



Liquid Haskell

Usable Language-Based Verification

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 overread in OpenSSL. 2015

Verification Prevents Bugs!

Verification Prevents Bugs!



Project Everest.
Builds a Verified HTTPS Stack.



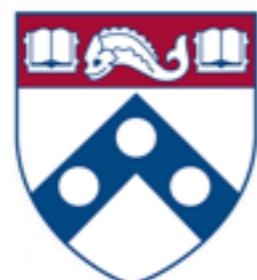
Verification Prevents Bugs!



Prove end-to-end Correctness of Whole Systems



Princeton



Penn



Yale



MIT

But Requires Experts

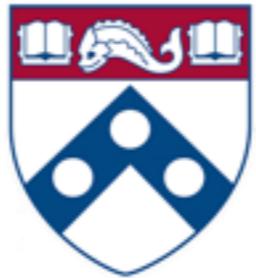
Inria



Carnegie
Mellon
University



Princeton



Penn



Yale



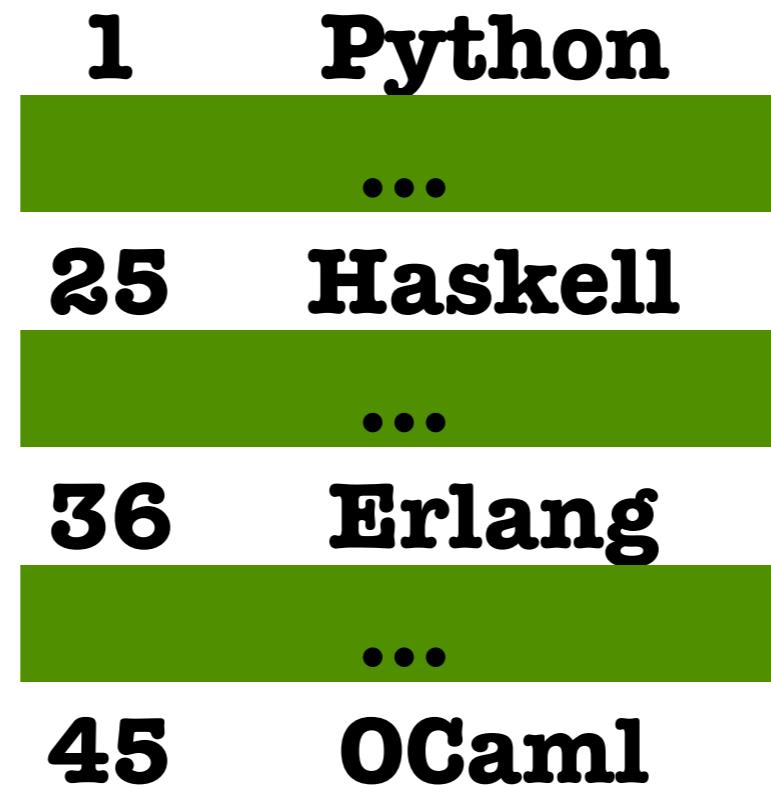
MIT

My Goal:
Make Verification Accessible

My Approach:
Turn a Language into a Verifier

Haskell

General Purpose Programming Language



Popularity of Programming Languages
by spectrum.ieee

Haskell

+

Refinement Types

=



Liquid Haskell

Haskell

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

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

Haskell

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

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

Refinement Types

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



Liquid Haskell

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

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



Liquid Haskell

Use Haskell's syntax, runtime, and **users**!

Used in **Industry**: to speed up code



AWAKE



Well-Typed

Used in **Education**: in Haskell courses!





Liquid Haskell

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

IV. Status: Industry & Education

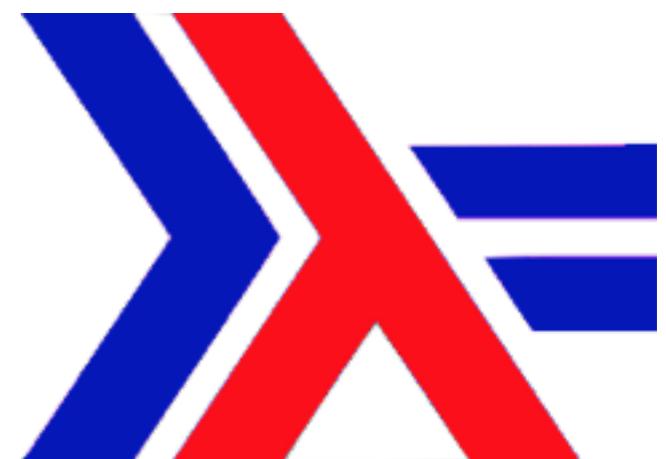
I. Static Checks: Fast & Safe Code

In theory Haskell is already SAFE!

Haskell

Functional Programming Language

Safe because of
Strong Typing + λ -calculus

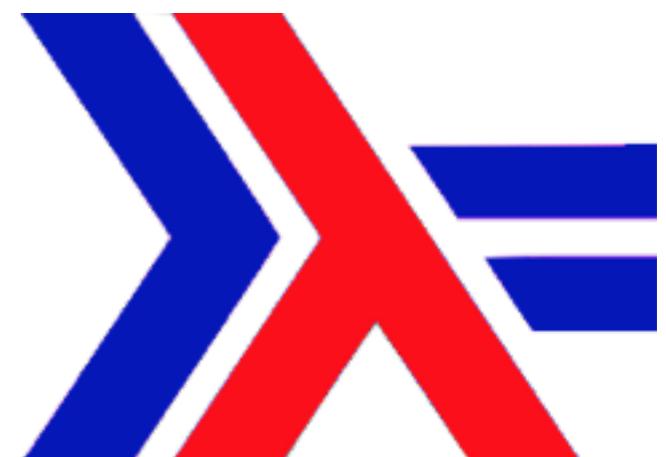


Haskell

Practical Programming Language

Fast Memory Manipulation with C wrappers

Allowing Overread Bugs



The Heartbleed Bug



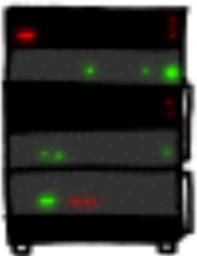
Buffer overread in OpenSSL. 2015

HOW THE HEARTBLEED BUG WORKS:

