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

Liquid Haskell

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

This report gives a brief overview of LiquidHaskell, a tool that extends Haskell with refinement types. Refinement types are types that extends expressiveness of Haskell types systems by providing predicates that can verify invarients of the program. This report explains briefly how SMT solvers leveraged by LiquidHaskell and how to use LiquidHaskell by providing some examples. Finally, we discuss the limitations of LiquidHaskell and compare it with other tools.

1 Introduction

Programming verfication is an important step in software developments. It is the process of verifying that a program behaves as it expected. There has been a lot of research in this area and many tools have been developed. Type safety is one of the important features of programming languages that helps to prevent runtime errors. Despite catching many errors at compile time, type systems are not powerful enough to catch all the errors. On the other, testing is another way to verify the program, but it is not always possible to test all the possible inputs. Consider the following example:

```haskell
average    :: [Int] -> Int
average xs = sum xs `div` length xs
```

The example above is a simple function that calculates the average of a list of integers. This can be a source of runtime error if the list is empty. While this can be caught by testing, it is not always possible to test all the possible inputs. Another way to verify the program is to use program verification tools. These tools use mathematical logic to check the program. One of such tools that is used in Haskell programming language is LiquidHaskell. LiquidHaskell (LH) extends Haskell with refinement types which are types that extend the expressiveness of Haskell. With refinement types, we can provide invariant that the program should satisfy [3].

In this report, after a short backgrouund on program verfication using SMT in section 2, we will explain how LH works and how it uses SMT solvers to verify the program in section 3. Then in section 4.4 we will provide some examples how to use LH to verify persistent stack. Finally in section 5 we discuss the limiations of LH and compare it with other tools.

2 Overview

2.1 Type Systems and Program Verification

Programming verification ensures that software behaves as expected by using formal methods. One common approach is type safety, which helps catch errors at compile time. However, traditional type systems in functional languages like Haskell often lack expressiveness to specify and enforce complex program properties statically [3]. While testing is another method for verification, it is impractical to cover all possible inputs, especially in large-scale applications.

2.2 Refinement Types

Refinement types extend conventional type systems by attaching logical predicates to types. This allows for more precise specifications of program invariants. For instance, in LiquidHaskell, a type annotation ensures that a function always receives a positive integer:

```haskell
{-@ incr :: Pos -> Pos @-}
incr :: Int -> Int
incr x = x + 1
```

Here, `Pos` is a refinement type that guarantees the argument and result are strictly positive integers. Unlike standard type systems, refinement types leverage logical constraints to statically verify program correctness [8].

2.3 SMT Solvers

Satisfiability Modulo Theories (SMT) solvers extend SAT solvers by incorporating various mathematical theories (e.g., arithmetic, arrays, lists). These solvers determine the satisfiability of logical formulas with complex constraints [2]. A common SMT solver used in verification is Z3, which supports multiple theories and decision procedures.

For example, given the constraints:

$$x + y \leq 10 \quad \text{and} \quad x = y - 7 \tag{1}$$

An SMT solver can determine whether a valid assignment exists for x and y.

2.4 LiquidHaskell and SMT

LiquidHaskell integrates refinement types with SMT solvers to perform automated verification. It encodes refinement predicates as logical formulas and delegates their satisfiability checking to an SMT solver like Z3 [8].

For example, a refined list function can ensure non-empty lists:

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

This ensures that `tail` is only applied to non-empty lists, preventing runtime errors. The SMT solver checks that all function applications adhere to their specifications, making LiquidHaskell a powerful tool for static verification.

2.5 SMT Solvers

SMT (Satisfiability Modulo Theories) solvers are tools that can check the satisfiability of logical formulas in a specific theory. SMT solvers extends the concept of SAT solvers by adding theories (e.g., the theory of equality, of integer numbers, of real numbers, of arrays, of lists, and so on) to the boolean logic [2]. While SAT solvers can only check the satisfiability of boolean formulas, SMT solvers can check the satisfiability of formulas that contain variables from different theories. As an example, consider the following formula:

$$\varphi = (x \lor y) \land (\neg x \lor z) \tag{2}$$

SAT solver can check the satisfiability of the formula φ by checking if there is an assignment for the variables x, y, z. For instance, $x = true, y = false, z = true$ is an assignment that makes φ *true*.

