NIKI VAZOU

PROGRAMMING WITH REFINEMENT TYPES

AN INTRODUCTION TO LIQUIDHASKELL

Version 13, March 15th, 2024.

Copyright © 2024 Niki Vazou

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Contents

1 *Refinement Types* 7

Installation 8

Basic Refinement Types 8

Subtyping 9

From Subtyping to Verification Conditions 9

Verification Conditions 10

Primitive Operations 12

Function Types 13

Branching and Recursion 15

Refined Polymorphism 16

Putting it all together: Safe Indexing 17

Summary 18

Further Reading 20

Cheatsheet 20

2 *Data Types* 21

3 *Case Study: Insertion Sort* 23

4 *Abstract Refinement Types* 25

5 *Theorem Proving* 27

6 *Data Propositions* 29

List of Exercises

1

Refinement Types

Refinement types are types refined with logical predicates that enforce a variety of invariants at compile time. In this course, we will learn the refinement type system of Liquid Haskell.

If you follow this course via a brouser, you can just click the check button that exists on the code spinnets to run Liquid Haskell on your file. If you follow it on an editor, then compile the code using the Haskell compiler and turn on the 'Liquid Haskell plugin", but uncommenting the following line:

```
{- OPTIONS_GHC -fplugin=LiquidHaskell #-}
{-@ LIQUID "--no-termination" @-}
module Lecture_01_RefinementTypes where

main :: IO ()
main = return ()
```

Either way, you can now use the `Liquid Haskell` type checker, for example to check that division by zero is not possible.

```
test :: Int -> Int
test x = 42 `div` 2
```

If we call `div` with zero, directly or even indirectly via the `x` argument, then at runtime we will get a division by zero error.

```
ghci> test 0
*** Exception: divide by zero
```

Liquid Haskell comes with a refined type for the division operator that specified that the second argument must be non-zero.

```
div :: Int -> {v:Int | v /= 0} -> Int
```

The above type specifies that the second argument must be non-zero and is automatically checked at compile time, using an SMT solver. Today, we will learn how these checks are performed, and how to write and use refined types in Haskell.

Installation

Liquid Haskell exists on hackage, which is the Haskell package repository, so you can install it using `cabal`, `stack`, or instal it from source:

- `cabal install liquidhaskell` will install the Liquid Haskell plugin that you can use in your Haskell projects.
- Source installation will let you clone Liquid Haskell from github and install it.

The source code of these class notes can also be downloaded and executed from github[1]

[1] These notes are adjusted from the Liquid Haskell Tutorial.

Note: This is the first time I am giving these lectures, so I would appreciate any feedback you might have, via the virtual classroom, pull requests, or email at `niki.vazou@imdea.org`.

Basic Refinement Types

Did you note that `2` is a good argument for the division operator?

But, what is the type of `2`?

In Haskell `2:Int`, but the same value can have many different refinement types. A basic refinement type has the form

$$\{v : b \mid p\}$$

where b is the base type (e.g., `Int`, `Bool`, etc.) and p is a logical predicate.

For example, the logical predicate that refines the type of `2` can have many different forms:

```
{-@ type Two      = {v:Int | v == 2}  @-}
{-@ type FortyTwo = {v:Int | v == 42} @-}
{-@ type NZero    = {v:Int | v /= 0}  @-}
{-@ type Pos      = {v:Int | v >  0}  @-}
{-@ type Neg      = {v:Int | v <  0}  @-}
{-@ type Nat      = {v:Int | 0 <= v}  @-}


two :: Int
two = 2
```

Question: What are good types for `two`?

Question: Can you find more types for `two`?

Subtyping

So, what is the type of `2`? In Liquid Haskell, integers and other constants, e.g., booleans, characters, etc., are given a *singleton type*, meaning a type that has only one value. So, the typing rule for integers is:

$$\frac{}{\Gamma \vdash i : \{\texttt{Int} \mid v = i\}}\ \texttt{T-Int}$$

In an unrefined system this would be *the only* type for `2`. But refinement types have the notion of *subtyping*, which can give an expression many different types.

