LiquidHaskell

Mehran Shahidi, Saba Safarnezhad
Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, Department of Computer Science

This report provides a brief overview of `LiquidHaskell`, a tool that extends Haskell with refinement types. Refinement types are types that extend the expressiveness of Haskell's type system by providing predicates that can specify invariants of the program. This report illustrates features of `LiquidHaskell` through a small formalization and demonstrates its application with several examples. Finally, we discuss its limitations and compare it with other tools.

1 Introduction

Two main trends in deductive verifiers are *Satisfiability Modulo Theory* (SMT)-based and *Typed-Theory* (TT)-based approaches. TT-based verifiers leverage type-level computation (normalization) to facilitate principled reasoning about terminating user-defined functions, whereas SMT-based verifiers use, among other tools, SMT solvers to check the satisfiability of universally-quanitified axioms—axioms that encode the semantics of user-defined functions within a specific theory (e.g., linear arithmetic, strings, sets, or bitvectors). Refinement types, and in particular the technique known as Refinement Reflection (see Section 3.3) combine the best of both worlds by fusing types with SMT based validity checking [10].

In this report, we focus on `LiquidHaskell`, a tool that extends Haskell with refinement types. After a short overview on refinement types and SMT solvers in section 2, we explain `LiquidHaskell`'s features in section 3. Then, in section 4, we provide an example of verifying `Insertion Sort`. Finally in section 5 we discuss the limitations of `LiquidHaskell` and compare it with other tools.

2 Overview

2.1 Refinement Types

Refinement types extend conventional type systems by attaching logical predicates to types. This allows for more precise type specifications and can potentially detect more errors at compile time [8].

Consider the following function:

```
divide :: Int -> Int -> Int
```

The standard type system ensures that the function `divide` takes two integers and returns an integer. For example, if we call `divide` with arguments of type `Bool`, the type system will show the error at compile time. However, it does not detect the error if the function is called with the second argument being zero.

In a refinement type system, we can define more precise types as follows:

```
type Pos = {v:Int | v > 0}
type Nat = {v:Int | v >= 0}
```

These are refinements of the basic `Int` type, where the logical predicates state that `v` is strictly positive (`Pos`) and non-negative (`Nat`), respectively. We can use these refinement types to annotate functions with preconditions and postconditions. For instance:

```
divide :: Nat -> Pos -> Int
```

This type signature specifies that the function `divide` takes a non-negative integer as its first argument and a positive integer as its second argument. Consequently, if we call `divide`, the type checker will verify if the specifications meet. As an instance, the following function is rejected by the type checker:

```
bad :: Nat -> Nat -> Int
bad x y = x `div` y
```

To be able to verify this, the refinement type system translates the annotation into a so-called *subtyping* query as follows [8]:

$$\begin{array}{c} x : \{\, x : \mathrm{Int} \mid x \geq 0\}, \\ y : \{\, y : \mathrm{Int} \mid y \geq 0\} \end{array} \vdash \ \{\, y : \mathrm{Int} \mid y \geq 0\} \preceq \ \{\, v : \mathrm{Int} \mid v > 0\}.$$

The notation $\Gamma \vdash \tau_1 \preceq \tau_2$ means that in the type environment Γ, τ_1 is a subtype of τ_2. The subtype query states, given the type environment in which x and y have type `Nat`, the type of y should be a subtype of `divide`'s second parameter v where v is a positive integer. Then the typing system translates this query into a verification condition (VC)- logical formulas whose validity ensures that the type specification is satisfied [8]. The translation of the subtyping query to VCs is shown in Figure 1. Based on this translation we would have the following VC:

$$(x \geq 0) \wedge (y \geq 0) \Rightarrow (v \geq 0) \Rightarrow (v > 0) \tag{1}$$

This VC is meant to express that, under the environment where x and y are non-negative, the property "if v is non-negative then v is strictly positive" must hold. This is unsatisfiable since 0 is non-negative but not strictly positive, which is what the verifier should detect for the `bad` function.