On the other hand, SMT solvers can check the satisfiability of formulas that contain variables that required arithmetic theory as following formula:

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

2.6 Z3 SMT Solvers

Overall System Architecture of Z3

The overall system architecture of Z3 is aimed at efficiently solving Satisfiability Modulo Theories (SMT) problems. Here's a detailed breakdown of the major components and their interactions:

2.6.1 Interfaces to Z3

Z3 can be interacted with using SMT-LIB2 scripts supplied as text files or through a pipe. Besides this, high-level programming languages can make API calls to Z3, which are proxies for calls over a C-based API [5].

2.6.2 Logical Formulas

Z3 accepts logical formulas built from atomic variables and logical connectives, which can include symbols defined by various theories. These formulas use basic building blocks like Boolean variables, integers, and reals, combined with logical operators such as `And`, `Or`, `Not`, `Implies`, and `Xor`. Z3 handles complex logical expressions by integrating symbols from multiple theories, such as arrays and arithmetic, allowing it to model a wide range of problems. The formulas generally follow the SMT-LIB2 standard, ensuring interoperability between different SMT solvers. This versatility makes Z3 a powerful tool for solving diverse logical problems efficiently [5].

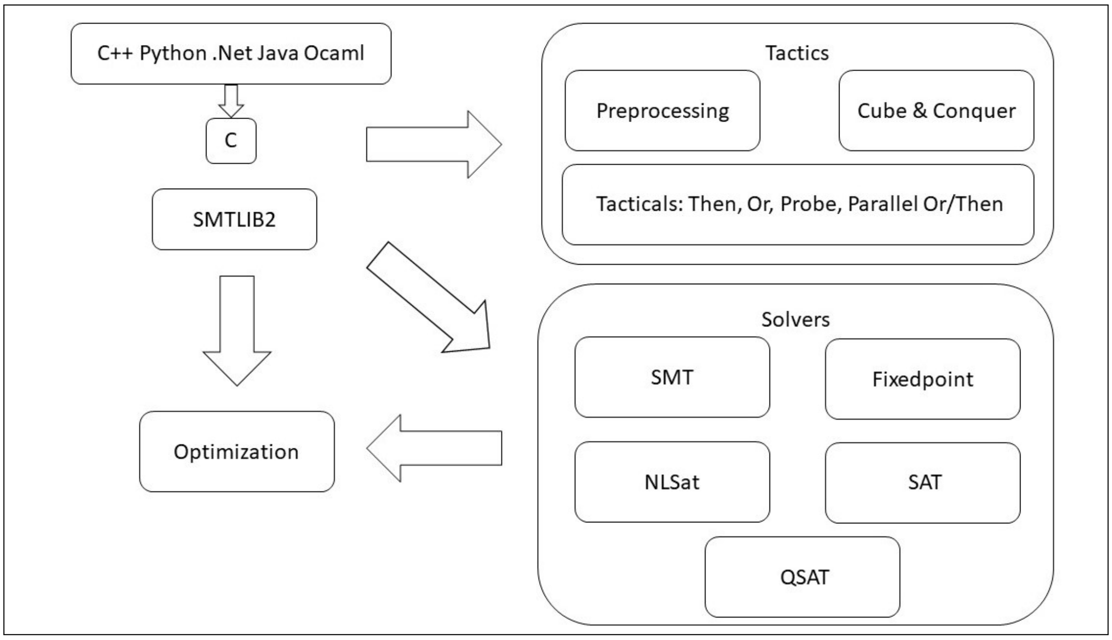

Figure 1: Overall system architecture of Z3 [5]

2.6.3 Theories

Z3 supports multiple theories, including Equality and Uninterpreted Functions (EUF), Arithmetic (both linear and non-linear), Arrays, Bit-Vectors, Algebraic Datatypes, Sequences, and Strings. Each theory has specialized decision procedures for solving related formulas [5].

