduct a user study to understand where interactivity matters when doing large-scale proofs, so we can prioritize features.
- Based on this user study and benchmarked against other tools, implement a set of commands to add interactivity to LiquidHaskell; we envision that this

could be done with an integration with the Language Server Protocol, allowing for reuse between multiple editors.

- LiquidHaskell could become a less black-box system, maybe with a LiquidHaskell API that users can call programmatically, allowing for new tools and developments.

Bibliography

[1] T. Freeman and F. Pfenning, "Refinement types for ml," *SIGPLAN Not.*, vol. 26, no. 6, pp. 268–277, May 1991, ISSN: 0362-1340. DOI: 10.1145/113446.113468. [Online]. Available: https://doi.org/10.1145/113446.113468.

[2] H. Xi and F. Pfenning, "Eliminating array bound checking through dependent types," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, Montreal Quebec Canada: ACM, May 1998, pp. 249–257, ISBN: 978-0-89791-987-6. DOI: 10.1145/277650.277732. [Online]. Available: https://dl.acm.org/doi/10.1145/277650.277732 (visited on 11/12/2024).

[3] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for haskell," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '14, New York, NY, USA: Association for Computing Machinery, Aug. 19, 2014, pp. 269–282, ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628161. [Online]. Available: https://doi.org/10.1145/2628136.2628161 (visited on 11/12/2024).

[4] N. Vazou, A. Tondwalkar, V. Choudhury, *et al.*, "Refinement reflection: Complete verification with smt," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158141. [Online]. Available: https://doi.org/10.1145/3158141.

[5] Y. Liu, J. Parker, P. Redmond, L. Kuper, M. Hicks, and N. Vazou, "Verifying replicated data types with typeclass refinements in liquid haskell," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428284. [Online]. Available: https://doi.org/10.1145/3428284.

[6] N. Vazou, L. Lampropoulos, and J. Polakow, "A tale of two provers: Verifying monoidal string matching in liquid haskell and coq," *SIGPLAN Not.*, vol. 52, no. 10, pp. 63–74, Sep. 2017, ISSN: 0362-1340. DOI: 10.1145/3156695.3122963. [Online]. Available: https://doi.org/10.1145/3156695.3122963.

[7] U. Norell, *Towards a practical programming language based on dependent type theory*, vol. 32.

[8] S. Marlow *et al.*, "Haskell 2010 language report," *Available online http://www.haskell. org/(May 2011)*, 2010.

[9] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly, "System f with type equality coercions," in *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, 2007, pp. 53–66.

[10] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 159–169, ISBN: 9781595938602. DOI: 10.1145/1375581.1375602. [Online]. Available: https://doi.org/10.1145/1375581.1375602.

[11] N. Vazou, J. Breitner, R. Kunkel, D. Van Horn, and G. Hutton, "Theorem proving for all: Equational reasoning in liquid haskell (functional pearl)," *SIGPLAN Not.*, vol. 53, no. 7, pp. 132–144, Sep. 2018, ISSN: 0362-1340. DOI: 10.1145/3299711.3242756. [Online]. Available: https://doi.org/10.1145/3299711.3242756.

[12] "Liquidhaskell is a ghc plugin." (2020), [Online]. Available: https://ucsd-progsys.github.io/liquidhaskell-blog/2020/08/20/lh-as-a-ghc-plugin.lhs/ (visited on 03/27/2025).

[13] "Liquidhaskell is a ghc plugin." (2020), [Online]. Available: https://downloads.haskell.org/ghc/latest/docs/users_guide/extending_ghc.html#compiler-plugins (visited on 03/27/2025).

[14] A. D. Napoli, *Implementing a ghc plugin for liquid haskell*, https://well-typed.com/blog/2020/08/implementing-a-ghc-plugin-for-liquid-haskell/, Accessed: 2025-04-11.

