Liquid Haskell
Haskell as a Theorem Prover

Niki Vazou
University of Maryland
Software bugs are everywhere

Airbus A400M crashed due to a software bug.
— May 2015
Software bugs are everywhere

The Heartbleed Bug.
Buffer overflow in OpenSSL. 2015
HOW THE HEARTBLEED BUG WORKS:

Server, are you still there? If so, reply "POTATO" (6 letters).

User Meg wants these 6 letters: POTATO.
User Ida wants pages about "irl games". Unlocking secure records with master key 5130985733435.
Nerio (chrome user) sends this message: "POTATO".

POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

User Olivia from London wants pages about "how bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834be962e2eb0ff9b13b669).

HMM... BIRD

User Olivia from London wants pages about "how bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834be962e2eb0ff9b13b669).

SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server’s master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "ColHeRaSt".
Make bugs difficult to express

Using Modern Programming Languages

F#, Ocaml, Erlang, Scala, Haskell

Because of

Strong Types + λ-Calculus
Make bugs difficult to express

Using Modern Programming Languages

F#, OCaml, Erlang, Scala, Haskell

Because of Strong Types + λ-Calculus

Well Typed Programs cannot go wrong!
λ> :m +Data.Text Data.Text.TextUnsafe
λ> let pack = "hat"

λ> :t takeWord16
takeWord16 :: Text -> Int -> Text

VS.
λ> :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"
λ> takeWord16 pack True
Type Error: Cannot match `Bool` vs `Int`
λ> :m +Data.Text Data.Text.Text.Unsafe
λ> let pack = "hat"

λ> takeWord16 pack 500
"hat\58456\2594\SOH\NUL..."
Valid Values for `takeWord16`?

takeWord16 :: t:Text -> i:Int -> Text

All Ints

..., -2, -1, 0, 1, 2, 3, ...
Valid Values for `takeWord16`?

takeWord16 :: t:Text -> i:Int -> Text

Valid Ints
0, 1 ..., len t

Invalid Ints
len t + 1, ...
Refinement Types

take :: \( t : \text{Text} \rightarrow \{ v : \text{Int} \mid v \leq \text{len } t \} \rightarrow \text{Text} \)
Refinement Types

take :: t:Text -> \{v:Int | v <= len t\} -> Text

\[
\lambda > :m \ +Data.Text\ Data.Text.Text.Unsafe
\]
\[
\lambda > let\ pack = "hat"
\]
\[
\lambda > take\ pack\ 500
\]
Refinement Type Error
Refinement Types

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

\[
\begin{align*}
\lambda : & \text{m (+Data.Text Data.Text.Text.Unsafe)} \\
\lambda & \text{let pack = "hat"}
\end{align*}
\]

\[
\begin{align*}
\lambda & \text{take pack 500} \\
\text{Refinement Type Error}
\end{align*}
\]

Liquid Haskell
Refinement Types

Checks valid arguments, under facts.
Checks valid arguments, under facts.

take :: t:Text -> \{v | v <= \text{len } t\} -> \text{Text}

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

len x=3 => v=500 => v<=\text{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 <= \text{len } t\} -> Text

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

len x = 3 => v = 500 => v <= \text{len } x
Checks valid arguments, under facts.

```haskell
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-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
Liquid Haskell:
Checks valid arguments, under facts.
What are interesting facts?
What are interesting facts?

len "hat" = 3
What are interesting facts?

len "hat" = 3

(xs ++ ys) ++ zs == xs ++ (ys ++ zs)
What are interesting facts?

- `len "hat" = 3`
- `(xs ++ ys) ++ zs == xs ++ (ys ++ zs)`
- `f is == mapReduce f is`
What are interesting facts?

len "hat" = 3

(xs ++ ys) ++ zs == xs ++ (ys ++ zs)

f is == mapReduce f is

Theorems about Haskell functions
Theorems about Haskell functions

(Liquid) Haskell as a theorem prover.
(Liquid) Haskell as a theorem prover.

Specify theorems as Refinement Types

Prove theorems in Haskell

Use Liquid Haskell to check correctness
(Liquid) Haskell as a theorem prover.

data \text{L} \ a = \text{N} \mid \text{C} \ a \ (\text{L} \ a)
(Liquid) Haskell as a theorem prover.