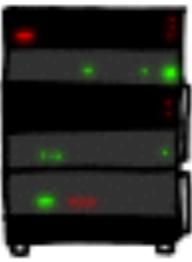
SERVER, ARE YOU STILL THERE?
IF SO, REPLY "POTATO" (6 LETTERS).



User Eric wants pages about "boats". User Erica requests secure connection using key "4538538374224". User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 5130985733435. Macie (chrome user) sends this message: "H



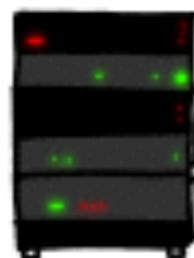
POTATO



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



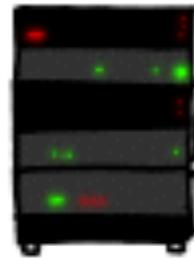
User Olivia from London wants pages about "new bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 345 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bfff84)



HMM...



BIRD



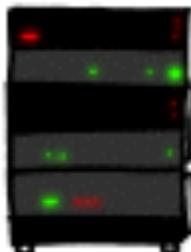
SERVER, ARE YOU STILL THERE?

a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas

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

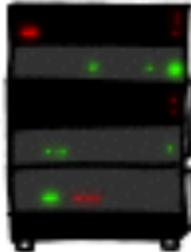


a connection. Jake requested pictures of deer.
User Meg wants these 500 letters: HAT. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
"snakes but not too long". User Karen wants to
change account password to "CoHoBaSt". User



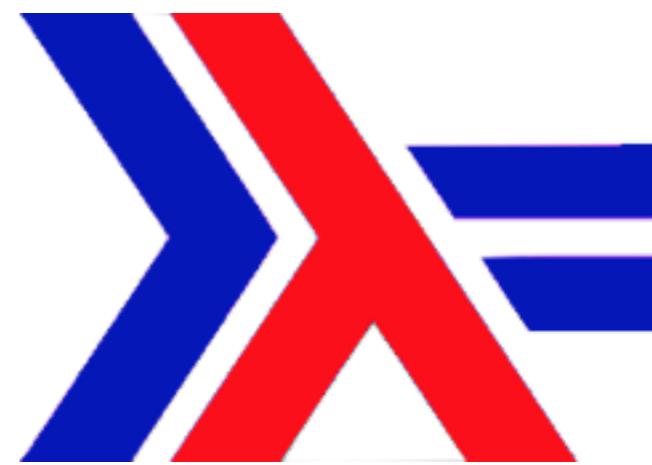
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User

a connection. Jake requested pictures of deer.
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 "CoHoBaSt". User





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



Liquid Haskell

Refinement Types



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 => $v = 500$ => $v < \text{len } x$

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 < 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



Checks valid arguments, under facts.

Refinement Types for Haskell, ICFP'14
by **Vazou**, Seidel, Jhala, Vytiniotis, and Peyton-Jones



Checks valid arguments, under facts.

Static Checks



Liquid Haskell

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

IV. Status: Industry & Education



Liquid Haskell

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

IV. Status: Industry & Education

II. Application: Speed up Parsing

Application: Speed up Parsing

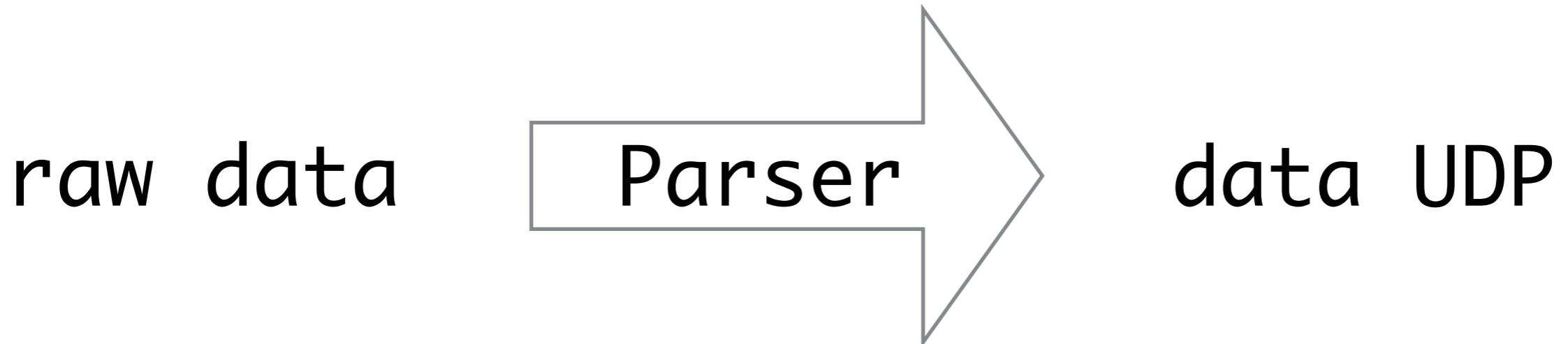
Simplified Application of Liquid Haskell

by Gabriel Gonzalez



Application: Speed up Parsing

UDP: User Datagram Protocol



Application: Speed up Parsing

```
data UDP = UDP
  { udpSrcPort :: Text -- 2 chars
  , udpDestPort :: Text -- 2 chars
  , udpLength :: Text -- 2 chars
  , udpChecksum :: Text -- 2 chars
  }
```