[15] E. BRADY, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, no. 5, pp. 552–593, 2013. DOI: 10.1017/S095679681300018X.

[16] L. d. Moura and S. Ullrich, "The lean 4 theorem prover and programming language," in *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 625–635, ISBN: 978-3-030-79875-8. DOI: 10.1007/978-3-030-79876-5_37. [Online]. Available: https://doi.org/10.1007/978-3-030-79876-5_37.

[17] T. R. D. Team, *The rocq prover*, version 9.0, Apr. 2025. DOI: 10.5281/zenodo.15149629. [Online]. Available: https://doi.org/10.5281/zenodo.15149629.

[18] P. Redmond, G. Shen, and L. Kuper, "Toward hole-driven development with liquid haskell," *arXiv preprint arXiv:2110.04461*, 2021.

[19] "Holes by nikivazou." (2019), [Online]. Available: https://github.com/ucsd-progsys/liquidhaskell/pull/1440 (visited on 04/26/2025).

[20] M. P. Gissurarson, "Suggesting valid hole fits for typed-holes (experience report)," *SIGPLAN Not.*, vol. 53, no. 7, pp. 179–185, Sep. 2018, ISSN: 0362-1340. DOI: 10.1145/3299711.3242760. [Online]. Available: https://doi.org/10.1145/3299711.3242760.

[21] H. Blanchette, N. Vazou, and L. Lampropoulos, "Liquid proof macros," in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, ser. Haskell 2022, Ljubljana, Slovenia: Association for Computing Machinery, 2022, pp. 27–38, ISBN: 9781450394383. DOI: 10.1145/3546189.3549921. [Online]. Available: https://doi.org/10.1145/3546189.3549921.

[22] S. Juhošová, A. Zaidman, and J. Cockx, "Pinpointing the learning obstacles of an interactive theorem prover," in *International Conference on Program*

Comprehension, ICPC (2025). https://sarajuhosova. com/assets/files/2025-icpc. pdf Accepted, 2025.

[23] H. C. Tavante, "A data-centered user study for proof assistant tools.," in *PPIG*, 2021.

A

Appendix

A.1 Implementation Code

A.1.1 Detect Typed Hole Function

```
detectTypedHole :: CoreExpr -> Maybe (RealSrcSpan, Var)
detectTypedHole e =
  case stripTicks e of
    Var x | isVarHole x ->
      case lastTick e of
        Just (SourceNote src _) -> Just (src, x)
        _                       -> Nothing
    _ -> Nothing
```

A.1.2 Detect Typed Hole Helper Functions

```
-- Remove Initial App and sequent Tick nodes from an expression.
stripTicks :: CoreExpr -> CoreExpr
stripTicks (App (Tick _ e) _) = stripTicks e
stripTicks (Tick _ e)         = stripTicks e
stripTicks e          = e

-- Traverse the expression to get the last Tick information.
lastTick :: Expr b -> Maybe CoreTickish
lastTick (Tick t e) =
  case lastTick e of
    Just t' -> Just t'
    Nothing -> Just t
lastTick (App e a) =
  case lastTick a of
    Just ta -> Just ta
    Nothing -> lastTick e
lastTick _ = Nothing

-- A helper to check if the variable name indicates a typed hole.
isVarHole :: Var -> Bool
```

```
isVarHole x = isHoleStr (F.symbolString (F.symbol x))
  where
    isHoleStr s =
      case break (== '.') s of
        (_, '.':rest) -> rest == "hole"
        _             -> False
```

A.1.3 Example of patch to constraint generation

```
cconsE'   e t
  = do
       when (warnOnTermHoles (getConfig ))  maybeAddHole
       ...
  where
    maybeAddHole = do
      let isItHole = detectTypedHole e
      case isItHole of
        Just (srcSpan, x) -> do
          addHole (RealSrcSpan srcSpan Strict.Nothing) x t
        _ -> return ()
```