```hs
data L a = N | C a (L a)

N ++ ys = ys
(C x xs) ++ ys = C x (xs ++ ys)
```
(Liquid) Haskell as a theorem prover.

\[ N \ ++ \ ys = ys \]
\[ (\mathcal{C} \ x \ xs) \ ++ \ ys = \mathcal{C} \ x \ (xs \ ++ \ ys) \]

assoc :: \( \mathbb{L} \ a \rightarrow \mathbb{L} \ a \rightarrow \mathbb{L} \ a \rightarrow () \)
(Liquid) Haskell as a theorem prover.

\[ \text{assoc} :: \text{xs} : \text{L} a \rightarrow \text{ys} : \text{L} a \rightarrow \text{zs} : \text{L} a \rightarrow \{ v : () \mid (\text{xs} ++ \text{ys}) ++ \text{zs} = \text{xs} ++ (\text{ys} ++ \text{zs}) \} \]
(Liquid) Haskell as a theorem prover.

\[
\begin{align*}
N \ ++ \ ys &= ys \\
(C \ x \ xs) \ ++ \ ys &= C \ x \ (xs \ ++ \ ys)
\end{align*}
\]

\[\text{assoc} :: \ xs: L \ a \rightarrow \ ys: L \ a \rightarrow \ zs: L \ a \rightarrow \ \\
\{ \ (xs \ ++ \ ys) \ ++ \ zs \ == \ xs \ ++ \ (ys \ ++ \ zs) \ \} \]
(Liquid) Haskell as a theorem prover.

What is the body of assoc?

```haskell
assoc :: xs:L a -> ys:L a -> zs:L a ->
  { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs = ???
```
What is the body of `assoc`?

A unit Haskell value showing that left-hand side == right-hand side

```
assoc :: xs:L a -> ys:L a -> zs:L a ->
{ (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs = ???
```
(Liquid) Haskell as a theorem prover.

What is the body of `assoc`?

A unit Haskell value showing that left-hand side == right-hand side

