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.

Inria

Microsoft Research

Carnegie Mellon University

Verification Prevents Bugs!

Prove end-to-end Correctness of Whole Systems

Princeton

Penn

Yale

MIT

But Requires Experts

Princeton Penn Yale MIT

My Goal:

Make Verification Accessible

My Approach:

Turn a Language into a Verifier

Haskell

General Purpose Programming Language

1 Python

...

25 Haskell

...

36 Erlang

...

45 OCaml

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

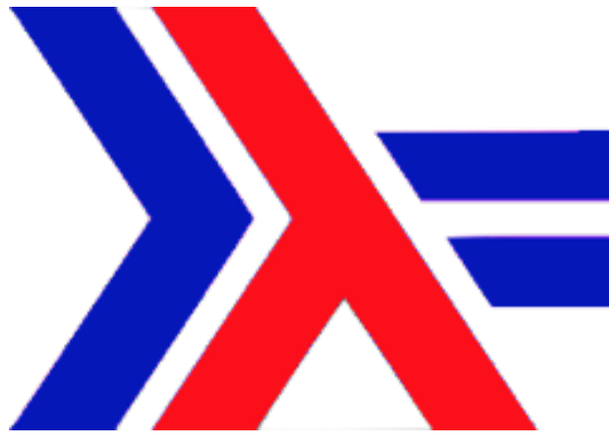
In theory Haskell is already SAFE!

Haskell

Functional Programing Language

Safe because of

Strong Typing + λ-calculus



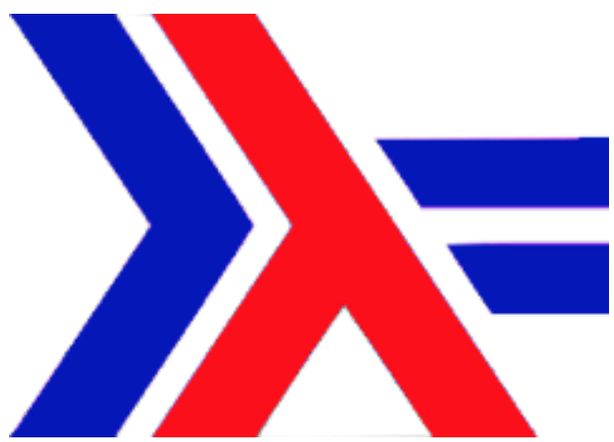
Haskell

Practical Programing Language

Fast Memory Manipulation

with C wrappers

Allowing Overread Bugs

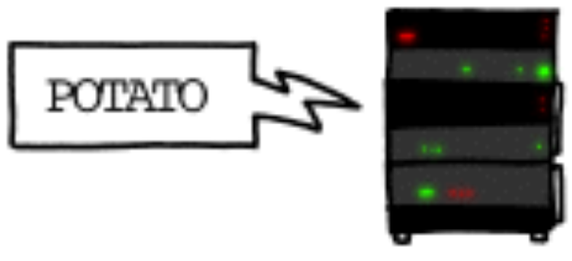


The Heartbleed Bug

Buffer overread in OpenSSL. 2015

HOW THE HEARTBLEED BUG WORKS:

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



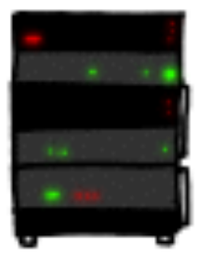
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

User Olivia from London wants pages about "man 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: 834ba962e2cab9ff89bd3bff8

HMM...

User Olivia from London wants pages about "man 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: 834ba962e2cab9ff89bd3bff8