A.1.4 Function Check ANF Hole in Expressions

```
checkANFHoleInExpr :: CoreExpr -> SpecType -> CG ()
checkANFHoleInExpr e t = do
  let vars = collectVars e
  forM_ vars $ \var -> do
    isANF <- isANFInHole var
    case isANF of
      Just uniqueVar -> addHoleANF uniqueVar var e t
      _ -> return ()
collectVars :: CoreExpr -> [Var]
collectVars (Var x) = [x]
collectVars (App e1 e2) = collectVars e1 ++ collectVars e2
collectVars (Lam x e) = x : collectVars e
collectVars (Let (NonRec x e1) e2) = x : collectVars e1 ++
↪  collectVars e2
collectVars (Let (Rec xes) e) =
  let (xs, es) = unzip xes
  in xs ++ concatMap collectVars es ++ collectVars e
collectVars (Case e x _ alts) =
  x : collectVars e ++ concatMap collectAltVars alts
  where collectAltVars (Alt _ xs e') = xs ++ collectVars e'
collectVars _ = []
```

A.1.5 Emit Consolidated Warning

```
emitConsolidatedHoleWarnings :: CG ()
emitConsolidatedHoleWarnings = do
  holes     <- gets hsHoles
  holeExprs <- gets hsHolesExprs
  mapAnfs   <- gets hsANFHoles
  binds'    <- gets binds
  let anfVars' = M.toList $ F.beBinds binds'
  let anfVars'' = map (\(_, (s, v, _)) -> (s, v)) anfVars'

  let  bindEnv = undoANF id
       $ HashMap.Lazy.filterWithKey (\sym _ -> F.anfPrefix
       ↪  `F.isPrefixOfSym` sym)
       $ HashMap.Lazy.unions $ map HashMap.Lazy.fromList
       ↪  [anfVars'']
  let mapAnfs' = M.fromList $ map (\(k, (v, _)) -> (F.symbol k,
  ↪  F.symbol v)) $ M.toList mapAnfs
  let mergedHoles
                 = [(h
                   , holeInfo
                   , M.findWithDefault [] (h, srcSpan) holeExprs
                   )
                   | ((h, srcSpan), holeInfo) <- M.toList holes
                   ]

  forM_ mergedHoles $ \(h, holeInfo, anfs) -> do
    let          = snd . info $ holeInfo
    let anfs'    = map (\(v, x, t) -> (F.symbol v, x,
    ↪  prettifySpecType t bindEnv mapAnfs')) anfs
    let ctx' = M.fromList [(x, dropQualifiers t) | (x, t) <-
    ↪  M.toList (reLocal $ renv  )]

    addWarning $ ErrHole (hloc holeInfo) "hole found" ctx' (F.symbol
    ↪  h) (htype holeInfo) anfs'

    prettifySpecType :: SpecType -> M.HashMap F.Symbol F.SortedReft
    ↪  -> M.HashMap F.Symbol F.Symbol -> SpecType
    prettifySpecType t anfs holes = mapReft undoANF' t
      where
        undoANF' :: RReft -> RReft
        undoANF' (MkUReft (F.Reft (v, e)) p) =
            let f = inlineInExpr (`HashMap.Lazy.lookup` anfs) in
            MkUReft { ur_reft = (F.Reft (v, undoANFExpr holes (f
            ↪  e))), ur_pred = p }
        undoANFExpr :: M.HashMap F.Symbol F.Symbol -> F.Expr ->
        ↪  F.Expr
```

```
    undoANFExpr anfMap expr =
      F.mapExpr substAnf expr
      where
        substAnf e@(F.EVar x) =
          case M.lookup x anfMap of
            Just e' -> undoANFExpr anfMap (F.EVar e')
            Nothing -> e
        substAnf e = e

-- | Drop qualifiers from refinement types to make them more
↪  readable
dropQualifiers :: SpecType -> SpecType
dropQualifiers = mapReft (dropQualifiersReft .
↪  simplifyANFVarReft)

dropQualifiersReft :: RReft -> RReft
dropQualifiersReft (MkUReft (F.Reft (v, e)) p) =
  MkUReft (F.Reft (v, dropQualifiersExpr e)) p

dropQualifiersExpr :: F.Expr -> F.Expr
dropQualifiersExpr = F.mapExpr simplifySymbol

simplifySymbol :: F.Expr -> F.Expr
simplifySymbol (F.EVar x) = F.EVar (simplifyName x)
simplifySymbol (F.EApp (F.EVar f) e) = F.EApp (F.EVar
↪  (simplifyName f)) e
simplifySymbol e = e

simplifyName :: F.Symbol -> F.Symbol
simplifyName s
  | "GHC.Types." `F.isPrefixOfSym` s = GM.dropModuleNames s
  | "GHC.Types_LHAssumptions." `F.isPrefixOfSym` s =
  ↪  GM.dropModuleNames s
  | "Data.List." `F.isPrefixOfSym` s = GM.dropModuleNames s
      | otherwise = s

simplifyANFVarReft :: RReft -> RReft
simplifyANFVarReft (MkUReft (F.Reft (v, e)) p) =
  MkUReft (F.Reft (simplifyANFVar v, simplifyANFExpr e)) p

simplifyANFVar :: F.Symbol -> F.Symbol
simplifyANFVar v
  | "lq_anf$" `F.isPrefixOfSym` v = F.symbol ("_anf" :: [Char])
  | "lq_tmp$" `F.isPrefixOfSym` v = F.symbol ("_tmp" :: [Char])
  | "##" `L.isInfixOf` F.symbolString v =
      let parts = L.Split.splitOn "##" (F.symbolString v)
      in if length parts > 1
```

```
            then F.symbol (head parts)
            else v
      | otherwise = v

simplifyANFExpr :: F.Expr -> F.Expr
simplifyANFExpr = F.mapExpr simplifyANFSymbol

simplifyANFSymbol :: F.Expr -> F.Expr
simplifyANFSymbol (F.EVar x) = F.EVar (simplifyANFVar x)
simplifyANFSymbol e = e
```