2.6.4 Solver

Z3 provides services for deciding the satisfiability of formulas. This includes handling incrementality, scopes, assumptions, cores, models, and other methods like statistics, proofs, and retrieving solver state. Specialized solvers are included for different types of problems [5].

SMT Solver

The SMT solver in Z3 is a general-purpose solver that integrates various theories to decide the satisfiability of logical formulas. It uses the CDCL(T) architecture, which combines Conflict-Driven Clause Learning (CDCL) with theory solvers [5].

Fixedpoint Solver

The Fixedpoint solver in Z3 is used for reasoning about recursive definitions and fixed-point computations. It includes a Datalog engine and supports relational algebra and Property Directed Reachability (PDR) algorithms [5]

NLSat Solver

The NLSat solver is specialized for non-linear arithmetic problems. It uses a combination of algebraic methods and SAT solving techniques to handle polynomial constraints [5].

SAT Solver

The SAT solver in Z3 is designed for propositional logic problems. It uses advanced techniques like in-processing and co-processing to efficiently solve Boolean satisfiability problems [5].

QSAT Solver

The QSAT solver handles quantified Boolean formulas (QBF). It extends the capabilities of the SAT solver to deal with quantifiers, providing solutions for more complex logical problems [5].

2.6.5 Tactics

Tactics in Z3 are used for preprocessing, formula simplification, and the generation of subgoals. they are essential to decompose complex problems into relatively simple fragments. The preprocess tactics simplify the input formulae before they are fed into the main solver. It can include heuristics such as the propagation of constants, arithmetic simplification, and removal of redundant constraints. The Cube and Conquer tactic is used to split the search space into smaller sub-problems. (cubes) that can be solved independently. This is a very useful scheme for parallel solving and the sum time required to solve, it greatly shortens. Tacticals are combiners which make composition of multiple tactics [5].

2.6.6 Optimization

Z3 allows the use of optimization services along with user-defined problems in order to study the satisfiability of the problem while maximizing or minimizing objective functions. This applies to those problems which require solutions that are to be optimized under given constraints [5].

2.6.7 Applications and Example of Z3

Software analysis verification and symbolic execution along with software analysis verification rely heavily on making use of the Z3 platform. Z3 operates through SMT-LIB2 scripts for textual logical problem inputs as well as program application interfaces using Python fundamental sources and C++ constructs and OCaml programming code. Both Z3 users prefer the Python API because of its straightforward nature and SMT-LIB2 stands out for presenting logical problems with predefined logics including arithmetic, arrays and bit-vectors.

To be sure, consider 4 and its satisfiability problem, we are wondering is there a value assignment of Boolean variables Tie, Shirt, such that three clauses conjuction :

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

```
   (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)
```

Formula 4 can be solved in SMTLIB2 as following code:
When we run this, Z3 responds:

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

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.

```
  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 Tie and Shirt that makes 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:

```
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
```

Theories Involved 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

LH is a tool that extends Haskell with refinement types. Refinement types are types that extend the expressiveness of Haskell types systems by providing predicates that can verify invarients of the program [8]. In this section, we will explain how to use LH and how it works. LH is available as a GHC plugin. To use it, you need to add its dependecies to the cabal file as following [1]:

```
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
```

By adding this dependency, LH can now check your program at compile time or via code linter in your IDE of choice. In following sections, we will explain how to use LH to verify the program.

3.1 Type Refinement

Refinement types allows to constrain the type of the variables by adding predicates to the types [3]. For example we can define natural numbers as following:

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

Now if you configure your IDE to use Haskell LSP, it will show 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.

Using refinement types, one can define pre-conditions and post-conditions of the functions [3]. 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. Typicall haskell types only allow to introduce the *Maybe* type which postpone the handling of error to the other part of the program [3]. Using refinement types, we can define the type of tail function as following:

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

Now our function is a total function as it doesn't allow non-empty list to be passed to *tail*. However, it can't check the following case.

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

When calling functions, LH won't look into the body of function to see if the first application of *tail* gives the valid non-empty list to the second *tail*. To allow LH consider the above example as safe, we need to also specify 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 fucntion allows us to have stronger arguments about our program. We can take this further by refining the data types. Consider the following example [3]:

```
 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 stoed ‘size‘ always matches the length of the ‘elems‘ list, as formalized in the refinement annotation [3] . 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 LH. 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, the LH 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. 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 Emp = B.empty
bag (Cons x xs) = B.put x (bag xs)
```

This code add 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 following example, we can use the `bag` function to reason about the program:

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

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

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. LH 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 ar 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 the LH, 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 LH to verify that ++ function is associative.

3.4 Verification

LH allows to structure proofs to follow the style of calculational or equational reasoning popularized in classic texts and implemented in proof assistants like Coq and Agda . LH is equiped with a family of equation combinators for logical operators in the theory QF-UFLIA [10]. In the following example, we show how to use LH 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 defines as following:

```
  (===) :: 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, LH 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 corresponding 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 the intrinsic approach to leverage the refinement types so that we don't need to proof the correctness separately. Insertion sort is a simple sorting algorithm that builds the final sorted list 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 = Emp | Cons a (List a) deriving (Eq, Show)

-- Insert operation
{-@ reflect insert @-}
insert :: (Ord a) => a -> List a -> List a
insert x Emp = Cons x Emp
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 Emp = Emp
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 Emp = True
isSorted (Cons x xs) =
  isSorted xs && case xs of
    Emp        -> 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 LH 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 Emp = 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 Emp = Cons x Emp
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 Emp = Emp
insertSort (Cons x xs) = insert x (insertSort xs)
```

By verifying these properties using LiquidHaskell, we ensure that the implementation of insertion sort is functionally correct and adheres to the desired specifications.

4.4 Conclusion

To summarize, we use the proposition-as-types principle to leverage LiquidHaskell for program verification. Haskell's expressive type system, combined with LiquidHaskell's refinement types, allows for precise specification of program invariants.

5 Conclusions, Results, Discussion

Liquid Haskell, enhanced with **Refinement Reflection** and **Proof by Logical Evaluation (PLE)**, is a verification system that aims to combine the strengths of **SMT-based** and **Type Theory (TT)-based** 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**: Liquid Haskell 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 into 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 (e.g., Agda)

- **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**: Liquid Haskell uses PLE to automate equational reasoning without explicit lemmas, by emulating type-level computation within SMT logic.
- **Proof Style**: Proofs in Liquid Haskell are written as Haskell programs.
- **SMT Integration**: Liquid Haskell uses SMT solvers for decidable theories, while TT-based systems often require users to handle these proofs manually [5, 9].
- **Expressiveness**: Both systems can express sophisticated proofs. Liquid Haskell is shown to be able to express any natural deduction proof [10].
- **Practicality**: Liquid Haskell reuses an existing language and its ecosystem, allowing proofs and programs to be written in the same language.

Comparison with Other SMT-Based Verifiers (e.g., Dafny)

- **Axiomatization**: Existing SMT-based verifiers such as Dafny use axioms to encode user defined functions which can lead to incomplete verification and matching loops. Liquid Haskell, 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**: Liquid Haskell 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

Liquid Haskell 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, Liquid Haskell aims to simplify the process of verifying program correctness when compared to other tools.

References

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

[2] 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`.

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

[4] Adithya Murali, Lucas Peña, Ranjit Jhala & P. Madhusudan (2023): *Complete First-Order Reasoning for Properties of Functional Programs. Proceedings of the ACM on Programming Languages* 7(OOPSLA2), pp. 1063–1092, doi:10.1145/3622835. Available at `https://dl.acm.org/doi/10.1145/3622835`.

[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`.