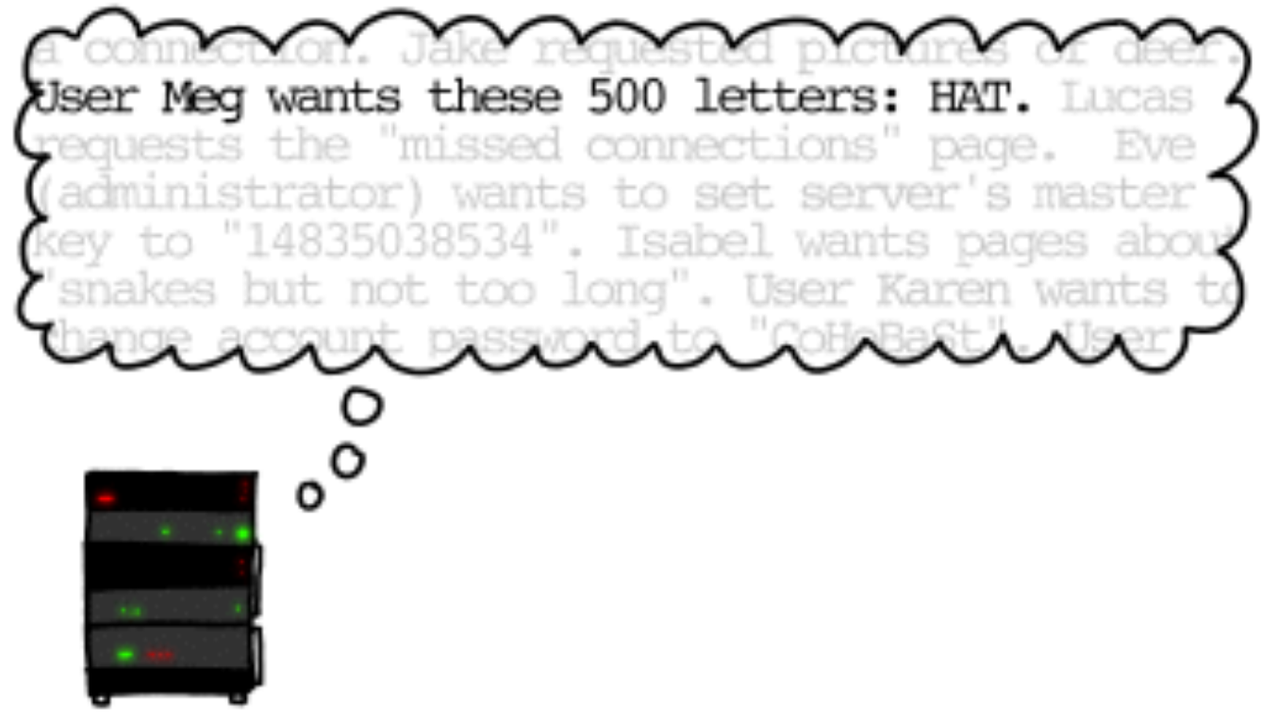
BIRD

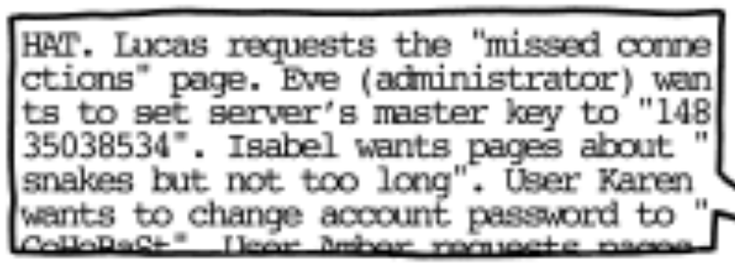


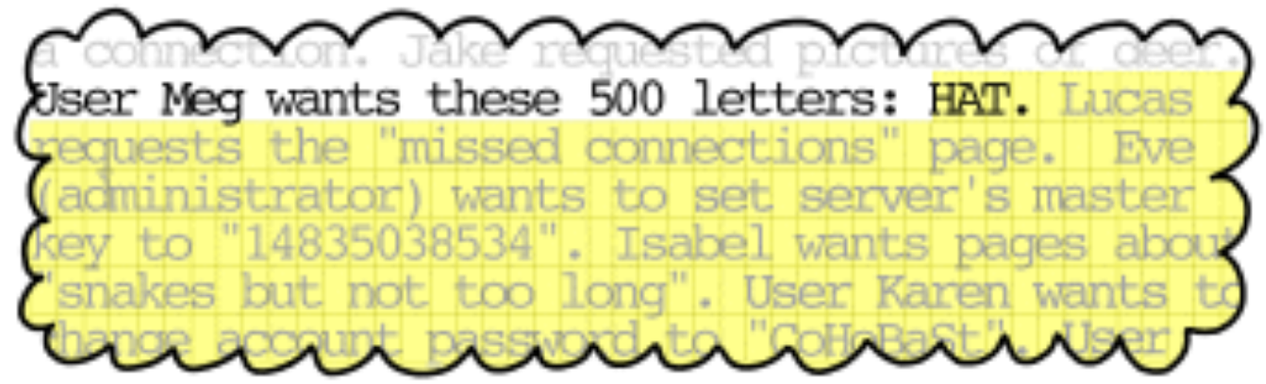
SERVER, ARE YOU STILL THERE?

User Meg wants these 500 letters: HAT. Lucas

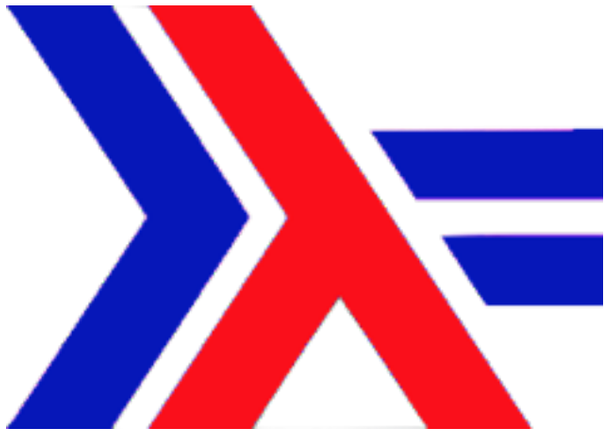
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).