The rule for subtyping is the following:

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 \preceq \tau_2}{\Gamma \vdash e : \tau_2}\ \texttt{T-Sub}$$

This rule states that in an expression, like 2, that has type τ_1 and τ_1 is a subtype of τ_2, then 2 can also have type τ_2.

So, $2 : \{\texttt{Int} \mid 0 \leq v\}$, because $\{\texttt{Int} \mid v = 2\} \preceq \{\texttt{Int} \mid 0 \leq v\}$. But, let's see how subtyping is decided.

From Subtyping to Verification Conditions

The subtyping rule for base types, e.g., integers, booleans, etc., is the following:

$$\frac{\texttt{Sub-Base}}{\Gamma \vdash \forall v : b.p_1 \Rightarrow p_2} \; \Gamma \vdash \{v : \texttt{b} \mid p_1\} \preceq \{v : \texttt{b} \mid p_2\}$$

Meaning that $\{v : \texttt{b} \mid p_1\} \preceq \{v : \texttt{b} \mid p_2\}$, if p_1 "implies" p_2 for all values v of type b. To check implications, we use the below two implication rules that are based on the SMT solver[2].

[2] SMT stands for Satisfiability Modulo Theories and are automated tools that check the satisfiability of logical formulas. Known SMT solvers are Z_3, CVC_5, etc.

$$\frac{\texttt{I-Emp}}{\texttt{SmtValid}(c)} \; \emptyset \vdash c \qquad \frac{\texttt{I-Ext}}{\Gamma \vdash \forall x.p \Rightarrow c} \; \Gamma, x : \{x : \texttt{b} \mid p\} \vdash c$$

Example: When we check that 2 is indeed a natural number, the following derivation takes place

$$\frac{\underline{\texttt{T-Int}} \; \emptyset \vdash 2 : \{v : \texttt{Int} \mid 2 = v\} \qquad \frac{\texttt{T-Sub}}{\frac{\texttt{Sub-Base}}{\frac{\texttt{I-Emp}}{\texttt{SmtValid}(\forall v{:}\texttt{Int}.2{=}v{\Rightarrow}0{\leq}v)} \; \emptyset{\vdash}\forall v{:}\texttt{Int}.2{=}v{\Rightarrow}0{\leq}v} \; \emptyset \vdash \{v : \texttt{Int} \mid 2 = v\} \preceq \{v : \texttt{Int} \mid 0 \leq v\}} \; \emptyset \vdash 2 : \{v : \texttt{In}}$$

So, the type derivation succeeds, because the SMT can indeed decide that the implication $2 = v \Rightarrow 0 \leq v$ is valid.

In general, we call such implications *verification conditions* and the main task of a refinement type checker is to reduce the type checking problem to validity of verification conditions.

Verification Conditions

Liquid Haskell takes great care to ensure that type checking is decidable and efficient. To achieve this, it has to be careful to generate verification conditions that are decidable and efficiently checkable by the SMT solver. The rules `Sub-Base` and `I-Ext`, presented above, are the main rules that generate verification conditions from the predicates found in the refinement types. Below is the syntax of the predicates and verification conditions:

Predicates	p	$::=$	x, y, z	*variables*
		$\mid$	$\text{true}, \text{false}$	*booleans*
		$\mid$	$0, -1, 1, \ldots$	*numbers*
		$\mid$	$\neg p, p_1 \wedge p_2, p_1 \vee p_2$	*boolean operators*
		$\mid$	$p_1 = p_2$	*equality*
		$\mid$	$p_1 + p_2, p_1 - p_2, \ldots$	*linear arithmetic*
		$\mid$	$f(p_1, \ldots, p_n)$	*uninterp. functions*
Verification Conditions	c	$::=$	p	*predicates*
		$\mid$	$c_1 \wedge c_2$	*conjunction*
		$\mid$	$\forall x : b.p \Rightarrow c$	*implication*

Verification Conditions can be predicates, as a base case, conjunction, so that many verification conditions can be gathered together, and implications, as generated by the `I-Ext` rule. Most of the syntax of predicates should be familiar to you.

Question: Do we need more boolean operators? Or maybe less?

Uninterpreted functions are essentially logical functions that always return the same value for the same input.

$$\forall x\ y. x = y \Rightarrow f(x) = f(y)$$

They are essential for program verification because 1. they can be used to encode program functions in the logic and 2. they can be used to capture ideas not directly implemented.