Application: Speed up Parsing

```
udpP :: Text -> UDP
udpP ts =
  let (udp1, ts1) = splitAt 2 ts
  let (udp2, ts2) = splitAt 2 ts1
  let (udp3, ts3) = splitAt 2 ts2
  let (udp4, ts4) = splitAt 2 ts3
in UDP udp1 upd2 upd3 upd4
```

Safe but Slow (4 runtime checks)

Solution: Merge checks

Application: Speed up Parsing

```
udpP :: Text -> Maybe UDP
udpP ts =
  if length ts >= 8 then
    let (udp1, ts1) = US.splitAt 2 ts
    let (udp2, ts2) = US.splitAt 2 ts1
    let (udp3, ts3) = US.splitAt 2 ts2
    let (udp4, ts4) = US.splitAt 2 ts3
    in Just (UDP udp1 upd2 upd3 upd4)
  else Nothing
```

Safe and Fast (1 runtime check!)

Application: Speed up Parsing

```
udpP :: Text -> Maybe UDP
udpP ts =
  if length ts >= 8 then
    let (udp1, ts1) = US.splitAt 2 ts
    let (udp2, ts2) = US.splitAt 2 ts1
    let (udp3, ts3) = US.splitAt 4 ts2
    let (udp4, ts4) = US.splitAt 2 ts3
    in Just (UDP udp1 upd2 upd3 upd4)
  else Nothing
```

Safe and Fast, but error prone

Application: Speed up Parsing

```
udpP :: Text -> Maybe UDP
udpP ts =
  if length ts >= 8 then
    let (udp1, ts1) = US.splitAt 2 ts
      in Just (UDP udp1 upd2 upd3 upd4)
  else Nothing

splitAt :: i:Int -> t:{i < len t} ->
(tl:{i = len tl}, tr:{len tr = len t - i})
```

Enforce Static Checks!

Application: Speed up Parsing

GHC OK, but Liquid Error

Error

```
udpP :: Text -> Maybe UDP
```

```
udpP ts =  
  if length ts >= 8 then  
    let (udp1, ts1) = US.splitAt 2 ts  
    let (udp2, ts2) = US.splitAt 2 ts1  
    let (udp3, ts3) = US.splitAt 4 ts2  
    let (udp4, ts4) = US.splitAt 2 ts3  
    in Just (UDP udp1 upd2 upd3 upd4)  
  else Nothing
```

Enforce Static Checks!

Application: Speed up Parsing



```
udpP :: Text -> Maybe UDP
udpP ts =
  if length ts >= 8 then
    let (udp1, ts1) = US.splitAt 2 ts
    let (udp2, ts2) = US.splitAt 2 ts1
    let (udp3, ts3) = US.splitAt 4 ts2
    let (udp4, ts4) = US.splitAt 2 ts3
    in Just (UDP udp1 upd2 upd3 upd4)
  else Nothing
```

Provably Correct & Faster (x6) Code!

Application: Speed up Parsing

```
udpP :: Text -> Maybe UDP
udpP ts =
  if length ts >= 8 then
    let (udp1, ts1) = US.splitAt 2 ts
    let (udp2, ts2) = US.splitAt 2 ts1
    let (udp3, ts3) = US.splitAt 4 ts2
    let (udp4, ts4) = US.splitAt 2 ts3
    in Just (UDP udp1 udp2 udp3 udp4)
  else Nothing
```

Provably Correct & Faster (x6) Code!

Haskell Code: SMT-Automatic Verification

Application: Speed up Parsing

Provably Correct & Faster (x6) Code!

SMT-Automatic Verification

SMT-Automatic Verification

How expressive can we get?



Liquid Haskell

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

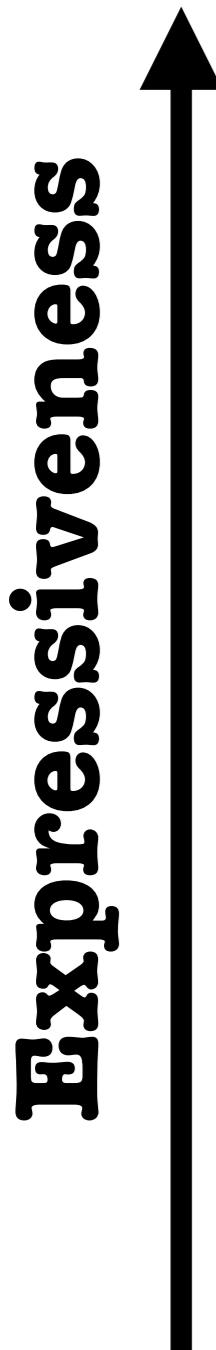
III. Expressiveness: Theorem Proving

IV. Status: Industry & Education

III. Expressiveness: Theorem Proving

Expressiveness : Theorem Proving using SMT-Verification

SMT-Verification



Critical Properties (Goal):

“The plane will not crash!”

Theorem Proving (POPL’18):

“If $f(x) < f(x+1)$, then f is monotonic”

Safe Indexing (Haskell’14):

“Your passwords are safe!”

SMT-Verification

Refinement Reflection, POPL'18

for Theorem Proving in Haskell!

