LiquidHaskell

Theorem Proving for All

Niki Vazou

institute
iMdea
software

Haskell

+

Refinement Types

=

LiquidHaskell

Haskell

```
take :: [a] -> Int -> [a]
```

```
> take [1,2,3] 2
> [1,2]
```

Haskell

```haskell
take :: [a] -> Int -> [a]
```

```
> take [1,2,3] 500
> ???
```

Refinement Types

```
take :: xs:[a] -> {i:Int | i < len xs} -> [a]
```

LiquidHaskell

```
take :: xs:[a] -> {i:Int | i < len xs} -> [a]
```

```
> take [1,2,3] 500
> Refinement Type Error!
```

LiquidHaskell

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

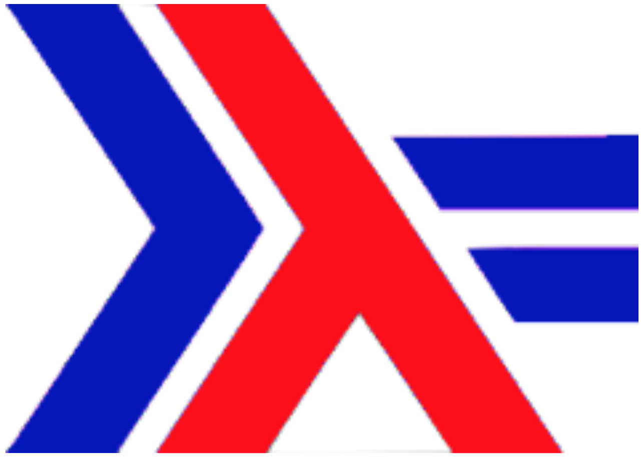
III. Expressiveness: Theorem Proving

I. Static Checks: Fast & Safe Code

The Heartbleed Bug

Buffer overread in OpenSSL. 2015

in



```
module Data.Text where
take :: t:Text -> i:Int -> Text
```

```
> take "hat" 500
> *** Exception: Out Of Bounds!
```

Runtime Checks

```
take :: t:Text->i:Int->Text
take t i | i < len t
  = Unsafe.take t i
take t i
  = error “Out Of Bounds!”
```

Safe, but slow!

No Checks

```
take :: t:Text -> i:Int -> Text
take t i | ~~i < len t~~
  = Unsafe.take t i
~~take t i~~
~~  = error “Out Of Bounds!”~~
```

Fast, but unsafe!

No Checks

```
take :: t:Text->i:Int->Text
take t i | ~~i < len t~~
  = Unsafe.take t i
~~take t i~~
~~  = error “Out Of Bounds!”~~
```

Overread

```
> take "hat" 500
> “hat\58456\2594\SOH\NUL…
```

Static Checks

```
take :: t:Text->i:Int->Text
take t i | i < len t
  = Unsafe.take t i
take t i
  = error “Out Of Bounds!”
```

Static Checks

```
take :: t:Text->i:{i < len t}->Text
take t i | i < len t
 = Unsafe.take t i
take t i
 = error “Out Of Bounds!”
```

Static Checks

```
take :: t:Text->i:{i < len t}->Text
take t i | ~~i < len t~~
  = Unsafe.take t i
~~take t i~~
~~  = error “Out Of Bounds!”~~
```

Static Checks

```
take :: t:Text->i:{i < len t}->Text
take t i
 = Unsafe.take t i
```

Static Checks

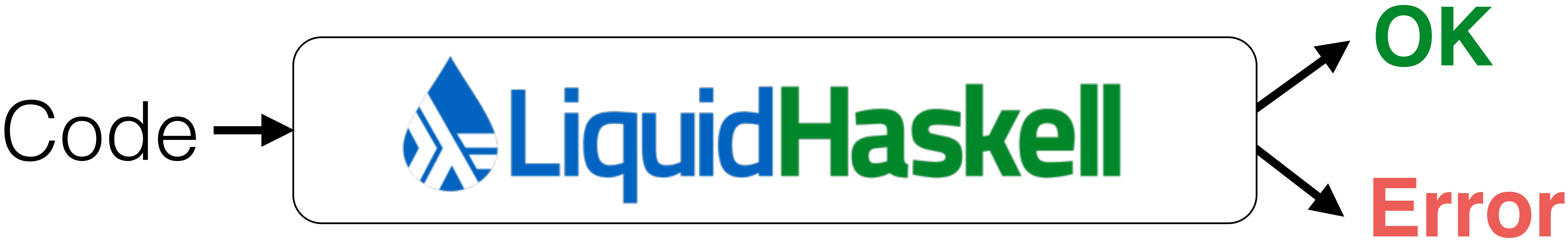
```
take :: t:Text->i:{i < len t}->Text
take t i
 = Unsafe.take t i
```

```
> take "hat" 500
```

Type Error

LiquidHaskell

Refinement Types



Code

LiquidHaskell

OK

Error

Checks valid arguments, under facts.

