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

Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

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
```

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.

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 8 we will provide some examples how to use LH to verify persistent stack. Finally in section 9 we discuss the limiations of LH and compare it with other tools.

2 Background

Refinement Types Refinement types add predicates to the types [2]. For example, consider the following type:

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

Predicate Predicates are haskell expressions that evaluate to boolean.

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

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

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

2.2 Z3 SMT Solvers

Overall System Architecture of Z3

The overall system architecture of Z3 is designed to efficiently solve Satisfiability Modulo Theories (SMT) problems. Here's a detailed breakdown of the key components and their interactions:

2.2.1 Interfaces to Z3

Z3 can be interacted with using SMT-LIB2 scripts supplied as text files or through a pipe. Additionally, high-level programming languages can make API calls to Z3, which are proxies for calls over a C-based API. The tutorial focuses on using the Python front-end for interfacing with Z3.

2.2.2 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.

2.2.3 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.

2.2.4 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

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.

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.

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.

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.

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.

2.2.5 Tactics

Tactics in Z3 are used for pre-processing, simplifying formulas, and creating sub-goals. They are essential for breaking down complex problems into more manageable parts.

Preprocessing

Preprocessing tactics simplify the input formulas before they are passed to the main solver. This can include techniques like constant propagation, simplification of arithmetic expressions, and elimination of redundant constraints.

Cube and Conquer

The Cube and Conquer tactic is used to partition the search space into smaller sub-problems (cubes) that can be solved independently. This approach is particularly useful for parallel solving and can significantly reduce the overall solving time.

tacticals

Tacticals are combinators that allow the composition of multiple tactics. Examples include:

- `then`: Applies a sequence of tactics to the input goal.
- `par-then`: Applies tactics in parallel to the input goal.
- `or-else`: Applies the first tactic and, if it fails, applies the second tactic.
- `repeat`: Repeatedly applies a tactic until no subgoal is modified.

The architecture of Z3 is designed to be flexible and powerful, supporting a wide range of logical theories and providing robust solver services. It allows for efficient interaction through various interfaces and includes advanced features for optimization and logical analysis.

2.2.6 Optimization

Z3 provides optimization services that allow users to solve satisfiability problems while maximizing or minimizing objective functions. This is useful for problems that require finding the best solution under given constraints.

2.2.7 Advanced Features

Z3 includes specialized procedures for enumerating consequences (backbone literals), which are useful for certain types of logical analysis.

The architecture of Z3 is designed to be flexible and powerful, supporting a wide range of logical theories and providing robust solver services. It allows for efficient interaction through various interfaces and includes advanced features for optimization and logical analysis.

This LaTeX code will format the text with a section heading, a list of examples, and a concluding paragraph, making it clear and well-structured.

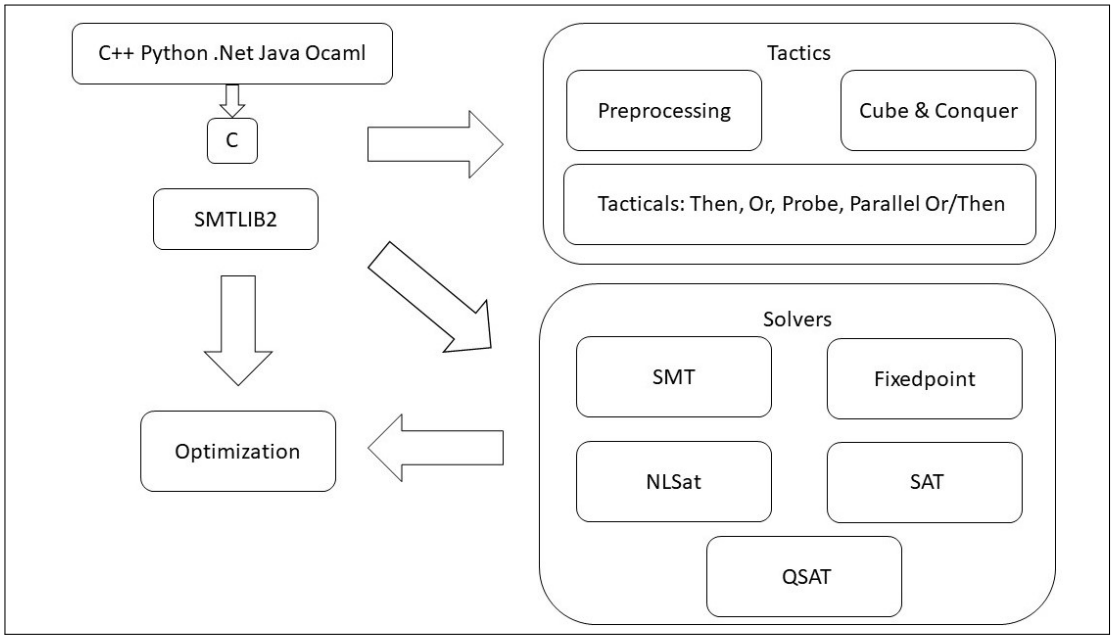

Figure 1: Overall system architecture of Z3 [4]

Consider following example:

$$(Tie \vee Shirt) \wedge (\neg Tie \vee Shirt) \wedge (\neg Tie \vee \neg Shirt) \tag{3}$$

Formula 3 can be solved in SMTLIB2 as following code:

```
(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 script will produce output of the code which is:

```
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 with Z in the following example.

Formula 3 can also be solved by Z3 SMT solver as following code:

```
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]
```

```
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))
print(s.check(Not(fml)))
```

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. In this section, we will explain how to use LH and how it works. LH is available as a GHC plugin. To use LH, you need to add LH dependecy to the cabal file as following:

```
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 [2]. 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 [2]. 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 [2]. 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:

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

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

In the above example, we introduced a new data type *Slist* which is a list with a size. We also added a refinement to the data type that ensures that the size of the list is equal to the length of the list. This ensures that the size of the list is always correct.

The only thing in above example, 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 capabilities of LH to define and execute our own functions in the refinement logic and reason about them.

3.3 Lifting Functions to the Refinement Logic

When our program become more complex, we need to define our own functions in the refinement logic and reason about a function in another function. In LH, we can define our own functions in the refinement logic using reflection. There are two ways to define a function in the refinement logic: reflection and measure. In this section, we will explain how to use reflection to define a function in the refinement logic.

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

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

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

3.4 Verification

4 Example Application

In this section, we discuss the persistent stack data structure and how to verify its functional correctness using LiquidHaskell. Persistent data structures are immutable and support efficient access to previous versions. In this paper, we aim to prove the functional correctness of a persistent stack implementation, ensuring its operations satisfy the expected behavior.

5 Definition of the Persistent Stack

The stack is defined as a list-based data structure with two main operations: `push` and `pop`. Below is the Haskell implementation:

Listing 1: Persistent Stack Implementation

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

module PersistentStack where

-- Define the Stack type
data Stack a = Empty | Cons a (Stack a) deriving (Eq, Show)

-- Push operation
{-@ reflect push @-}
push :: a -> Stack a -> Stack a
push x s = Cons x s

-- Pop operation
{-@ reflect pop @-}
pop :: Stack a -> Maybe (a, Stack a)
pop Empty = Nothing
pop (Cons x s) = Just (x, s)

-- Top operation
{-@ reflect top @-}
top :: Stack a -> Maybe a
top Empty = Nothing
top (Cons x _) = Just x

-- IsEmpty operation
```

```
{-@ reflect isEmpty @-}
isEmpty :: Stack a -> Bool
isEmpty Empty = True
isEmpty _     = False
```

6 Specification

To prove the correctness of the `push` and `pop` operations, we define their specifications in LiquidHaskell.

6.1 Push Operation

The `push` operation adds an element to the stack. Its specification ensures that the new stack is not empty and that the top of the stack is the newly pushed element:

Listing 2: Push Specification

```
{-@ pushSpec :: x:a -> s:Stack a
  -> { top (push x s) == Just x && not (isEmpty (push x s)) } @-}
```

6.2 Pop Operation

The `pop` operation removes the top element from the stack. Its specification ensures that if the stack is non-empty, `pop` returns the correct element and the remainder of the stack:

Listing 3: Pop Specification

```
{-@ popSpec :: s:Stack a -> { isEmpty s || case pop s of
    Nothing -> True;
    Just (x, s') -> top s == Just x && top (push x s') == Just x } @-}
```

7 Proofs

Using LiquidHaskell, we can prove the above specifications. Here, we outline the proof strategies for the main operations:

7.1 Proof of Push Correctness

The correctness of `push` follows directly from its definition:

pushSpec: Let s = Empty or Cons.
By definition of `push`, `push x s` = Cons xs.
Then `top (push x s)` = Just x.
Also, `isEmpty (push x s)` = False.

7.2 Proof of Pop Correctness

The correctness of `pop` depends on the case analysis of the stack:

- Case `s = Empty`: `pop s = Nothing`, satisfying the specification.
- Case `s = Cons x s'`: `pop s = Just (x, s')`, ensuring `top s = Just x`.

8 Conclusion

This paper demonstrates the use of LiquidHaskell to verify the functional correctness of a persistent stack. Future work includes extending this methodology to more complex data structures such as queues and deques.

9 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`.

[7] 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`.