“If $f(x) < f(x+1)$, then f is monotonic”

Fibonacci in Haskell

```
fib :: i:{Int | 0≤i} -> {v:Int | 0<v ∧ i≤v}
fib i
| i≤1          = 1
| otherwise     = fib (i-1) + fib (i-2)
```

Fibonacci in Haskell

```
fib :: i:{Int | 0≤i} -> {v:Int | 0<v ∧ i≤v}
fib i
| i≤1          = 1
| otherwise     = fib (i-1) + fib (i-2)
```

How to express **theorems** about functions?

\forall i. $0 \leq i \Rightarrow \text{fib } i \leq \text{fib } (i+1)$

How to express **theorems** about functions?

Step 1: Definition

In SMT **fib** is “Uninterpreted Function”

\forall i j. i = j => fib i = fib j

How to connect logic **fib** with Haskell **fib**?

How to connect logic fib with Haskell fib?

```
fib :: i:{Int | 0≤i} -> {v:Int | 0<v∧i≤v}
fib i
| i≤1          = 1
| otherwise    = fib (i-1) + fib (i-2)
```

Not Decidable!

SMT Axiom

\forall i.

if $i \leq 1$ then $\text{fib } i = 1$
else $\text{fib } i = \text{fib } (i-1) + \text{fib } (i-2)$

How to connect logic fib with Haskell fib?

```
fib :: i:{Int | 0≤i} -> {v:Int | 0<v ∧ i≤v}
fib i
| i≤1          = 1
| otherwise     = fib (i-1) + fib (i-2)
```

Refinement Reflection

```
fib :: i:{Int | 0≤i} -> {v:Int | v=fib i ∧
    if i≤1 then fib i = 1
    else fib i = fib (i-1) + fib (i-2)
}
```

Refinement Reflection

Step 1: Definition

Step 2: Reflection

```
fib :: i:{Int | 0≤i} -> {v:Int | v=fib i ∧
    if i≤1 then fib i = 1
    else fib i = fib (i-1) + fib (i-2)
}
```

Refinement Reflection

Step 1: Definition

Step 2: Reflection

```
fib :: i:{Int | 0≤i} -> {v:Int | v=fib i ∧  
    if i≤1 then fib i = 1  
    else fib i = fib (i-1) + fib (i-2)  
}
```

Step 3: Application

```
fib 0 :: {v:Int | v=fib 0 ∧ fib 0 = 1}
```

Application is Type Level Computation

fib 0

fib 0 = 1

Application Type Level Computation

fib 0

fib 0 = 1

fib 1

fib 1 = 1

fib 2

fib 2 = fib 1 + fib 0

fib i

?

- ? if $i \leq 1$ then fib i = 1
- ? else fib i = fib (i-1) + fib (i-2)

Application Type Level Computation

fib 0

fib 0 = 1

fib 1

fib 1 = 1

fib 2

fib 2 = fib 1 + fib 0

if 1 < i then

fib i

fib i = fib (i-1) + fib (i-2)

if $i \leq 1$ then fib i = 1

else fib i = fib (i-1) + fib (i-2)

Application Type Level Computation

fib 0

fib 0 = 1

fib 1

fib 1 = 1

fib 2

fib 2 = fib 1 + fib 0

if 1 < i then

fib i

fib i = fib (i-1) + fib (i-2)

fib (i+1)

fib (i+1) = fib i + fib (i-1)

Theorem Proving

```
fibUp :: i:Nat -> {fib i ≤ fib (i+1)}
fibUp i
| i == 0
= fib 0 <. fib 1
*** QED
| i == 1
= fib 1 <=. fib 1 + fib 0 <= fib 2
*** QED
| otherwise
= fib i
<=. fib i + fib (i-1)
<=. fib (i+1)
*** QED
```

Reflection for Theorem Proving

Theorems are refinement types.

Proofs are functions.

Check that functions prove theorems.

Proofs are functions.

`fibUp :: i:Nat -> {fib i ≤ fib (i+1)}`

Proofs are functions.

`fibUp :: i:Nat -> {fib i ≤ fib (i+1)}`

Let's call them!

`fibUp 4 :: {fib 4 ≤ fib 5}`

`fibUp i :: {fib i ≤ fib (i+1)}`

`fibUp (j-1) :: {fib (j-1) ≤ fib j}`

Proofs are functions. Let's call them!

```
fibMono :: i:Nat -> j:{Nat | i < j}  
         -> {fib i ≤ fib j}
```

```
fibMono i j  
| i + 1 == j  
= fib i  
<=. fib (i+1) ? fibUp i  
==. fib j  
*** QED  
| otherwise  
= fib i  
<=. fib (j-1) ? fibMono i (j-1)  
<=. fib j      ? fibUp (j-1)  
*** QED
```

Proofs are functions. Let's call them!

```
fibMono :: i:Nat -> j:{Nat | i < j}  
         -> {fib i ≤ fib j}
```

```
fibMono i j  
| i + 1 == j  
= fib i  
<=. fib (i+1) ? fibUp i  
==. fib j  
*** QED  
| otherwise  
= fib i  
<=. fib (j-1) ? fibMono i (j-1)  
<=. fib j      ? fibUp (j-1)  
*** QED
```

Proofs are functions. Let's abstract them!

```
fibMono :: i:Nat -> j:{Nat | i < j}
          -> fib:(Nat -> Int)
          -> (k:Nat -> {fib k ≤ fib (k+1)})
          -> {fib i ≤ fib j}
```

```
fibMono i j fib fibUp
| i + 1 == j
= fib i
<=. fib (i+1) ? fibUp i
==. fib j
*** QED
| otherwise
= fib i
<=. fib (j-1) ? fibMono i (j-1)
<=. fib j      ? fibUp (j-1)
*** QED
```



Liquid Haskell

Refinement Reflection for **Expressiveness**

 Liquid Haskell

Refinement Reflection for **Expressiveness**

Idea:

Encode HO specs in FO SMT decidable logic

Metatheory:

THEOREM 4.1. [Soundness of λ^R]

- **Denotations** If $\Gamma; R \vdash p : \tau$ then $\forall \theta \in [\![\Gamma]\!]. \theta \cdot p \in [\![\theta \cdot \tau]\!]$.
- **Preservation** If $\emptyset; \emptyset \vdash p : \tau$ and $p \hookrightarrow^* w$, then $\emptyset; \emptyset \vdash w : \tau$.