A.2 Set of Proof Examples

A.2.1 Example 0

```
{-@ LIQUID "--warn-on-term-holes" @-}

module Example0 where
    hole = undefined

    {-@ listLength :: xs:[a] -> {v : Nat | v == len xs} @-}
    listLength :: [a] -> Int
    listLength [] = 0
    listLength (_:xs) = 1 + hole
```

A.2.2 Example 1

```
{-@ LIQUID "--warn-on-term-holes" @-}

module Example1 where
    import Language.Haskell.Liquid.ProofCombinators (Proof)
    hole = undefined

    {-@ listLength :: xs:[a] -> {v : Nat | v == len xs} @-}
    listLength :: [a] -> Int
    listLength [] = 0
    listLength (_:xs) = 1 + listLength xs

    {-@ measure listLength @-}

    {-@ listLengthProof :: xs:[a] -> {listLength xs == len xs} @-}
    listLengthProof :: [a] -> Proof
    listLengthProof = hole
```

A.2.3 Example 2

```
{-@ LIQUID "--exact-data-cons" @-}
-- Based on
↪  https://ucsd-progsys.github.io/liquidhaskell-blog/2016/10/06/structural-induc

module Example2 where
    import Prelude hiding ((<>))
    import Language.Haskell.Liquid.ProofCombinators ((===), (***),
    ↪  QED(QED), Proof)

    hole = undefined

    {-@ reflect empty @-}
    empty  :: [a]
    empty  = []

    {-@ infix <> @-}
    {-@ reflect <> @-}
    (<>) :: [a] -> [a] -> [a]
    [] <> xs = xs
    (x:xs) <> ys = x : (xs <> ys)

    {-@ leftId  :: x:[a] -> { (empty <> x) == x } @-}
    leftId :: [a] -> Proof
    leftId x
        =   empty <> x
        === hole
        === x
        *** QED

    {-@ rightId  :: x:[a] -> { (x <> empty) == x } @-}
    rightId :: [a] -> Proof
    rightId x = hole

    {-@ assoc  :: x:[a] -> y:[a] -> z:[a] -> { (x <> (y <> z)) ==
    ↪  ((x <> y) <> z) } @-}
    assoc :: [a] -> [a] -> [a] -> Proof
    assoc x y z = hole
```