For example, in Liquid Haskell, we use the `measure` keyword to define uninterpreted functions.

```
{-@ measure isPrime :: Int -> Bool @-}
```

Given the property of being a function, Liquid Haskell, via SMT, can prove than on same input, `isPrime` returns the same output.

```
{-@ uninteprCheck :: x:Int -> y:Int
                  -> {v:() | x = y => isPrime x = isPrime y } @-}

uninteprCheck :: Int -> Int -> ()
uninteprCheck _ _ = ()
```

Primitive Operations

Up to now, we have seen how type checking (of base types) is reduced to checking of verification conditions. We also saw that constants, like integers and booleans, are given singleton types. But, what about other operations, like addition, subtraction, etc.? Remember that the type of `div` was refined to ensure that the second argument is non-zero. The same happens for other "primitive" operations, like addition, subtraction, etc. They all come with refined types that essentially map their operations to the SMT primitives.

So, in Liquid Haskell we have the following refined types for the basic operations:

```
(+)  :: Num a => x:a -> y:a -> {v:a | v = x + y}
(-)  :: Num a => x:a -> y:a -> {v:a | v = x - y}
(&&) :: x:Bool -> y:Bool -> {v:Bool | v = x && y }
(||) :: x:Bool -> y:Bool -> {v:Bool | v = x || y }
(==) :: Eq a => x:a -> y:a -> {v:Bool | v = (x = y)}
```

Such specifications of primitive operators come are *trusted* assumptions, required to connect the primitives of the programming language to the SMT solver. Thus, the refinement rule for such constants is:

$$\frac{}{\Gamma \vdash c : \text{tyConst}(c)}\ \texttt{T-Const}$$

Question: What is the verification condition of the problem below?

```
{-@ threePlusSix :: {v:Int | 0 <= v} @-}
threePlusSix :: Int
threePlusSix = 3 + 6
```

Question: What is the verification condition of the problem below?

```
{-@ plusSix :: x:Int -> {v:Int | x <= v} @-}
plusSix :: Int -> Int
plusSix x = x + 6
```

Question: Can you make the above code return only natural numbers?

Function Types

Refinements on function arguments define *preconditions*, i.e., assumptions about the arguments of the function, while refinements on the return type define *postconditions*, i.e., guarantees about the return value of the function.

For example, addition of two odd numbers is guaranteed to be even:

```
{-@ type Odd  = {v:Int | v mod 2 = 1} @-}
{-@ type Even = {v:Int | v mod 2 = 0} @-}

{-@ addOdds :: x:Odd -> y:Odd -> Even @-}
addOdds :: Int -> Int -> Int
addOdds x y = x + y
```

Increasing a positive number is guaranteed to be positive:

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

Question: What is the verification condition of the problem below?

So, the verification condition generated when checking the output, puts into the typing context the input and then checks the postcondition:

$$\frac{\text{T-Fun}}{\Gamma; x : \tau_x \vdash e : \tau} \; \Gamma \vdash \lambda x.e : x : \tau_x \to \tau$$

When type checking function applications, the subtyping rule is used to *weaken* the type of the argument into the correct type (e.g., make 2 a natural number).

$$\frac{\text{T-App}}{\Gamma \vdash e : (x : \tau_x \to \tau) \quad \Gamma \vdash y : \tau_x} \; \Gamma \vdash e\ y : x : \tau_x \to \tau[x/y]$$

Note: The type of the result can contain the input variable which is substitute. For example, what is a precise type for `incr2`?

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

```
incr2 :: Int -> Int
incr2 x = incr (incr x)
```

Note: Refinement types assume that application only happens with variables as arguments. This is not a limitation because internally we can always use a let binding to bind the argument to a variable. E.g., the above definition of `incr2` is equivalent to:

```
incr2 x = let y = incr x in incr y
```

This transformation is called ANF (administrative normal form). Can you think why it is required?

Subtyping exists also on function types and it gets interesting...