E PROOF OF SOUNDNESS

We prove Theorem 4.1 of § D by reduction to Soundness of λ^U [Vazou et al. 2014a].

THEOREM E.1. [Denotations] If $\Gamma \vdash p : \tau$ then $\forall \theta \in [\Gamma]. \theta \cdot p \in [\theta \cdot \tau]$.

PROOF. We use the proof from [Vazou et al. 2014b] and specifically Lemma 4 that is identical to the statement we need to prove. Since the proof proceeds by induction in the type derivation, we need to ensure that all the modified rules satisfy the statement.

- T-EXACT Assume $\Gamma \vdash e : \{v : B \mid \{r\} \wedge v = e\}$. By inversion $\Gamma \vdash e : \{v : B \mid \{r\}\}(1)$. By (1) and IH we get $\forall \theta \in [\Gamma]. \theta \cdot e \in [\theta \cdot \{v : B \mid \{r\}\}]$. We fix a $\theta \in [\Gamma]$. We get that if $\theta \cdot e \rightsquigarrow^* w$, then $\theta \cdot \{r\}[v/w] \rightsquigarrow^* \text{True}$. By the Definition of $=$ we get that $w = w \rightsquigarrow^* \text{True}$. Since $\theta \cdot (v = e)[v/w] \rightsquigarrow^* w = w$, then $\theta \cdot (\{r\} \wedge v = e)[v/w] \rightsquigarrow^* \text{True}$. Thus $\theta \cdot e \in [\theta \cdot \{v : B \mid \{r\} \wedge v = e\}]$ and since this holds for any fixed θ , $\forall \theta \in [\Gamma]. \theta \cdot e \in [\theta \cdot \{v : B \mid \{r\} \wedge v = e\}]$.
- T-LET Assume $\Gamma \vdash \text{let } \text{rec } x : \tau_x = e_x \text{ in } p : \tau$. By inversion $\Gamma, x : \tau_x \vdash e_x : \tau_x(1), \Gamma, x : \tau_x \vdash p : \tau(2)$, and $\Gamma \vdash \tau(3)$. By IH $\forall \theta \in [\Gamma, x : \tau_x]. \theta \cdot e_x \in [\theta \cdot \tau_x](1) \forall \theta \in [\Gamma, x : \tau_x]. \theta \cdot p \in [\theta \cdot \tau](2)$. By (1') and (2') and (3) $\forall \theta \in [\Gamma]. \theta \cdot p \in [\Gamma]$.
- T-REFL Assume $\Gamma \vdash e \in p : \tau$. By IH we get that all the modified rules are closed under evaluation.
- T-FIX In Theorem 4.1 of [Vazou et al. 2014a] and Paolini 2004

THEOREM E.2. [Preservation]

PROOF. In [Vazou et al. 2014b] we prove that Type Preservation is ensured by Lemma 7. Thus, it suffices to show that all the modified rules satisfy Lemma 7.

LEMMA E.3. If $\emptyset \vdash p : \tau$ and $p \rightsquigarrow p'$ then $\emptyset \vdash p' : \tau$.

PROOF. Since Type Preservation in λ^U is proved by induction on the type derivation tree, we need to ensure that all the modified rules satisfy the statement.

- T-EXACT Assume $\emptyset \vdash p : \{v : B \mid \{r\} \wedge v = p\}$. By inversion $\emptyset \vdash p : \{v : B \mid \{r\}\}$. By IH we get $\emptyset \vdash p' : \{v : B \mid \{r\}\}$. By rule T-EXACT we get $\emptyset \vdash p' : \{v : B \mid \{r\} \wedge v = p'\}$. Since subtyping is closed under evaluation, we get $\emptyset \vdash \{v : B \mid \{r\} \wedge v = p'\} \leq \{v : B \mid \{r\} \wedge v = p\}$. By rule T-SUB we get $\emptyset \vdash p' : \{v : B \mid \{r\} \wedge v = p\}$.

By 25, the above set is not empty, and hence τ is valid under d . \square

Example: Fibonacci is increasing In § 2 we verified that under a definition d that includes fib, the term fibUp proves

$$n : \text{Nat} \rightarrow \{\text{fib } n \leq \text{fib } (n + 1)\}$$

Thus, by Theorem D.3 we get

$$\forall n. 0 \leq n \rightsquigarrow^* \text{True} \Rightarrow \text{fib } n \leq \text{fib } (n + 1) \rightsquigarrow^* \text{True}$$

for all assignments $\sigma \models p$.

Embedding Functions As λ^P is a first-order logic, we embed λ -abstraction and application using the uninterpreted functions lam and app. We embed λ -abstractions using lam as shown in

COROLLARY E.5. If $\Gamma \vdash e : \text{Bool}$, e reduces to a value and $\Gamma \vdash e \rightsquigarrow p$, then for every $\theta \in [\Gamma]$ and every $\sigma \in \theta^\perp$, $\theta^\perp \cdot e \rightsquigarrow^* \text{True}$ iff $\sigma^\beta \models p$.

referring to the proofs in [Vazou et al. 2014b].