Refinement type systems are designed to exclude any arbitrary functions and only include formulas from decidable logics [8]. These VCs are then passed to an SMT solver to check their satisfiability. In this case, the SMT solver would reject the `bad` function as the VC is unsatisfiable. In the next section, we provide a brief introduction to SMT solvers and how they can be used in the context of `LiquidHaskell`.

$$
\begin{aligned}
(\!|\Gamma \vdash b_1 \preceq b_2|\!) \quad &\doteq (\!|\Gamma|\!) \Rightarrow (\!|b_1|\!) \Rightarrow (\!|b_2|\!) \\
(\!|\{x : \text{Int} \mid r\}|\!) \quad &\doteq r \\
(\!|x : \{v : \text{Int} \mid r\}|\!) \quad &\doteq \text{“x is a value”} \Rightarrow r[x/v] \\
(\!|x : (y : \tau_y \to \tau)|\!) \quad &\doteq \text{true} \\
(\!|x_1 : \tau_1, \dots, x_n : \tau_n|\!) \quad &\doteq (\!|x_1 : \tau_1|\!) \wedge \cdots \wedge (\!|x_n : \tau_n|\!)
\end{aligned}
$$

Figure 1: Notation: Translation to VCs [8]

2.2 SMT Solvers

SAT solvers are designed to determine the satisfiability of Boolean formulas [3]. For example, consider the following formula that is intended to be solved by SAT solvers::

$$\varphi = (x \vee y) \wedge (\neg x \vee z) \tag{2}$$

A SAT solver can check the satisfiability of the formula φ by checking if there is an assignment to the variables x, y, z such that the statement evaluates to *true*. For instance, the assignment $x = true, y = false, z = true$ satisfies the formula φ.

SMT solvers extend SAT solvers by incorporating additional theories—such as equality, integer arithmetic, real arithmetic, arrays, and lists—into Boolean logic [3]. As an example, consider the following formula that contains variables that require arithmetic theory:

$$x + y \leq 10 \quad \wedge \quad x = y - 7 \tag{3}$$

In the following section, we take a closer look at the Z3 SMT solver through some examples.

2.2.1 Applications and Example of Z3

Consider the satisfiability problem 4 involving three clauses. We aim to determine whether there exists an assignment of Boolean variables Tie and Shirt such that the conjunction of the following clauses holds:

$$(Tie \vee Shirt) \wedge (\neg Tie \vee Shirt) \wedge (\neg Tie \vee \neg Shirt) \tag{4}$$

Formula 4 can be solved in SMTLIB2 as the following code:

When we run this, Z3 responds:

```
sat
(model
  (define-fun Tie () Bool false)
  (define-fun Shirt () Bool true)
)
```

```
(set-logic QF_UF)
(declare-const Tie Bool)
(declare-const Shirt Bool)
(assert (or Tie Shirt))
(assert (or (not Tie) Shirt))
(assert (or (not Tie) (not Shirt)))
(check-sat)
(get-model)
```

This SMT-LIB2 script sets up the problem, declares the variables, asserts the constraints, checks for satisfiability, and retrieves the model, just like the Python code does for Formula 4 with z3 in the following example.

```python
from z3 import Bools, Solver, Or, Not
Tie, Shirt = Bools('Tie Shirt')
s = Solver()
s.add(Or(Tie, Shirt),
      Or(Not(Tie), Shirt),
      Or(Not(Tie), Not(Shirt)))
print(s.check())
print(s.model())
```

The output of the code is:

```
sat

[Tie = False, Shirt = True]
```

When calling s.check(), the solver determines that the assertions are satisfiable (sat)—meaning there is a way to assign values to the Tie and Shirt that make all the conditions true. One possible solution is Tie = false and Shirt = true, which can be retrieved using s.model().