Question: Which of the following functions can be applied to `higher`?

```
fII, fIP, fIN, fPI, fPP, fPN, fNI, fNP, fNN :: Int -> Int
higher :: (Int -> Int) -> Int
fII = undefined
fIP = undefined
fIN = undefined
fPI = undefined
fPP = undefined
fPN = undefined
fNI = undefined
fNP = undefined
fNN = undefined
higher = undefined


{-@ fII :: Int -> Int @-}
{-@ fIP :: Int -> Pos @-}
{-@ fIN :: Int -> Nat @-}
{-@ fPI :: Pos -> Int @-}
{-@ fPP :: Pos -> Pos @-}
{-@ fPN :: Pos -> Nat @-}
{-@ fNI :: Nat -> Int @-}
{-@ fNP :: Nat -> Pos @-}
{-@ fNN :: Nat -> Nat @-}
```

```
{-@ higher :: (Nat -> Nat) -> Nat @-}

testhigher = higher fNN
```

As we should have figured out, the rule says that the result type should be a subtype but the argument a subpertype.

$$\frac{\Gamma \vdash \tau_{x2} \preceq \tau_{x1} \quad \Gamma; x_2 : \tau_{x2} \vdash \tau_1[x_1/x_2] \preceq \tau_2}{\Gamma \vdash x_1 : \tau_{x1} \to \tau_1 \preceq x_2 : \tau_{x2} \to \tau_2}\ \textsc{Sub-Fun}$$

We call the above rule on the argument *contravariant* and on the result *covariant*. Also, note that the result is checked under a context that contains the strongest argument!

Branching and Recursion

Let's compute the absolute value of a number.

```
abs :: Int -> Int
abs x = if x > 0 then x else -x
```

Question: What is the type of `abs`? **Question:** What is the verification condition generated?

Refinement types are *branch sensitive*, meaning that the type of the result of a branch depends on the condition of the branch.

The typing rule for branches takes this sensitivity into account:

$$\frac{\Gamma \vdash x : \{v : \texttt{bool} \mid p\} \quad \Gamma; y : \{y : \texttt{bool} \mid x\} \vdash e_1 : \tau \quad \Gamma; y : \{y : \texttt{bool} \mid \neg x\} \vdash e_2 : \tau}{\Gamma \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 : \tau}\ \textsc{T-If}$$

Note that the branch is also in ANF, so that it can get into the refinements. The typing uses a fresh variable y to capture the condition of the branch.

Of course, branching makes more sense when accompanied with recursive functions. Let's confirm that the sum of the first n natural numbers is greater then n:

```
sumN :: Int -> Int
sumN n = if n == 0 then 0 else n + sumN (n - 1)
```

Question: What is the type of `sumN`?

Question: What is the verification condition generated?

Type checking of recursive functions is itself a recursive process. Meaning, to check the type of `sumN`, we need to assume that `sumN` has the correct type!

$$\frac{\texttt{T-Rec}}{\Gamma; f : \tau_f \vdash e_f : \tau_f \quad \Gamma; f : \tau_f \vdash e : \tau} \; \Gamma \vdash \text{let rec } f = e_f \text{ in } e : \tau$$

Refined Polymorphism

The truth is polymorphism is a difficult topic in the area of programming languages. But, as a first step let's only see its great points and for refinement types, the great benefit of polymorphism is that any polymorphic function can be instantiated to refined values. For example, the identity function can be instantiated to propagate natural numbers:

```
myid :: a -> a
myid x = x

testPoly :: Int
{-@ testPoly :: Nat @-}
testPoly = higher myid
```

This is extremely powerful because it allows us to write generic code to propagate any application specific refinements! Next, we will see how this takes effect when using generic structures (e.g., arrays or in general data types). But now, for completeness, lets see the rules for polymorphism.