$\Gamma \vdash e \rightsquigarrow p$

sort Fun s_x s,
action of sort
er and body,
the binder x
ynolds 1972]
 $e e'$, where e

$\Gamma \vdash s \tau$

include

$p, \Gamma \vdash e' \rightsquigarrow$

by inversion
 $\rightsquigarrow p_i$. By IH
 $\models p_n(\tau)$

$\vdash e \Gamma \vdash e \rightsquigarrow p$,
 $D \bar{p}_i \bar{p} \bar{p}_j =$
fully applied

β is defined
 $\vdash \Gamma \vdash e \rightsquigarrow p$,
 $D_j \bar{p}^j$ we get

identity of
the $\Gamma \vdash e_x \rightsquigarrow$

\bar{y} .
 $\vdash \text{app } D \bar{y}, (\bar{y}_i,$

approximated, as
 \square

$(v : B \mid e_2)]$.
true.
 \square

subsumption
m that enjoys

decidable logic
[Barrett et al.

constructors D
on $x \bar{p}$ of an

representing
 $d <$, have the

expressions, are

represents all

ables to terms

intuitively if

te p is valid if

) to ensure
ing [Vazou
rules to use
].

et of values
to logic by
Vazou et al.

iting boolean
equivalence.
d app satisfy

es. These ele-
* ar), binary
trying, and so
s to primitive
e level terms

empty environ-
"bottom" in
et al. 2014a].

placing the
hat uses an
oundness of

ool. By IH,

$\vdash e_2 \rightsquigarrow p_2$.
 $p_1 : (\tau_1)$ and

$v \equiv \text{False}$,

ns of § 5 by
nt.
(x). But by

$\vdash e \dots \text{else } p_n$
 $\vdash p$ and $\Gamma \vdash$
and selectors,

$\vdash \Gamma \vdash e \rightsquigarrow p$,
 $D_j \bar{p}^j$ we get

$\vdash \tau \in [\theta^\perp]$, if

Vazou et al.
tions,

thus it has a
since v is a

$\vdash \text{app } D \bar{y}, (\bar{y}_i,$

$\vdash \bar{y}$.
 $\vdash \text{app } D \bar{y}, (\bar{y}_i,$

approximated, as
 \square

$\vdash v \equiv \text{False}$,

ion $\theta \in [\Gamma]$
 $\sigma^\beta \in [\theta^\perp]$.



Liquid Haskell as a Theorem Prover

Refinement Reflection, POPL'18

by **Vazou**, Tondwalkar, Choudhury, Scott,
Newton, Wadler, and Jhala

A Tale of Two Provers, Haskell'17

by **Vazou**, Lampropoulos, and Polakow

 Liquid Haskell as a Theorem Prover

“Verified Parallelization of String Matching”

A Tale of Two Provers, Haskell’17

by **Vazou**, Lampropoulos, and Polakow



Liquid Haskell

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

IV. Status: Industry & Education



Liquid Haskell

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

IV. Status: Industry & Education

IV. Status: Industry & Education



Liquid Haskell on papers



Refinement Reflection, POPL'18

by **Vazou**, Tondwalkar, Choudhury, Scott,
Newton, Wadler, and Jhala

Refinement Types for Haskell, ICFP'14

by **Vazou**, Seidel, Jhala, Vytiniotis, and Peyton-Jones



Liquid Haskell on papers



Liquid Haskell on github

ucsd-progsys / liquidhaskell

Unwatch 22 Star 473 Fork 75

Code Issues 201 Pull requests 5 Projects 0 Wiki Insights Settings

Liquid Types For Haskell Edit

Add topics

8,382 commits 94 branches 19 releases 38 contributors

Branch: develop New pull request Create new file Upload files Find file Clone or download

nikivazou Merge pull request #1223 from ucsd-progsys/HaskellFunInRefs ... Latest commit 82f5baa 5 days ago

benchmarks move tests into pos a month ago

devel Fix travis error 6 days ago

Liquid Haskell main dev team



Niki Vazou & Ranjt Jhala,
Liquid Haskell dev team

Liquid Haskell in the real world

Education



Will Kunkel,
undergrad @UMD



Rachel Xu & Xinyue Zhang,
undergrads @Bryn Mawr College

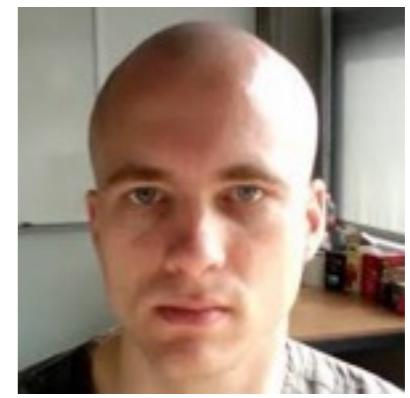


Niki Vazou & Ranjt Jhala,
Liquid Haskell dev team

Industry



Gabriel Gonzalez
Awake Security



Edsko de Vries
Well-Typed

Liquid Haskell in Education

This is Will Kunkel.