```haskell
assoc :: xs:L a -> ys:L a -> zs:L a ->
  { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }
assoc xs ys zs = ()
```

Type Error!
(Liquid) Haskell as a theorem prover.

A unit Haskell value showing that left-hand side == right-hand side

assoc :: xs:L a -> ys:L a -> zs:L a -> 
\{ (xs ++ ys) ++ zs == xs ++ (ys ++ zs) \} 
assoc xs ys zs
== (xs ++ ys) ++ zs
...
==. xs ++ (ys ++ zs)
*** QED
(Liquid) Haskell as a theorem prover.

A unit Haskell value showing that the left-hand side == right-hand side

\[
\text{assoc} :: \text{xs}:\text{L} \ a \rightarrow \text{ys}:\text{L} \ a \rightarrow \text{zs}:\text{L} \ a \rightarrow \\
\{ (\text{xs} \ ++ \ \text{ys}) \ ++ \ \text{zs} = \text{xs} \ ++ \ (\text{ys} \ ++ \ \text{zs}) \} \\
\text{assoc} \ \text{xs} \ \text{ys} \ \text{zs} \\
= \quad (\text{xs} \ ++ \ \text{ys}) \ ++ \ \text{zs} \\
\text{...} \\
==. \ \text{xs} \ ++ \ (\text{ys} \ ++ \ \text{zs}) \\
*** \ \text{QED}
\]
A unit Haskell value showing that left-hand side == right-hand side

\[
\text{assoc :: } \text{xs} : \text{L a} \rightarrow \text{ys} : \text{L a} \rightarrow \text{zs} : \text{L a} \rightarrow \\
\{ (\text{xs} ++ \text{ys}) ++ \text{zs} == \text{xs} ++ (\text{ys} ++ \text{zs}) \}
\]

\[
\text{assoc xs ys zs} \\
= (\text{xs} ++ \text{ys}) ++ \text{zs} \\
\ldots \\
==. \text{xs} ++ (\text{ys} ++ \text{zs}) \\
*** \text{QED}
\]
(Liquid) Haskell as a theorem prover.

A unit Haskell value showing that left-hand side == right-hand side

assoc :: xs: :L a -> ys: :L a -> zs: :L a ->
{ (xs ++ ys) ++ zs == xs ++ (ys ++ zs) }  
assoc xs ys zs
= (xs ++ ys) ++ zs
...  
==. xs ++ (ys ++ zs)  
*** QED
(Liquid) Haskell as a theorem prover.

\[ N \quad ++ \quad ys \quad = \quad ys \]
\[ (C \quad x \quad xs) \quad ++ \quad ys \quad = \quad C \quad x \quad (xs \quad ++ \quad ys) \]

assoc :: xs:L a -> ys:L a -> zs:L a -> 
\{ \ (xs \quad ++ \quad ys) \quad ++ \quad zs \quad == \quad xs \quad ++ \quad (ys \quad ++ \quad zs) \ \} 
assoc xs ys zs
= (xs ++ ys) ++ zs
==. xs ++ (ys ++ zs)
*** QED
(Liquid) Haskell as a theorem prover.

\[
\begin{align*}
N \mathbin{++} ys & = ys \\
(C \times xs) \mathbin{++} ys & = C \times (xs \mathbin{++} ys)
\end{align*}
\]

**assoc**: \(xs: L\ a \rightarrow ys: L\ a \rightarrow zs: L\ a \rightarrow \{ (xs \mathbin{++} ys) \mathbin{++} zs = xs \mathbin{++} (ys \mathbin{++} zs) \} \)

assoc \(N\) \(ys\) \(zs\)

\[=
\begin{align*}
(N \mathbin{++} ys) \mathbin{++} zs
\end{align*}
\]

\[==.
\begin{align*}
N \mathbin{++} (ys \mathbin{++} zs)
\end{align*}
\]

*** QED
(Liquid) Haskell as a theorem prover.

\[ N ++ ys = ys \]
\[ (C \times xs) ++ ys = C \times(xs ++ ys) \]

assoc :: \(xs : L a \rightarrow ys : L a \rightarrow zs : L a \rightarrow\)
\[\{ (xs ++ ys) ++ zs == xs ++ (ys ++ zs) \}\]
assoc \(N\) \(ys\) \(zs\)
\[= (N ++ ys) ++ zs \]
\[==. N ++ (ys ++ zs)\]

*** QED
(Liquid) Haskell as a theorem prover.

\[
N \mathbin{\text{++}} ys = ys \\
(C \times xs) \mathbin{\text{++}} ys = C \times (xs \mathbin{\text{++}} ys)
\]

\[
\text{assoc} :: \text{xs} :: \text{L a} \to \text{ys} :: \text{L a} \to \text{zs} :: \text{L a} \to \\
\{ (\text{xs} \mathbin{\text{++}} \text{ys}) \mathbin{\text{++}} \text{zs} == \text{xs} \mathbin{\text{++}} (\text{ys} \mathbin{\text{++}} \text{zs}) \}
\]

\[
\text{assoc } N \text{ ys zs} \\
= (N \mathbin{\text{++}} ys) \mathbin{\text{++}} zs \\
==. ys \mathbin{\text{++}} zs \\
==. N \mathbin{\text{++}} (ys \mathbin{\text{++}} zs) \\
*** \text{QED}
\]
(Liquid) Haskell as a theorem prover.

\[
\begin{align*}
N \ ++ \ ys & = ys \\
(C \times xs) \ ++ \ ys & = C \times (xs \ ++ \ ys)
\end{align*}
\]

\[
\text{assoc} :: \ L a \rightarrow y \rightarrow z \rightarrow \\
\{ (x \ ++ \ y) \ ++ \ z \equiv x \ ++ \ (y \ ++ \ z) \}
\]

assoc N ys zs

=  \ (N \ ++ \ ys) \ ++ \ zs

==.  y \ ++ \ z

==.  N \ ++ \ (y \ ++ \ z)

*** QED
(Liquid) Haskell as a theorem prover.

\[ N \,++\, ys = ys \]
\[ (C \times xs) \,++\, ys = C \times (xs \,++\, ys) \]

assoc :: \( \text{xs}\,:\text{L}\, a \rightarrow \text{ys}\,:\text{L}\, a \rightarrow \text{zs}\,:\text{L}\, a \rightarrow \)
\{ (xs ++ ys) ++ zs == xs ++ (ys ++ zs) \}
assoc N ys zs
= (N ++ ys) ++ zs
==. ys ++ zs
==. N ++ (ys ++ zs)
*** QED
(Liquid) Haskell as a theorem prover.

\[ N \mathbin{\mathbf{++}} ys = ys \]
\[ (\mathcal{C} \times xs) \mathbin{\mathbf{++}} ys = \mathcal{C} \times (xs \mathbin{\mathbf{++}} ys) \]

\[
\text{assoc} :: \mathbf{xs} : \mathbf{L} a \rightarrow \mathbf{ys} : \mathbf{L} a \rightarrow \mathbf{zs} : \mathbf{L} a \rightarrow \\
\{ (\mathbf{xs} \mathbin{\mathbf{++}} \mathbf{ys}) \mathbin{\mathbf{++}} \mathbf{zs} = \mathbf{xs} \mathbin{\mathbf{++}} (\mathbf{ys} \mathbin{\mathbf{++}} \mathbf{zs}) \}
\]

\[
\text{assoc} N \mathbf{ys} \mathbf{zs} = (N \mathbin{\mathbf{++}} \mathbf{ys}) \mathbin{\mathbf{++}} \mathbf{zs} \\
\text{==. } \mathbf{ys} \mathbin{\mathbf{++}} \mathbf{zs} \\
\text{==. } N \mathbin{\mathbf{++}} (\mathbf{ys} \mathbin{\mathbf{++}} \mathbf{zs}) \\
\text{*** QED}
\]
(Liquid) Haskell as a theorem prover.

Demo
Specify theorems as Refinement Types
Prove theorems in Haskell
Use Liquid Haskell to check correctness

(Liquid) Haskell as a theorem prover.

Thanks!
END