Checks valid arguments, under facts.

```
take :: t:Text->{v|v < len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

```
take :: t:Text -> {v | v < len t} -> Text

heartbleed = let x = "hat"
             in  take x 500
```

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v < len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v < len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

```
take :: t:Text -> {v | v < len t} -> Text

heartbleed = let x = "hat"
             in  take x 500
```

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v < len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v < len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

SMT-query

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v < len t}->Text
heartbleed = let x = "hat"
             in  take x 500
```

SMT-invalid

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v < len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

Checker reports **Error**

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

```
take :: t:Text->{v|v < len t}->Text

heartbleed = let x = "hat"
             in  take x 500
```

Checker reports **Error**

```
len x = 3 => v = 500 => v < len x
```

Checks valid arguments, under facts.

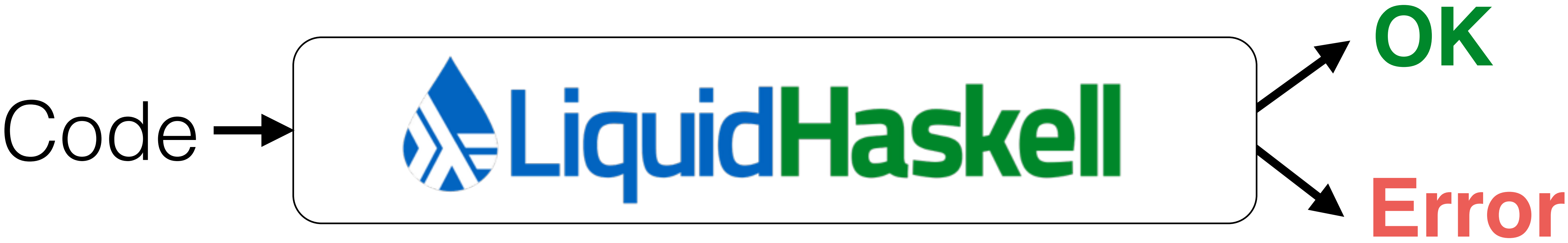
```
take :: t:Text->{v|v < len t}->Text

heartbleed = let x = "hat"
             in  take x  2
```

Checker reports **OK**

SMT-valid

```
len x = 3 => v = 2 => v < len x
```



Code

LiquidHaskell

OK

Error

Checks valid arguments, under facts.

Static Checks

LiquidHaskell

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

LiquidHaskell

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

II. Application: Speed up Parsing

DEMO

Application: Speed up Parsing

Provably Correct & Faster Code!

SMT-Automatic Verification

SMT-Automatic Verification

How expressive can we get?

LiquidHaskell

I.Static Checks : Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

III. Expressiveness: Theorem Proving

Theorem: For any x, `reverse [x] = [x]`

Proof.

```
       reverse [x]
-   applying reverse on [x]
=      reverse [] ++ [x]
-   applying reverse on []
=      [] ++ [x]
-   applying ++ on [] and [x]
=      [x]
       QED
```

Proof is in pen-and-paper :(

Theorem: For any x, `reverse [x] = [x]`

Proof.

```
         reverse [x]
   –   applying reverse on [x]
   =     reverse [] ++ [x]
   –   applying reverse on []
   =     [] ++ [x]
   –   applying ++ on [] and [x]
   =     [x]
         QED
```

Proof is not machine checked.

Theorem: For any x, `reverse [x] = [x]`

Proof.

```
      reverse [x]

–   obviously!

=     [x]
      QED
```

Proof is not machine checked.

Theorem: For any x, `reverse [x] = [x]`

Proof.

```
     reverse [x]
-   applying reverse on [x]
=    reverse [] ++ [x]
-   applying reverse on []
=    [] ++ [x]
-   applying ++ on [] and [x]
=    [x]
     QED
```

Proof is not machine checked.

Check it with Liquid Haskell!

Theorems as Refinement Types

Theorem:

For any x, reverse [x] = [x]

Refinement Type:

```
x:a → { v:() | reverse [x] = [x] }
```

SMT equality

Theorems as Refinement Types

Theorem:

For any x, reverse [x] = [x]

Refinement Type:

```
x:a → { reverse [x] = [x] }
```

```
x:a → { reverse [x] = [x] }
```

Proof.

```
      reverse [x]
 –  applying reverse on [x]
 =    reverse [] ++ [x]
 –  applying reverse on []
 =    [] ++ [x]
 –  applying ++ on [] and [x]
 =    [x]
      QED
```

How to connect theorem with proof?