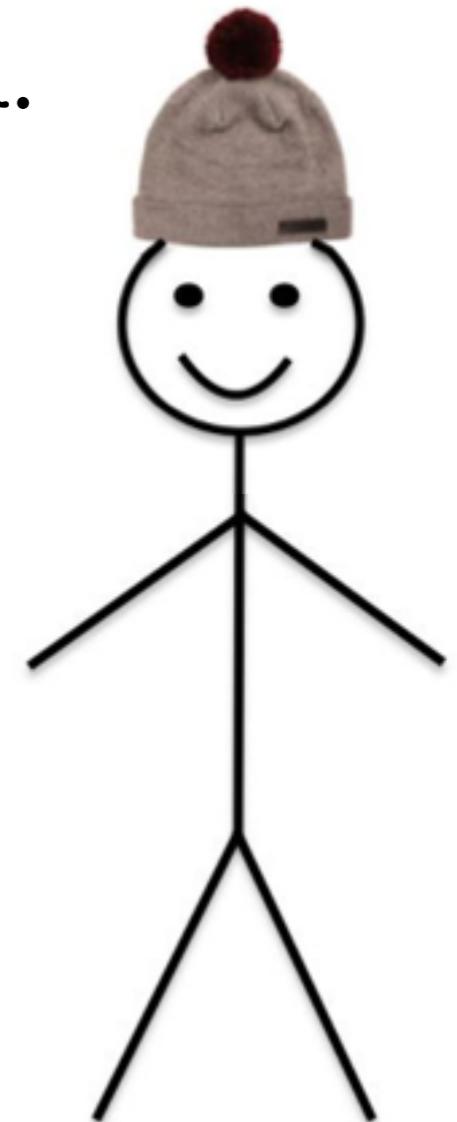
Will took my undergrad course on Haskell.

Will learnt Liquid Haskell in 2 weeks.

Will was **3rd** in POPL'18 **Student Research Competition** with his project “Comparing Liquid Haskell and Coq”

Will is smart.

Be like Will.



Liquid Haskell in Education

Two winners in POPL'18 Student Research Competition



“Comparing Liquid Haskell and Coq”



Will Kunkel,
undergrad @UMD



“Comparing Dependent Haskell,
Liquid Haskell and F*”



Rachel Xu & Xinyue Zhang,
undergrads @Bryn Mawr College

Liquid Haskell in Education

Two winners in POPL'18 Student Research Competition



“Comparing Liquid Haskell and Coq”



Will Kunkel,
undergrad @UMD



“Comparing Dependent Haskell,
Liquid Haskell and F*”



Rachel Xu & Xinyue Zhang,
undergrads @Bryn Mawr College



Liquid Haskell in the real world

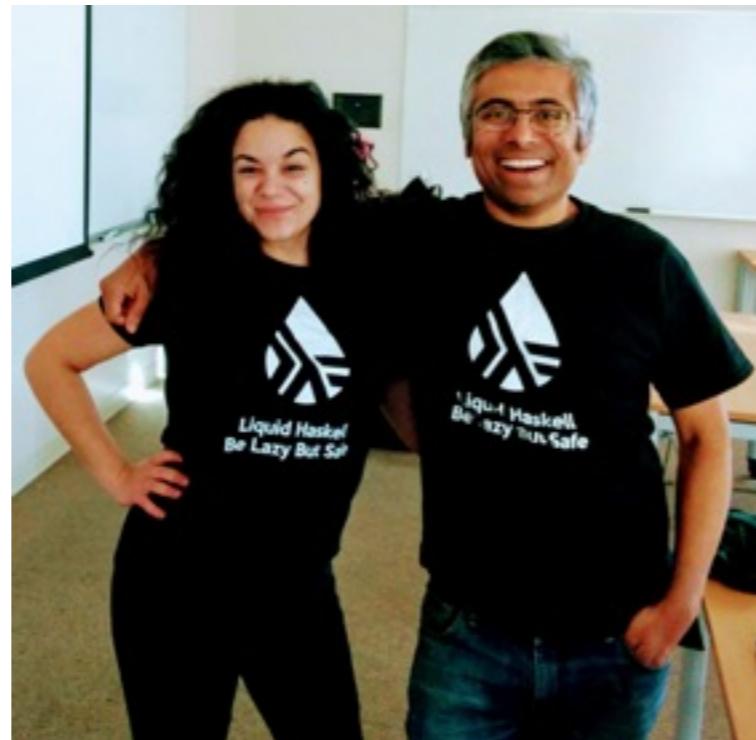
Education



Will Kunkel,
undergrad @UMD



Rachel Xu & Xinyue Zhang,
undergrads @Bryn Mawr College



Niki Vazou & Ranjt Jhala,
Liquid Haskell dev team



Gabriel Gonzalez
Awake Security



Edsko de Vries
Well-Typed



Liquid Haskell in Industry

Gabriel Gonzalez



Static checks during parsing.

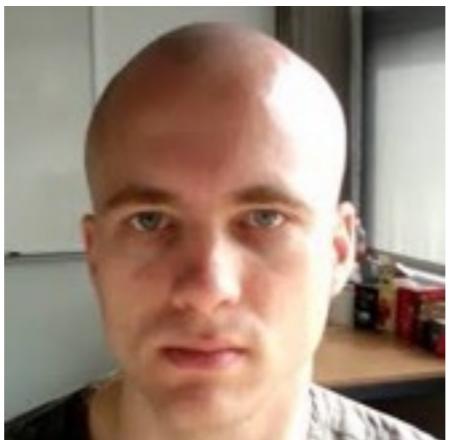
Provably Correct & Faster (x6) Code!

Huge speedup for internet traffic parsing.