A.2.4 Example 3

```
{-@ LIQUID "--exact-data-cons" @-}
-- Based on paper: Theorem Proving for All: Equational Reasoning in
↪  Liquid Haskell (Functional Pearl)
```

```
module Example3 where
    import Prelude hiding ((<>), reverse, length, (++))
    import Language.Haskell.Liquid.ProofCombinators ((===), (***),
    ↪  QED(QED), Proof)

    hole = undefined

    {-@ length :: [a] -> {v:Int | 0 <= v } @-}
    length :: [a] -> Int
    length [] = 0
    length (_:xs) = 1 + length xs
    {-@ measure length @-}

    {-@ reverse :: is:[a] -> {os:[a] | length is == length os} @-}
    reverse :: [a] -> [a]
    reverse [] = []
    reverse (x:xs) = reverse xs ++ [x]

    {-@ (++) :: xs:[a] -> ys:[a] -> {zs:[a] | length zs == length
    ↪  xs + length ys} @-}
    (++) :: [a] -> [a] -> [a]
    [] ++ ys = ys
    (x:xs) ++ ys = x : (xs ++ ys)
    {-@ infixl ++ @-}

    {-@ reflect reverse @-}
    {-@ reflect ++ @-}

    --- Structural Induction will be needed. It could suggest as
    ↪  the next step.
    {-@ involutionProof :: xs:[a] -> { reverse (reverse xs) == xs }
    ↪  @-}
    involutionProof :: [a] -> Proof
    involutionProof xs = hole

    {-@ distributivityP :: xs:[a] -> ys:[a] -> { reverse (xs ++ ys)
    ↪  == reverse ys ++ reverse xs } @-}
    distributivityP :: [a] -> [a] -> Proof
    distributivityP xs ys = hole
```

A.2.5 Example 4

```
{-@ LIQUID "--exact-data-cons" @-}
-- Based on paper: Theorem Proving for All: Equational Reasoning in
↪  Liquid Haskell (Functional Pearl)
```

```
module Example4 where
    import Prelude hiding ((<>), reverse, length, (++))
    import Language.Haskell.Liquid.ProofCombinators ((===), (***),
    ↪  (?), QED(QED), Proof)

    hole = undefined
    {-@ length :: [a] -> {v:Int | 0 <= v } @-}
    length :: [a] -> Int
    length [] = 0
    length (_:xs) = 1 + length xs
    {-@ measure length @-}

    {-@ reverse :: is:[a] -> {os:[a] | length is == length os} @-}
    reverse :: [a] -> [a]
    reverse [] = []
    reverse (x:xs) = reverse xs ++ [x]

    {-@ (++) :: xs:[a] -> ys:[a] -> {zs:[a] | length zs == length
    ↪  xs + length ys} @-}
    (++) :: [a] -> [a] -> [a]
    [] ++ ys = ys
    (x:xs) ++ ys = x : (xs ++ ys)
    {-@ infixl ++ @-}

    {-@ reflect reverse @-}
    {-@ reflect ++ @-}

    {-@ reverseApp :: xs:[a] -> ys:[a] -> {zs:[a] | zs == reverse
    ↪  xs ++ ys} @-}
    reverseApp :: [a] -> [a] -> [a]
    reverseApp [] ys
        = reverse [] ++ ys
        === [] ++ ys
        === ys
    reverseApp (x:xs) ys
        = reverse (x:xs) ++ ys
        === (reverse xs ++ [x]) ++ ys
        === (reverse xs ++ [x] ++ ys) ? hole -- I need a lemma here!
        ↪  Can the hole help me?
        === reverse xs ++ ([x] ++ ys)
        -- It continues here following the
```