Theorems are types
Proofs are programs

— Curry & Howard

```
singletonP :: x:a → { reverse [x] = [x] }
singletonP x
    =   reverse [x]
    -  applying reverse on [x]
    =   reverse [] ++ [x]
    -  applying reverse on []
    =   [] ++ [x]
    -  applying ++ on [] and [x]
    =   [x]
        QED
```

Proof as a Haskell function

```
singletonP :: x:a → { reverse [x] = [x] }
singletonP x
    =   reverse [x]
    –  applying reverse on [x]
    =   reverse [] ++ [x]
    –  applying reverse on []
    =   [] ++ [x]
    –  applying ++ on [] and [x]
    =   [x]
        QED
```

Proof as a Haskell function

```
singletonP :: x:a → { reverse [x] = [x] }

singletonP x
    =    reverse [x]
    -   applying reverse on [x]
    =    reverse [] ++ [x]
    -   applying reverse on []
    =     [] ++ [x]
    -   applying ++ on [] and [x]
    =     [x]
         QED
```

How to encode equality?

Equational Operator in (Liquid) Haskell

checks both arguments are equal

```
(==.) :: x:a -> y:{ a | x = y }
      -> {v:a | v = x && v = y }
x ==. y = y
```

returns 2nd argument,

to continue the proof!

```
singletonP :: x:a → { reverse [x] = [x] }
singletonP x
    =   reverse [x]
    –  applying reverse on [x]
    ==. reverse [] ++ [x]
    –  applying reverse on []
    =   [] ++ [x]
    –  applying ++ on [] and [x]
    =   [x]
        QED
```

```
singletonP :: x:a → { reverse [x] = [x] }
singletonP x
    =   reverse [x]
    –  applying reverse on [x]
    ==. reverse [] ++ [x]
    –  applying reverse on []
    ==. [] ++ [x]
    –  applying ++ on [] and [x]
    ==. [x]
        QED
```

```
singletonP :: x:a → { reverse [x] = [x] }

singletonP x
    =    reverse [x]
    –   applying reverse on [x]
    ==.  reverse [] ++ [x]
    –   applying reverse on []
    ==.  [] ++ [x]
    –   applying ++ on [] and [x]
    ==.  [x]
         QED
```

How to encode QED?

Define QED as data constuctor...

```
data QED = QED
```

... that casts anything into a proof
(i.e., a unit value).

```
(***) :: a -> QED -> ()
_ *** QED = ()
```

```
singletonP :: x:a → { reverse [x] = [x] }
singletonP x
    =   reverse [x]
    –   applying reverse on [x]
    ==. reverse [] ++ [x]
    –   applying reverse on []
    ==. [] ++ [x]
    –   applying ++ on [] and [x]
    ==. [x]
    *** QED
```

Theorem Proving in Haskell

Theorems are Types

```
singletonP :: x:a → { reverse [x] = [x] }
```

Theorem Application is Function Call

```
singletonP 1 :: { reverse [1] = [1] }
```

Theorem Application is Function Call

```
singletonP1 :: { reverse [1] = [1] }
singletonP1
  =   reverse [1]
    ? singletonP 1
  ==. [1]
  *** QED
```

```
(?) :: a -> () -> a
x ? _ = x
```

Theorem Proving for All

Reasoning about Haskell Programs in Haskell!

Equational operators (`==.`, `?`, `QED`, `***`)
let us encode proofs as Haskell functions
checked by Liquid Haskell.

Theorem Proving for All

Reasoning about Haskell Programs in Haskell!

How to encode inductive proofs?

Theorem: For any list x, reverse (reverse x) = x.

Proof.

Base Case:

```
      reverse (reverse [])
      – applying inner reverse
=     reverse []
      – applying reverse
=     []
      QED
```

Inductive Case:

```
      reverse (reverse (x:xs))
    – applying inner reverse
=     reverse (reverse xs ++ [x])
    – distributivity on (reverse xs) [x]
=     reverse [x] ++ reverse (reverse xs)
    – involution on xs
=     reverse [x] ++ xs
    – singleton on x
=     [x] ++ xs
    – applying ++
=     x:([] ++ xs)
    – applying ++
=     (x:xs)
      QED
```

Theorem: For any list x, reverse (reverse x) = x.

Proof.

Base Case:

```
    reverse (reverse [])
    – applying inner reverse
=   reverse []
    – applying reverse
=   []
    QED
```

Inductive Case:

```
    reverse (reverse (x:xs))
   – applying inner reverse
=   reverse (reverse xs ++ [x])
   – distributivity on (reverse xs) [x]
=   reverse [x] ++ reverse (reverse xs)
   – involution on xs
=   reverse [x] ++ xs
   – singleton on x
=   [x] ++ xs
   – applying ++
=   x:([] ++ xs)
   – applying ++
=   (x:xs)
    QED
```

Step 1: Define a recursive function!