The next example shows how Z3 reasons across multiple mathematical theories such as array theory, arithmetic, and uninterpreted functions. Z3's API analyzes the following Python snippet:

```python
Z = IntSort()
f = Function('f', Z, Z)
x, y, z = Ints('x y z')
A = Array('A', Z, Z)
fml = Implies(x + 2 == y, f(Store(A, x, 3)[y - 2]) == f(y - x + 1))
solve(Not(fml))

unsat
```

The integrated theories enabling this reasoning are:

- **Linear Integer Arithmetic (LIA)**: Handles integer constraints:

$$x + 2 = y \quad \text{and} \quad y - x + 1$$

- **Array Theory**: Manages array operations through `Store` and select operators:

$$\texttt{Store}(A, x, 3)[y - 2] \equiv \text{ite}(y - 2 = x, 3, A[y - 2])$$

  where ite denotes the if-then-else operator.

- **Uninterpreted Functions**: Treats function f as a black box respecting functional consistency:

$$\forall a, b : (a = b) \implies (f(a) = f(b))$$

This example illustrates Z3's theory combination mechanism, which:

- Ensures coherence across different mathematical domains
- Handles cross-theory constraints (e.g., array indices as arithmetic expressions)
- Enables verification of systems with mixed abstractions (memory, arithmetic, and black-box components)

This capability makes Z3 particularly useful for software verification, as real-world programs inherently integrate these concepts [5].

3 Working with LiquidHaskell

In this section, we will explain how `LiquidHaskell` works. `LiquidHaskell` is available as a GHC plugin. To use it, you need to add its dependencies to the cabal file as following [2]:

```
cabal-version: 1.12

name:           lh-plugin-demo
version:        0.1.0.0
...
...
  build-depends:
      liquid-prelude,
      liquid-vector,
      liquidhaskell,
      base,
      containers,
      vector
  default-language: Haskell2010
  ghc-options:  -fplugin=LiquidHaskell
```

With these dependencies, `LiquidHaskell` can check your program at compile time or through a code linter in your preferred IDE. Note that there are options such as `--reflection` and

`--ple`, which enable reflection and Proof by Logical Evaluation (PLE) in `LiquidHaskell`. You can either add them as plugin options in the Cabal file or use them directly in the source code as follows:

```
{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple" @-}
```

In the following sections, we will use both options, so we do not include them explicitly in the code snippets.

3.1 Type Refinement

Refinement types allow constraints to be placed on variables by adding predicates to their types [4]. For example, we can define natural numbers as follows:

```
{-@ type Nat = {v:Int | 0 <= v} @-}
```

Now if you configure your IDE to use Haskell LSP, it will show the following error if you try to assign a negative number to a variable of type Nat.

```
{-@ x :: Nat @-}
x = -1
>>> typecheck: Liquid Type Mismatch
  .
  The inferred type
    VV : {v : GHC.Types.Int | v == GHC.Num.negate (GHC.Types.I# 1)
                              && v == (-GHC.Types.I# 1)}
  .
  is not a subtype of the required type
    VV : {VV##493 : GHC.Types.Int | VV##493 >= 0}
  .
  Constraint id 2
```

The error message shows that the inferred type of the variable x is not a subtype of the required type.

Refinement types allow defining function preconditions and postconditions [4]. For example, consider the following function:

```
tail :: [a] -> [a]
tail (_:xs) = xs
tail [] = error "tail: empty list"
```

The function defined above is a partial function because it does not handle the case when the list is empty. Typical Haskell types only allow the introduction of the Maybe type, which postpones error handling to another part of the program [4]. Using refinement types, we can define the type of tail function as follows:

```
{-@ tail :: {v:[a] | 0 < len v} -> a @-}
tail :: [a] -> [a]
tail (x:_) = x
```

Now, our function is total, as it does not allow an empty list to be passed to `tail`.

```
x :: [Int]
x = tail (tail [1, 2])
```

When calling functions, `LiquidHaskell` won't look into the body of the function to see if the first application of the *tail* gives the valid non-empty list to the second *tail*. To allow `LiquidHaskell` consider the above example as safe, we need to also specify the post-condition for our function as following:

```
{-@ tail :: xs: {v:[a] | 0 < len v} -> {v:[a] | len v == len xs - 1} @-}
tail :: [a] -> [a]
tail (x:_) = x
```