To get a polymorphic system, we need the ability to abstract and instantiate over type variables:

$$\frac{\texttt{T-TAbs}}{\Gamma; \alpha \vdash e : \tau} \; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau \qquad \frac{\texttt{T-TInst}}{\Gamma \vdash e : \forall\alpha.\tau} \; \Gamma \vdash e[\tau_\alpha] : \tau[\tau_\alpha/\alpha]$$

Since we allow subtyping, we also need to allow subtyping on polymorphic types. For simplicity, we assume that the type variable is renamed to be the same:

$$\frac{\texttt{Sub-Abs}}{\Gamma; \alpha \vdash \tau_1 \preceq \tau_2} \; \Gamma \vdash \forall\alpha.\tau_1 \preceq \forall\alpha.\tau_2$$

Putting it all together: Safe Indexing

The major application of refinement types is to ensure indexing is safe. So, let's generate structures of arrays and safely index them. Meaning `ArrayN a n` is an array of `n` elements of type `a` and accessing them with an index less than 0 or greater than `n` is an out of bounds error.

```
type Array a = Int -> a
{-@ type ArrayN a N = {i:Nat | i < N} -> a @-}

new :: Int -> a -> Array a
{-@ new :: n:Nat -> a -> ArrayN a n @-}
new n x = \i -> if 0 <= i && i < n then x else error "Out of Bounds"

set :: Int -> Int -> a -> Array a  -> Array a
{-@ set :: n:Nat -> i:{Nat | i < n} -> a -> ArrayN a n -> ArrayN a n @-}
set n i x a = \j -> if i == j then x else a j

get :: Int -> Int -> Array a -> a
{-@ get :: n:Nat -> i:{Nat | i < n} -> ArrayN a n -> a @-}
get n i a = a i
```

Let's create an array with 42 elements:

```
{-@ arr42 :: ArrayN Int 42 @-}
arr42 :: Array Int
arr42 = new 42 0

getElem :: Int
{-@ getElem :: Int @-}
getElem = get 42 10 arr42
```

Question: What are good indices of `arr42`?

To put now *all* the features we learnt together, let's assume a function that checks for primality and use it to generate the next prime number.

```
{-@ type Prime = {v:Int | isPrime v } @-}

isPrime :: Int -> Bool
{-@ isPrime :: i:Int -> {v:Bool | v <=> isPrime i } @-}
isPrime = undefined
```

```
nextPrime :: Int -> Int
{-@ nextPrime :: Nat -> Prime  @-}
nextPrime x = if isPrime x then x else nextPrime (x + 1)
```

Question: Given `nextPrime` can you generate an array that contains only prime numbers?

```
{-@ primes :: n:Nat -> ArrayN Prime n @-}
primes :: Int -> Array Int
primes = undefined
```

Summary

To sum up the most important features of a refinement type system are:

- *implicit subtyping:* the type of 2 turn into a non zero without any user casts!
- *branch sensitivity:* the type of the result of a branch depends on the condition of the branch!
- *polymorphism:* the type of a function can depend on the type of its arguments!

We saw the most important rule of a refinement type system! Next, we will look at the data types (so that we can implement more structured arrays) and we will go into more examples on how to use refinement types. But, for completeness, let's put here the definition of the language that we have seen and the typing and subtyping rules.

- Syntax of the language:

Basic Types	b	::=	`Int` | `Bool`,...	
Types	τ	::=	$\{v : \mathtt{b} \mid p\}$	*base*
		|	$x : \tau_x \to \tau$	*function*
		|	$\forall \alpha.\tau$	*polymorphic*
Expressions	e	::=	x	*variables*
		|	c	*constants*
		|	$\lambda x.e$	*function*
		|	$e\ x$	*application*
		|	if x then e else e	*if*
		|	let $x = e$ in e	*let*
		|	let rec $f = e$ in e	*recursion*
		|	$\Lambda \alpha.e$	*type abs.*
		|	$e[\tau]$	*type appl.*

- Typing rules collected:

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 \preceq \tau_2}{\Gamma \vdash e : \tau_2}\ \texttt{T-Sub}$$

$$\frac{}{\Gamma \vdash c : \text{tyConst}(c)}\ \texttt{T-Const} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ \texttt{T-Var}$$

$$\frac{\Gamma; x : \tau_x \vdash e : \tau}{\Gamma \vdash \lambda x.e : x : \tau_x \to \tau}\ \text{T-Fun} \qquad \frac{\Gamma \vdash e : (x : \tau_x \to \tau) \quad \Gamma \vdash y : \tau_x}{\Gamma \vdash e\ y : x : \tau_x \to \tau[x/y]}\ \text{T-App}$$