A.2.6 Example 5

```
{-@ LIQUID "--exact-data-cons" @-}
-- Based on paper: Theorem Proving for All: Equational Reasoning in
↪  Liquid Haskell (Functional Pearl)

module Example5 where
    import Prelude hiding ((<>), reverse, length, (++))
    import Language.Haskell.Liquid.ProofCombinators ((===), (***),
    ↪  (?), QED(QED), Proof)

    hole = undefined

    data Tree = Leaf Int | Node Tree Tree

    {-@ reflect flatten @-}
    flatten :: Tree -> [Int]
    flatten (Leaf x) = [x]
    flatten (Node l r) = flatten l ++ flatten r


    {-@ length :: [a] -> {v:Int | 0 <= v } @-}
    length :: [a] -> Int
    length [] = 0
    length (_:xs) = 1 + length xs
    {-@ measure length @-}

    {-@ (++) :: xs:[a] -> ys:[a] -> {zs:[a] | length zs == length
    ↪  xs + length ys} @-}
    (++) :: [a] -> [a] -> [a]
    [] ++ ys = ys
    (x:xs) ++ ys = x : (xs ++ ys)
    {-@ infixl ++ @-}
    {-@ reflect ++ @-}


    {-@ flattenApp :: t:Tree -> ns:[Int] -> { v:[Int] | v ==
    ↪  flatten t ++ ns } @-}
    flattenApp :: Tree -> [Int] -> [Int]
    flattenApp t ns = hole -- Structural Induction
```

A.2.7 Example 6

```
{-@ LIQUID "--exact-data-cons" @-}
-- Based on paper: Theorem Proving for All: Equational Reasoning in
↪  Liquid Haskell (Functional Pearl)
```

```
{-@ infix    :   @-}

module Example6 where
    import Prelude hiding ((<>), reverse, length, (++))
    import Language.Haskell.Liquid.ProofCombinators ((===), (***),
    ↪  (?), QED(QED), Proof)
    hole = undefined
    {-@ length :: [a] -> {v:Int | 0 <= v } @-}
    length :: [a] -> Int
    length [] = 0
    length (_:xs) = 1 + length xs
    {-@ measure length @-}

    {-@ (++) :: xs:[a] -> ys:[a] -> {zs:[a] | length zs == length
    ↪  xs + length ys} @-}
    (++) :: [a] -> [a] -> [a]
    [] ++ ys = ys
    (x:xs) ++ ys = x : (xs ++ ys)
    {-@ infixl ++ @-}
    {-@ reflect ++ @-}


    data Expr = Val Int |  Add Expr Expr

    {-@ reflect eval @-}
    eval :: Expr -> Int
    eval (Val n) = n
    eval (Add e1 e2) = eval e1 + eval e2

    type Stack = [Int]
    type Code  = [Op]
    data Op  = PUSH Int | ADD

    {-@ reflect exec @-}
    exec :: Code -> Stack -> Maybe Stack
    exec []           s = Just s
    exec (PUSH n : c) s = exec c (n:s)
    exec (ADD : c)    (m:n:s) = exec c (m+n:s)
    exec _ _            = Nothing

    {-@ reflect comp @-}
    comp :: Expr -> Code
    comp (Val n) = [PUSH n]
    comp (Add x y) = comp x ++ comp y ++ [ADD]

    {-@ generalizedCorrectness
```

```
      :: e:Expr -> s:Stack -> {exec (comp e) s == Just ((eval
      ↪  e):s) }
@-}
generalizedCorrectness :: Expr -> Stack -> Proof
generalizedCorrectness e s = hole -- Structural Induction
```