3.2 Refined Data Types

In the above examples, we saw how refinements of input and output of function allow us to have stronger arguments about our program. We can take this further by refining the data types. Consider the following example [4]:

```
data Slist a = Slist { size :: Int, elems :: [a] }

{-@ data Slist a = Slist { size :: Nat, elems :: {v:[a] | len v == size} } @-}
```

This refined *Slist* data type ensures the stored ‘size‘ always matches the length of the ‘elems‘ list, as formalized in the refinement annotation [4]. This ensures that the size of the list is always correct.

The only thing that is missing is the definition of *len*. Fortunately, this function has already reflected by `LiquidHaskell`. In the following section, we show how can we use reflection or measure directives to define and execute any user-defined Haskell function in the refinement logic and reason about them.

3.3 Lifting Functions to the Refinement Logic

When our programs become more complex, we need to define our own functions in the refinement logic and reason about a function within another function. Refinement Reflection allows deep specification and verification by reflecting the code implementing a Haskell function into the function's output refinement type [7]. There are two ways to define and reason about a function in the refinement logic: `reflection` and `measure`.

`Measure` can be used on a function with one argument that is pattern-matched in the function body. Then, `LiquidHaskell` copies the function to the refinement logic, adds a refinement type to the constructor of the function's argument, and emits inferred global invariants related to the refinement [6]. Consider the following example:

```
data Bag a = Bag { toMap :: M.Map a Int } deriving Eq
{-@ measure bag @-}
{-@ bag :: Ord a => List a -> Bag a @-}
bag :: (Ord a) => List a -> Bag a
bag Nil = B.empty
bag (Cons x xs) = B.put x (bag xs)
```

This code adds the bag refinement type to the List data type. The `measure` directive is used to define the `bag` function, which is then copied to the refinement logic. It means that now the type of list constructors would have:

```
Nil  :: {v:List a | bag v = B.empty}
Cons :: x:a -> l:List a -> {v:List a | bag v = B.put x (bag l)}
```

So then we can use the `bag` function in the refinement logic to reason about the program. For instance, in the following example, we can use the `bag` function to reason about the program:

```
{-@ equalBagExample1 :: { bag(Cons 1 (Cons 3 Nil)) ==  bag( Cons 2 Nil) } @-}

>>     VV : {v : () | v == GHC.Tuple.Prim.()}
>>     .
>>     is not a subtype of the required type
>>       VV : {VV##2465 : () | bag (Cons 1 (Cons 3 Nil)) == bag (Cons 2 Nil)}
```

The $\{x = y\}$ is shorthand for $\{v : () \mid x = y\}$, where x and y are expressions. Note that equality for bags is defined as the equality of the underlying maps that already have a built-in equality function. `LiquidHaskell` can reason about the equality of bags by using the equality of the underlying maps and issuing a type error if the bags are not equal. If we define multiple measures for the same data type the refinements are conjoined together [6].

Reflection is another useful feature that allows the user to define a function in the refinement logic, providing the SMT solver with the function's behavior [10]. This has the advantage of allowing the user to define a function that is not pattern-matched in the function body. Additionally, with the use of a library of combinators provided by `LiquidHaskell`, we can leverage the existing programming constructs to prove the correctness of the program and use the principle of propositions-as-types (known as Curry-Howard isomorphism) [10] [11].

```
{-@ infixr ++ @-}
{-@ reflect ++ @-}

{-@ (++) :: xs:[a] -> ys:[a] -> { zs:[a] | len zs == len xs + len ys } @-}
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

The ++ function is defined in the refinement logic using the `reflect` directive. Now we can use the ++ function in the refinement logic to reason about the program. In the following subsection, we will show how to use `LiquidHaskell` to verify that the ++ function is associative.

3.4 Verification

`LiquidHaskell` allows structure proofs to follow the style of calculational or equational reasoning popularized in classic texts and implemented in proof assistants like Coq and Agda . It is equipped with a family of equation combinators for logical operators in the theory QF-UFLIA [10]. In the following example, we show how to use these combinators to verify that the ++ function is associative:

```
{-@ assoc :: xs:[a] -> ys:[a] -> zs:[a]
  -> { v: () | (xs ++ ys) ++ zs = xs ++ (ys ++ zs) } @-}
assoc :: [a] -> [a] -> [a] -> ()
assoc [] ys zs =
  ([] ++ ys)
    ++ zs
    === ys
    ++ zs
    === []
    ++ (ys ++ zs)
    *** QED

assoc (x : xs) ys zs =
  ((x : xs) ++ ys)
    ++ zs
    ===  x : (xs ++ ys) ++ zs
    === x
    : ((xs ++ ys) ++ zs)
      ? assoc xs ys zs
      === (x : xs)
      ++ (ys ++ zs)
      *** QED
```

These combinators are defined as follows:

```
  (===) :: x: a -> y: { a | x = y }  -> { v: a | v = x }
  data QED = QED
  (***) :: a -> QED -> ()
```

As you can see, some of the steps in the proof seem trivial if we are able to unfold the definition of the ++ function. For this purpose, `LiquidHaskell` provides `Proof by Logical Evaluation` (PLE) which allows us to unfold the definition of the function. The key idea in PLE is to mimic type-level computation within SMT-logics by representing the function in a guarded form and repeatedly unfolding function application terms by instantiating them with their definition corresponds to an enabled guard [10].

4 Example Application

In this section, we discuss the insertion sort algorithm and how to verify its functional correctness using `LiquidHaskell`. We take an intrinsic approach, leveraging refinement types so that we do not need to prove correctness separately. Insertion sort is a simple algorithm that builds a sorted list by inserting one element at a time. Using `LiquidHaskell`, we aim to ensure that the sorted list is both ordered and a permutation of the input.

4.1 Definition of Insertion Sort

Insertion sort is implemented in Haskell with two main components: the `insert` function, which places an element in its correct position in a sorted list, and the `insertSort` function, which recursively sorts the input list. Below is the Haskell implementation:

```haskell
{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple" @-}

module InsertionSort where

-- Define the List type
data List a = Nil | Cons a (List a) deriving (Eq, Show)

-- Insert operation
{-@ reflect insert @-}
insert :: (Ord a) => a -> List a -> List a
insert x Nil = Cons x Nil
insert x (Cons y ys)
  | x <= y    = Cons x (Cons y ys)
  | otherwise = Cons y (insert x ys)

-- Insertion Sort operation
{-@ reflect insertSort @-}
insertSort :: (Ord a) => List a -> List a
insertSort Nil = Nil
insertSort (Cons x xs) = insert x (insertSort xs)
```

4.2 Specification

To verify the correctness of the insertion sort, we define specifications that ensure the following: 1. The output list is sorted. 2. The output list is a permutation of the input list.

4.2.1 Sortedness Specification

We define a helper function, `isSorted`, to check whether a list is sorted:

```
{-@ reflect isSorted @-}
isSorted :: (Ord a) => List a -> Bool
isSorted Nil = True
isSorted (Cons x xs) =
  isSorted xs && case xs of
    Nil        -> True
    Cons x1 _  -> x <= x1
```

The `isSorted` function is then used to specify and verify the correctness of the `insert` and `insertSort` functions.

4.2.2 Insert Function Specification

The `insert` function places an element into a sorted list while maintaining its sortedness:

```
{-@ insert :: x:_ -> {xs:_ | isSorted xs} -> {ys:_ | isSorted ys} @-}
```

4.2.3 Insertion Sort Specification

The `insertSort` function ensures that the output is sorted and is a permutation of the input:

```
{-@ insertSort :: xs:_ -> {ys:_ | isSorted ys && bag xs == bag ys} @-}
```

Here, `bag` represents a multiset of elements, used to verify that the output is a permutation of the input.

4.3 Proofs

By incorporating the specifications into the insertion sort implementation, we can verify the correctness of the algorithm. Once we define the specification, we will realize that we need to prove the correctness of the `insert` function. As the specification is not enough for the `LiquidHaskell` to verify the correctness of the `insert` function, we need to prove the following lemma:

```
{-@ lem_ins :: y:_ -> {x:_ | y < x} -> {ys:_ | isSorted (Cons y ys)}
    -> {isSorted (Cons y (insert x ys))} @-}
lem_ins :: (Ord a) => a -> a -> List a -> Bool
lem_ins y x Nil = True
lem_ins y x (Cons y1 ys) = if y1 < x then lem_ins y1 x ys else True
```

This lemma ensures that the `insert` function maintains the sortedness property. Then with the use of `withProof`, we can use the lemma to prove the correctness of the `insert` function:

```
{-@ reflect insert @-}
{-@ insert :: x:_ -> {xs:_ | isSorted xs}
  -> {ys:_ | isSorted ys && Map_union (singelton x) (bag xs) == bag ys  } @-}
```

```
insert :: (Ord a) => a -> List a -> List a
insert x Nil = Cons x Nil
insert x (Cons y ys)
  | x <= y = Cons x (Cons y ys)
  | otherwise = Cons y (insert x ys) `withProof` lem_ins y x ys
```

4.3.1 Proof of Insertion Sort Correctness

The correctness of `insertSort` is established by combining the correctness of `insert` and ensuring that the output satisfies both the sortedness and permutation properties.

```
{-@ insertSort :: xs:_ -> {ys:_ | isSorted ys && bag xs == bag ys} @-}
insertSort :: (Ord a) => List a -> List a
insertSort Nil = Nil
insertSort (Cons x xs) = insert x (insertSort xs)
```

By verifying these properties using `LiquidHaskell`, we ensure that insertion sort is functionally correct and meets the desired specifications.

5 Conclusions, Results, Discussion

`LiquidHaskell`, enhanced with **Refinement Reflection** and **Proof by Logical Evaluation (PLE)**, combines the strengths of SMT-based and Type Theory (TT)-based verification approaches. It allows programmers to verify program correctness by leveraging a combination of refinement types, code reflection, and automated proof search [10].

Core Concepts

- **Refinement Types**: `LiquidHaskell` uses refinement types to specify program properties, extending basic types with logical predicates drawn from an SMT-decidable logic.
- **Refinement Reflection**: The implementation of a user-defined function is reflected in its output refinement type. This converts the function's type signature into a precise description of the function's behavior. At uses of the function, the definition is instantiated in the SMT logic [10].
- **Propositions as Types**: Proofs are written as regular Haskell programs, utilizing the Curry-Howard isomorphism. This allows programmers to use language mechanisms like branching, recursion, and auxiliary functions to construct proofs [10].
- **Proof by Logical Evaluation (PLE)**: PLE is a proof-search algorithm that automates equational reasoning. It mimics type-level computation within SMT-logics by representing functions in a guarded form and repeatedly unfolding function application terms by instantiating them with their definition corresponding to an enabled guard [10].

Comparison with Type Theory (TT) Based Systems

- **Type-Level Computation**: TT-based provers use type-level computation (normalization) for reasoning about user-defined functions, often requiring explicit lemmas or rewrite hints.
- **Automation**: `LiquidHaskell` uses PLE to automate equational reasoning without explicit lemmas, by emulating type-level computation within SMT logic.
- **Proof Style**: Proofs in `LiquidHaskell` are written as Haskell programs.
- **SMT Integration**: `LiquidHaskell` uses SMT solvers for decidable theories, while TT-based systems often require users to handle these proofs manually.
- **Expressiveness**: Both systems can express sophisticated proofs. `LiquidHaskell` is shown to be able to express any natural deduction proof [10].
- **Practicality**: `LiquidHaskell` reuses an existing language and its ecosystem, allowing proofs and programs to be written in the same language.

Comparison with Other SMT-Based Verifiers

- **Axiomatization**: Existing SMT-based verifiers such as Dafny use axioms to encode user-defined functions which can lead to incomplete verification and matching loops. `LiquidHaskell`, uses refinement reflection to encode functions, along with PLE for complete and terminating verification [10].
