SEMINAR REPORT
Seminar: Programming Languages in Winter term 2024/2025

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-quantified 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 3.4) combine the best of both worlds by fusing types with SMT based validity checking [11].

In this report, we focus on `LiquidHaskell`, a tool that extends Haskell with refinement types. After a short overview of 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 [9].

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. For 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 [9]:

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

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. The type system then translates this query into a verification condition (VC)- logical formulas whose validity ensures that the type specification is satisfied [9]. 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 [9]. 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`.

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

Figure 1: Notation: Translation to VCs [9]

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 requiring arithmetic reasoning:

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

`LiquidHaskell` uses SMT solvers to check the satisfiability of verification conditions (VCs) generated from refinement types. In the following section, we take a closer look at the Z3 SMT solver through some illustrative examples.

2.2.1 Applications and Examples of Z3

Z3 is a powerful SMT solver equipped with specialized algorithms for solving background theories such as linear arithmetic, bit-vectors, and arrays. It allows users to express constraints using the SMT-LIB2 language, which is the standard input format for SMT solvers. Additionally, Z3 provides APIs for a variety of programming languages, including Python, C++, Haskell, and Java [5].

To use `LiquidHaskell`, at least one of the SMT solvers it supports must be installed—namely, Z3, CVC4, or MathSat. Among these, Z3 is the most thoroughly tested and widely used with `LiquidHaskell` [4].

To demonstrate Z3's capabilities, we will express and solve a simple satisfiability problem using both `SMT-LIB2` and Python. Consider the following SAT problem (Equation 4) involving three clauses. We aim to determine whether there exists an assignment of Boolean values to *Tie* and *Shirt* such that the formula holds:

$$(Tie \lor Shirt) \land (\neg Tie \lor Shirt) \land (\neg Tie \lor \neg Shirt) \tag{4}$$

This formula can be expressed in `SMT-LIB2` as follows:

```
(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. Alternatively, we can use Z3's Python API to express and solve the same problem programmatically:

```python
from z3 import *

Tie = Bool('Tie')
Shirt = Bool('Shirt')
clauses = [
    Or(Tie, Shirt),
    Or(Not(Tie), Shirt),
    Or(Not(Tie), Not(Shirt))
]

solver = Solver()
solver.add(clauses)

if solver.check() == sat:
    print("Satisfiable")
    print(solver.model())
else:
    print("Unsatisfiable")
```

When we run code fragment written in `SMT-LIB2`, Z3 responds:

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

When calling `solver.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 `solver.model()`.

2.2.2 Combining Theories in Z3

A key feature of modern SMT solvers such as Z3 is their ability to reason across multiple logical theories simultaneously. This capability is critical in software verification tasks, where properties of programs often involve arithmetic constraints, memory access patterns, and abstract functions. The following example demonstrates Z3's theory combination mechanism through a small but non-trivial constraint involving linear arithmetic, arrays, and uninterpreted functions [5].

```
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))
```

The code above encodes a formula in Z3's Python API and checks whether its negation is satisfiable. The solver returns `unsat`, indicating that the negated formula is not satisfiable, and therefore the original implication holds in all models. This deduction results from Z3's ability to simultaneously reason within and across several background theories.

First, the theory of *Linear Integer Arithmetic (LIA)* governs constraints such as $x+2=y$ and expressions like $y-x+1$, which appear in both the antecedent and consequent of the implication. This theory enables Z3 to reason about numeric relationships and preserve equality propagation across expressions.

Second, the formula involves an array operation: `Store`(A, x, 3)[y - 2]. In Z3's semantics, this corresponds to a functional update of array A at index x, followed by a read at index $y-2$. The semantics of the `Store` and `Select` operators are formally defined using a conditional expression:

$$\texttt{Store}(A, x, 3)[y-2] \equiv \begin{cases} 3 & \text{if } y-2=x \\ A[y-2] & \text{otherwise} \end{cases}$$

This is expressed in Z3 using the `ite` operator, short for *if-then-else*.

Finally, the function symbol f is treated as an *uninterpreted function*—that is, a function with no specific definition, but one which satisfies the axiom of functional congruence: for all a and b, if $a=b$ then $f(a)=f(b)$. This allows Z3 to perform symbolic reasoning on applications of f, relying solely on equality relationships between its arguments.

What makes this example particularly illustrative is that the solver must combine these theories coherently. The antecedent provides a numeric constraint $x+2=y$, which is used to simplify the expression $y-2=x$ and resolve the result of the array access. This index then determines the value passed to the function f, which is subsequently compared to another function application involving arithmetic. Through this process, Z3 propagates equalities, evaluates array expressions, and applies congruence rules for uninterpreted functions—eventually concluding that the equality in the implication must hold.

This example highlights Z3's ability to reason across multiple logical domains in a unified manner. By combining arithmetic reasoning, memory modeling via arrays, and abstraction through uninterpreted functions, Z3 is able to resolve complex constraints that arise in software analysis. Such theory combination is especially valuable in verifying real-world programs, where data structures, numeric computations, and abstract logic often interact in nontrivial ways.

In the following section, we explore how this kind of logical reasoning is applied in practice using `LiquidHaskell`. `LiquidHaskell` allows programmers to embed logical properties directly into type annotations, and automatically verifies them using the same kind of background theories we have just examined. We begin by showing how to set up `LiquidHaskell` in a Haskell project and introduce its key verification features.

3 Working with LiquidHaskell

In this section, we will explain how to work with `LiquidHaskell`. `LiquidHaskell` is available as a GHC plugin. To use it in your Haskell project, you need to add its dependencies to 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. In the following sections, we explore various features of `LiquidHaskell` and demonstrate how it can be used to verify programs.

3.1 Type Refinement

As mentioned in Section 2, refinement types extend the expressiveness of the type system by providing predicates that specify program invariants.

`LiquidHaskell` uses Logically Qualified Data Types (Liquid Types) [6], with some modifications to ensure soundness under lazy evaluation, to specify these refinements in Haskell [9].

It ensures that type checking remains decidable and efficient by generating verification conditions (VCs) that are both decidable and efficiently solvable by SMT solvers [9].

The Liquid type annotations are provided by the programmer in the source file as Haskell comments. Consider the following example:

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

If you configure your IDE to use the Haskell LSP, it will display the following error if you attempt 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)}
  .
  is not a subtype of the required type
    VV : {VV##493 : GHC.Types.Int | VV##493 >= 0}
  .
  Constraint id 2
```