Liquid Haskell in Industry

Gabriel Gonzalez

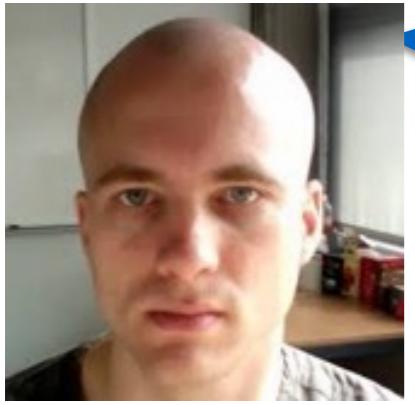


Edsko de Vries





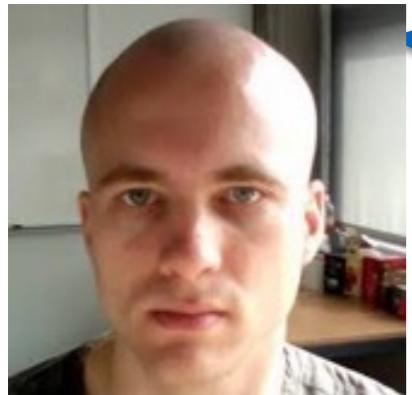
I am a Haskell consultant at a
cryptocurrency/blockchain company.
We have a blockchain algorithm written
in paper & a Haskell implementation.
We want Liquid Haskell to connect them.



I am a Haskell consultant at a
cryptocurrency/blockchain company.
We have a blockchain algorithm written
in paper & a **Haskell implementation**.
We want Liquid Haskell to connect them.

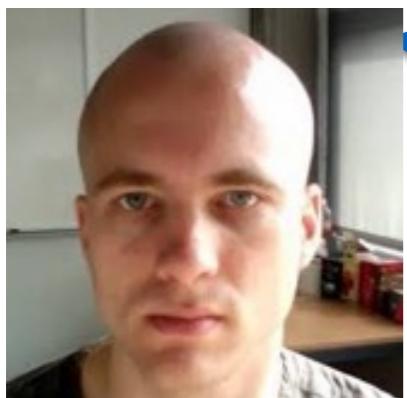
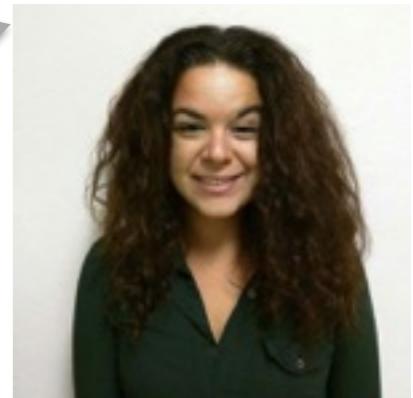
Awesome! Lmk if you need anything!



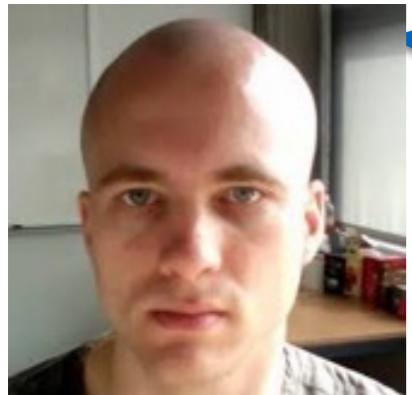


I am a Haskell consultant at a
cryptocurrency/blockchain company.
We have a blockchain algorithm written
in paper & a Haskell implementation.
We want Liquid Haskell to connect them.

Awesome! Lmk if you need anything!

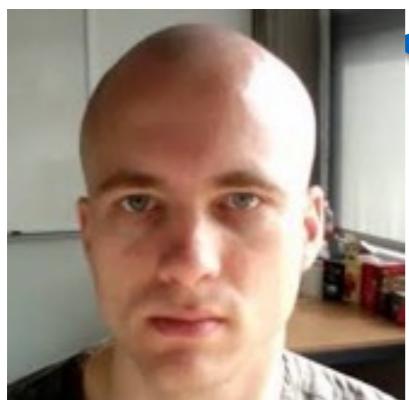
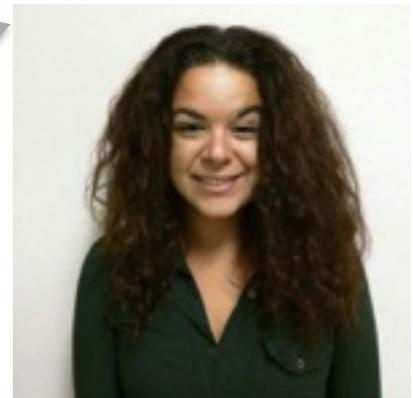


I want **interactive proof generation!**



I am a Haskell consultant at a
cryptocurrency/blockchain company.
We have a blockchain algorithm written
in paper & a **Haskell implementation**.
We want Liquid Haskell to connect them.

Awesome! Lmk if you need anything!



I want **interactive proof generation**!

Liquid Haskell has many users

7,765 downloads



many users, but two main devs



many users, but two main devs



Goal: Expand the Liquid Haskell Team

Goal: Expand the Liquid Haskell Team

To formalize & implement research ideas

e.g.,

compiler optimizations,
proof assistance,
real system verification.

Vision:
Embed Verification in Haskell Programming

Vision: Embed Verification in Haskell Programming

Specs are just comments **inside** the languages, ...
thus, learning effort is small.

Semi-**automatically** machine checked, via SMTs.
For runtime **optimizations**, by the user or the **compiler**.

Vision: Embed Verification in ~~Haskell~~-Programming mainstream

Ruby, JavaScript, Scala, ...

Refinement Types for Ruby

Refinement Types for TypeScript

SMT-Based Checking of Predicate-Qualified Types for Scala

Georg Stefan Schmid Viktor Kuncak

EPFL, Switzerland

{firstname.lastname}@epfl.ch

Univ



Liquid Haskell

Embed Verification in Haskell Programming

I. Static Checks: Fast & Safe Code

II. Application: Speed up Parsing

III. Expressiveness: Theorem Proving

IV. Status: Industry & Education

Thanks!