- **Fuel Parameter**: Dafny and F* use a fuel parameter to limit the instantiation of axioms which can lead to incompleteness [10]. PLE does not require any fuel parameter and is guaranteed to terminate.

Limitations

- **Debugging**: The increased automation can make it harder to debug failed proofs.
- **Interactivity**: `LiquidHaskell` lacks strong interactivity, unlike theorem provers with tactics and scripts.
- **Certificates**: It does not produce easily checkable certificates, unlike theorem provers [10].
- **Reflection Limitations**: Not all Haskell functions can be reflected into logic due to soundness or implementation constraints [10].

Conclusion

`LiquidHaskell` combines refinement types, code reflection, and PLE to offer a practical approach to program verification within an existing language. By leveraging SMT solvers for decidable theories and PLE to automate equational reasoning, `LiquidHaskell` aims to simplify the process of verifying program correctness when compared to other tools.

Acknowledgements

We would like to express our gratitude to the developers and maintainers of `LiquidHaskell`, whose research and documentation provided the foundation for this work.

Additionally, we acknowledge the use of AI-assisted tools throughout the preparation of this report. GitHub Copilot was employed for minor rewording and auto-completion in code snippets, helping streamline the coding process. For better comprehension of academic papers and extracting key information, we leveraged ChatGPT and NotebookLM. These tools assist in synthesizing complex concepts and structuring our explanations more effectively.

References

[1] *Software Verification.* Available at https://en.wikipedia.org/w/index.php?title=Software_verification&oldid=1262364177.

[2] *ucsd-progsys/lh-plugin-demo.* Available at https://github.com/ucsd-progsys/lh-plugin-demo. Original-date: 2020-06-24T23:10:49Z.

[3] Clark Barrett & Cesare Tinelli (2018): *Satisfiability Modulo Theories.* In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith & Roderick Bloem, editors: *Handbook of Model Checking*, Springer International Publishing, Cham, pp. 305–343, doi:10.1007/978-3-319-10575-8_11. Available at http://link.springer.com/10.1007/978-3-319-10575-8_11.

[4] Ranjit Jhala, Eric Seidel & Niki Vazou (2020): *Programming With Refinement Types.* Available at https://ucsd-progsys.github.io/liquidhaskell-tutorial/.

[5] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson & Christoph Wintersteiger: *Programming Z3.* Available at https://z3prover.github.io/papers/programmingz3.html.

[6] Niki Vazou: *Programming with Refinement Types Lecture.* Available at https://nikivazou.github.io/lh-course.

[7] Niki Vazou (2016): *Haskell as a Theorem Prover Blog.* Available at https://ucsd-progsys.github.io/liquidhaskell-blog/2016/09/18/refinement-reflection.lhs.

[8] Niki Vazou, Eric Seidel, Ranjit Jhala, Dimitrios Vytiniotis & Simon Peyton Jones (2014): *Refinement Types For Haskell. ACM SIGPLAN Notices* 49, doi:10.1145/2628136.2628161.

[9] Niki Vazou, Eric L. Seidel & Ranjit Jhala (2014): *LiquidHaskell: experience with refinement types in the real world.* In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, Haskell '14, Association for Computing Machinery, New York, NY, USA, pp. 39–51, doi:10.1145/2633357.2633366. Available at https://dl.acm.org/doi/10.1145/2633357.2633366.

[10] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler & Ranjit Jhala (2018): *Refinement reflection: complete verification with SMT. Proceedings of the ACM on Programming Languages* 2, pp. 1–31, doi:10.1145/3158141. Available at https://dl.acm.org/doi/10.1145/3158141.

[11] Philip Wadler (2015): *Propositions as types. Commun. ACM* 58(12), pp. 75–84, doi:10.1145/2699407. Available at https://dl.acm.org/doi/10.1145/2699407.