The error message indicates that the inferred type of the variable `x` is not a subtype of the required type. `LiquidHaskell` employs liquid typing rules—including the subtyping rule discussed in Section 2—to infer these types.

Particularly, in `LiquidHaskell`, integers and other constants (e.g., booleans, characters, etc.) are given a singleton type, meaning a type that has only one value [7]. For instance, the liquid typing rule for integers is:

$$\frac{}{\Gamma \vdash i : \{Int \mid v = i\}} \quad (T - Int)$$

Combined with the following subtyping rule:

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 \preceq \tau_2}{\Gamma \vdash e : \tau_2} \quad (SUBTYPE)$$

The refinement type checker, based on the $T - Int$ rule, expects that the expression -1 should have type $\{Int \mid v = -1\}$ and then be a subtype of the Nat type (i.e., when assigning a value to a variable of type Nat). According to the translation of the subtyping query into verification conditions (see Figure 1), the type checker generates the following VC:

$$-1 = v \Rightarrow v \geq 0$$

which in turn the SMT solver finds unsatisfiable. In the following section, we demonstrate how to refine function types.

3.2 Function Types

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

```haskell
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 allow the introduction of the `Maybe` type to loosen the output type or using a stronger input type (e.g., `NonEmpty`) to make the function total. The former approach postpones handling to another part of the program [4], while the latter requires the user to handle the empty list case earlier in the program. Using refinement types, we can define the type of the `tail` function as follows:

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

This type definition specifies that the input list should have a length greater than zero. `len` is a special function defined in the refinement logic that returns the length of the list. In Section 3.4, we will show how to define and use user-defined functions in the refinement logic. `LiquidHaskell` will check this condition at compile time and issue an error if it is not met. Consider the following example:

```haskell
x = tail []
```

`LiquidHaskell` checks that the type

```
{v : [a] | v == GHC.Types.[ ] && len v == 0 && len v >= 0}
```

is not a subtype of the required type

```
{v : [a] | len v > 0}
```

and issues an error. However, in some cases, such as in the following, the refinement checker cannot verify the function call and gives an error:

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

When checking the function call, `LiquidHaskell` does not inspect the body of the function to verify that the first application of *tail* produces a non-empty list for the second *tail*. To allow `LiquidHaskell` to consider the above example as safe, we need to also specify the postcondition for our function as follows:

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

This way, `LiquidHaskell` can reason about the output of the function as well and provides additional information to the refinement type system.

3.3 Refined Data Types

In the previous 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. We use the following example as an illustration, following [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. This ensures that the size of the list is always correct.

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.

3.4 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 [8]. 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 [7]. 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 copies it to the logic, which is then copied to the refinement logic. We can now specify the following refinement types for the list constructors:

```
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:

```
{-@ 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 :() | 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 [7].

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 [11]. 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) [11] [12].

```
{-@ 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.

3.5 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 [11]. 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 [11].

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:

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

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

module InsertionSort where

data List a = Nil | Cons a (List a) deriving (Eq, Show)

{-@ 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)

{-@ 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 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:

```haskell
{-@ 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:

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

`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. ChatGPT was especially usefull finding grammatical errors and suggesting improvements in the text.

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] Patrick M. Rondon, Ming Kawaguci & Ranjit Jhala (2008): *Liquid Types*. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, Association for Computing Machinery, New York, NY, USA, pp. 159–169, doi:10.1145/1375581.1375602.

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

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

[9] 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.

[10] 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`.

[11] 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`.

[12] 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`.