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:

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

One of the 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.

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 we will provide some examples how to use LH to verify **problem name** problem. Finally in section 5 we discuss the limiations of LH and compare it with other tools.

problem name

2 Background

Refinement Types Refinement types are types that extend the expressiveness of Haskell types by providing predicates that can verify invariants of the program.

Predicate Predicates are haskell expressions that evaluate to boolean.

SMT Solvers SMT solvers are used to check the satisfiability of the predicates.

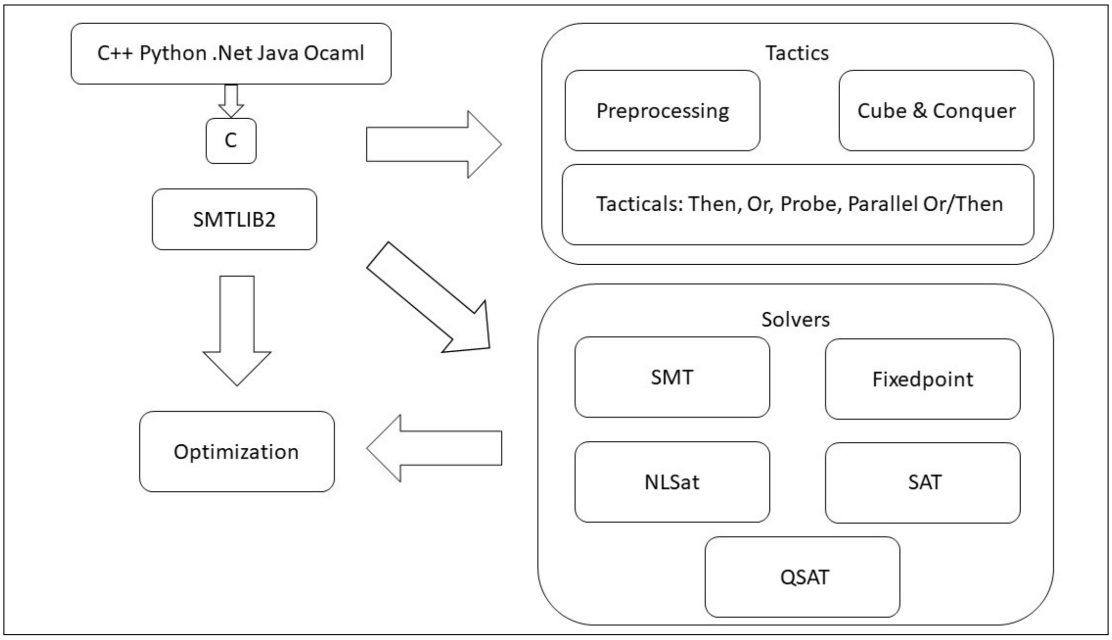

Figure 1: Overall system architecture of Z3 [4]

$\{x \mid \varphi(x)\}$

2.1 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 [1]. 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:

$$\varphi = (x \vee y) \wedge (\neg x \vee z) \tag{1}$$

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: Figure 1

$$x + y \leq 10 \quad and \quad x = y - 7 \tag{2}$$

Figure 2: A simple example of LiquidHaskell. in line 5 we define the type refinement.

2.2 Z3 SMT Solvers

3 Working with LiquidHaskell

```
p := (e r e)                  -- binary relation
   | (v e1 e2 ... en)         -- predicate (or alias) application
   | (p && p)                 -- and
   | (p || p)                 -- or
   | (p => p) | (p ==> p)     -- implies
   | (p <=> p)                -- iff
   | (not p)                  -- negation
   | true | True
   | false | False
```

```
1 {-# OPTIONS_GHC -fplugin=LiquidHaskell #-}

3 {-@ type Pos = {v:Int | 0 < v} @-}

5 {-@ incr :: Pos -> Pos @-}
6 incr :: Int -> Int
7 incr x = x + 1
```

4 Example Application

5 Conclusions, Results, Discussion

References

[1] 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`.

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

[3] 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`.

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

[5] P.D. Magnus, Tim Button, Robert Trueman & Richard Zach (2023): *forall x: Calgary.* Available at `https://forallx.openlogicproject.org/html/`.

[6] 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`.