Theorem: For any list x, reverse (reverse x) = x.

Proof.

```
involutionP []
=    reverse (reverse [])
     – applying inner reverse
=    reverse []
     – applying reverse
=    []
     QED
```

```
involutionP (x:xs)
=    reverse (reverse (x:xs))
   – applying inner reverse
=    reverse (reverse xs ++ [x])
   – distributivity on (reverse xs) [x]
=    reverse [x] ++ reverse (reverse xs)
   – involution on xs
=    reverse [x] ++ xs
   – singleton on x
=    [x] ++ xs
   – applying ++
=    x:([] ++ xs)
   – applying ++
=    (x:xs)
     QED
```

Step 1: Define recursive function

Step 2: Use equational proofs!

Theorem: For any list x, reverse (reverse x) = x.

Proof.

```
involutionP []
==. reverse (reverse [])
    - applying inner reverse
==. reverse []
    - applying reverse
==. []
*** QED
```

```
involutionP (x:xs)
==. reverse (reverse (x:xs))
   - applying inner reverse
==. reverse (reverse xs ++ [x])
   - distributivity on (reverse xs) [x]
==. reverse [x] ++ reverse (reverse xs)
   - involution on xs
==. reverse [x] ++ xs
   - singleton on x
==. [x] ++ xs
   - applying ++
==. x:([] ++ xs)
   - applying ++
==. (x:xs)
*** QED
```

Step 2: Use equational operators!

Step 3: Lemmata are function calls!

Theorem: For any list x, reverse (reverse x) = x.

Proof.

```
involutionP []
==. reverse (reverse [])
    – applying inner reverse
==. reverse []
    – applying reverse
==. []
*** QED
```

```
involutionP (x:xs)
==. reverse (reverse (x:xs))
   – applying inner reverse
==. reverse (reverse xs ++ [x])
   ? distributivityP (reverse xs) [x]
==. reverse [x] ++ reverse (reverse xs)
   ? involutionP xs
==. reverse [x] ++ xs
   ? singletonP x
==. [x] ++ xs
   – applying ++
==. x:([] ++ xs)
   – applying ++
==. (x:xs)
*** QED
```

Step 3: Lemmata are function calls!

Theorem: For any list x, reverse (reverse x) = x.

Proof.

```
involutionP []
==. reverse (reverse [])
    - applying inner reverse
==. reverse []
    - applying reverse
==. []
*** QED
```

```
involutionP (x:xs)
==. reverse (reverse (x:xs))
   - applying inner reverse
==. reverse (reverse xs ++ [x])
   ? distributivityP (reverse xs) [x]
==. reverse [x] ++ reverse (reverse xs)
   ? involutionP xs
==. reverse [x] ++ xs
   ? singletonP x
==. [x] ++ xs
   - applying ++
==. x:([] ++ xs)
   - applying ++
==. (x:xs)
*** QED
```

Note: Inductive hypothesis is recursive call!

Theorem: For any list x, reverse (reverse x) = x.

Proof.

```
involutionP []
==. reverse (reverse [])
    - applying inner reverse
==. reverse []
    - applying reverse
==. []
*** QED
```

```
involutionP (x:xs)
==. reverse (reverse (x:xs))
   - applying inner reverse
==. reverse (reverse xs ++ [x])
   ? distributivityP (reverse xs) [x]
==. reverse [x] ++ reverse (reverse xs)
   ? involutionP xs
==. reverse [x] ++ xs
   ? singletonP x
==. [x] ++ xs
   - applying ++
==. x:([] ++ xs)
   - applying ++
==. (x:xs)
*** QED
```

Question: Is the proof well founded?

LiquidHaskell

Used to encode pen-and-pencil proofs and function optimizations.

“Theorem Proving for All”, Haskell’18

https://bit.ly/2yjvJo3

LiquidHaskell

Used to encode pen-and-pencil proofs or even sophisticated security proofs.

“LWeb: Information Flow Security for Multi-Tier Web Applications”, POPL’19

https://bit.ly/2EcyDAh

LiquidHaskell

Used to encode pen-and-pencil proofs
or encode resource analysis.

“Liquidate your assets”

https://bit.ly/2Ht3uIG

To be presented at IMDEA:
by Martin Handley
Tue March 19 @10.45

LiquidHaskell

Used to encode pen-and-pencil proofs

But, proof interaction is missing.

LiquidHaskell

Theorem Proving for All

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

@nikivazou

Thanks!