$$\frac{\Gamma \vdash x : \{v : \mathtt{bool} \mid p\} \quad \Gamma; y : \{y : \mathtt{bool} \mid x\} \vdash e_1 : \tau \quad \Gamma; y : \{y : \mathtt{bool} \mid \neg x\} \vdash e_2 : \tau}{\Gamma \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 : \tau}\ \texttt{T-If}$$

$$\frac{\Gamma; \vdash e_x : \tau_x \quad \Gamma; x : \tau_x \vdash e : \tau}{\Gamma \vdash \text{let } x = e_x \text{ in } e : \tau}\ \texttt{T-Let}$$

$$\frac{\Gamma; f : \tau_f \vdash e_f : \tau_f \quad \Gamma; f : \tau_f \vdash e : \tau}{\Gamma \vdash \text{let rec } f = e_f \text{ in } e : \tau}\ \texttt{T-Rec}$$

$$\frac{\Gamma; \alpha \vdash e : \tau}{\Gamma \vdash \Lambda \alpha.e : \forall \alpha.\tau}\ \texttt{T-TAbs} \qquad \frac{\Gamma \vdash e : \forall \alpha.\tau}{\Gamma \vdash e[\tau_\alpha] : \tau[\tau_\alpha/\alpha]}\ \texttt{T-TInst}$$

- Subtyping rules collected:

$$\frac{\Gamma \vdash \forall v : b.p_1 \Rightarrow p_2}{\Gamma \vdash \{v : \mathtt{b} \mid p_1\} \preceq \{v : \mathtt{b} \mid p_2\}}\ \texttt{Sub-Base}$$

$$\frac{\Gamma \vdash \tau_{x2} \preceq \tau_{x1} \quad \Gamma; x_2 : \tau_{x2} \vdash \tau_1[x_1/x_2] \preceq \tau_2}{\Gamma \vdash x_1 : \tau_{x1} \to \tau_1 \preceq x_2 : \tau_{x2} \to \tau_2}\ \texttt{Sub-Fun}$$

$$\frac{\text{Sub-Abs}}{\Gamma; \alpha \vdash \tau_1 \preceq \tau_2} \; \Gamma \vdash \forall \alpha.\tau_1 \preceq \forall \alpha.\tau_2$$

Note: The definitions of the rules are syntactic. One subtyping rule exists for each type and one typing rule exists for each expression. The subtyping rule applies to all expressions and *in general* make the system not algorithmic (meaning, when does this rule apply?) To solve this problem and make the system algorithmic Liquid Haskell uses a *bidirectional* type checking algorithm. In general, the above rules are simplified in various ways (e.g., well formedness is not discussed).

Further Reading

These lectures notes are based on the Liquid Haskell Tutorial. For further reading on how to develop a refinement type checker for your own language, you can read the Refinement Types: A Tutorial and for the theoretical foundations of LiquidHaskell, the publication Mechanizing Refinement Types.

Cheatsheet

Here is the definition of the primes array.

```
primes :: Int -> Array Int
primes n = go  1 0  (new n (nextPrime 1))
  where
    go i j a
      | i < n     = go (i + 1) (j + 1) (set n j (nextPrime j) a)
      | otherwise = a
```

2

Data Types

```
{-# OPTIONS_GHC -fplugin=LiquidHaskell #-}
module Lecture_02_DataTypes where

main :: IO ()
main = return ()
```

3

Case Study: Insertion Sort

```
{-# OPTIONS_GHC -fplugin=LiquidHaskell #-}
module Lecture_03_CaseStudy_InsertionSort where

main :: IO ()
main = return ()
```

4
Abstract Refinement Types

```
{-# OPTIONS_GHC -fplugin=LiquidHaskell #-}
module Lecture_04_AbstractRefinementTypes where

main :: IO ()
main = return ()
```

5
Theorem Proving

```haskell
{-# OPTIONS_GHC -fplugin=LiquidHaskell #-}
module Lecture_05_TheoremProving where

main :: IO ()
main = return ()
```

6

Data Propositions

```haskell
{-# OPTIONS_GHC -fplugin=LiquidHaskell #-}
module Lecture_06_DataPropositions where

main :: IO ()
main = return ()
```