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!”
```

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

Overread

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 < 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

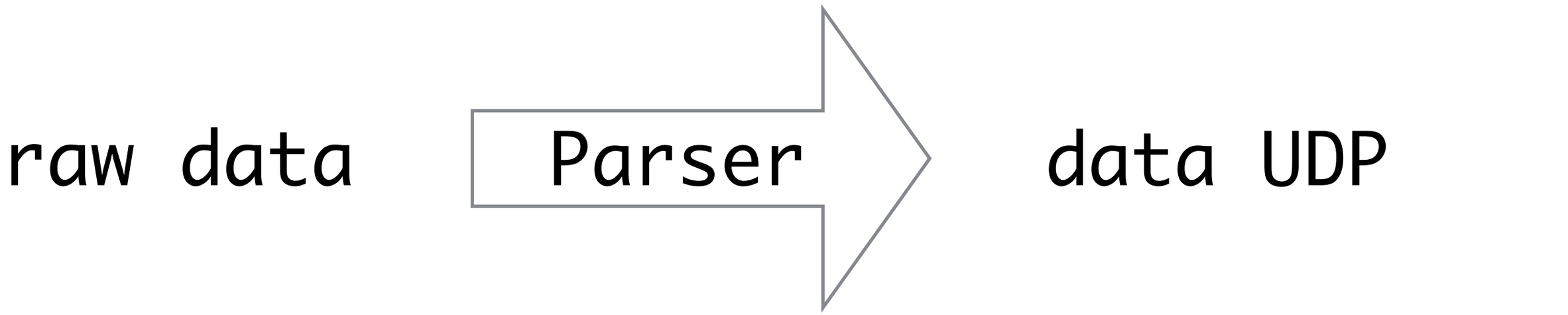
Application: Speed up Parsing

Simplified Application of Liquid Haskell

by Gabriel Gonzalez

Application: Speed up Parsing

UDP: User Datagram Protocol



Application: Speed up Parsing

```
data UDP = UDP
    { udpSrcePort :: 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 udp3 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 udp3 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 udp3 upd4)
else Nothing
```

Safe and Fast, bur 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 udp3 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 Par

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 udp3 upd4)
  else Nothing
```

Enforce Static Checks!

Application: Speed up Parsing

OK

```
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 udp3 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 upd2 udp3 upd4)
  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

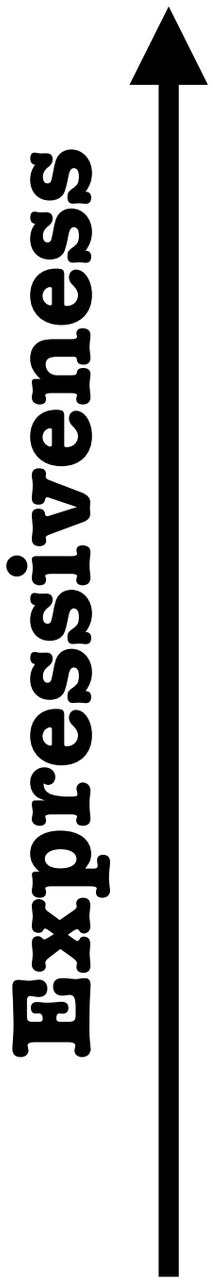
III. Expressiveness: Theorem Proving

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 ≤ i => fib i ≤ 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≤1 then fib i = 1
      else fib i = fib (i-1) + 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≤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≤1 then fib i = 1
else fib i = fib (i-1) + fib (i-2)
```

Application

```
fib 0
```

```
fib 1
```

```
fib 2
```

```
if 1<i then
```

```
fib i
```

```
fib (i+1)
```

Type Level Computation

```
fib 0 = 1
```

```
fib 1 = 1
```

```
fib 2 = fib 1 + fib 0
```

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

```
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**

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* $p \hookrightarrow^\star w$, *then* $\emptyset; \emptyset \vdash w : \tau$.

Refinement Reflection, POPL'18

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

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

Expressiveness

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

Expressiveness

Refinement Reflection, POPL'18
by **Vazou**, Tondwalkar, Choudhury, Scott, Newton, Wadler, and Jhala

Bounded Refinement Types, ICFP'15
by **Vazou**, Bakst, and Jhala

Abstract Refinement Types, ESOP'13
by **Vazou**, Rondon, and Jhala

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

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

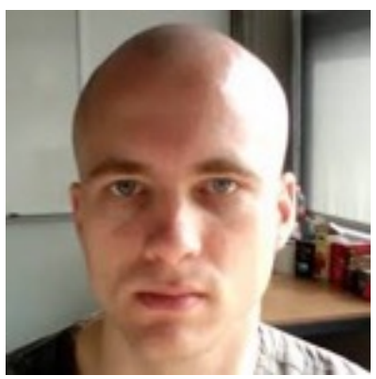
Will Kunkel,
undergrad @UMD

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

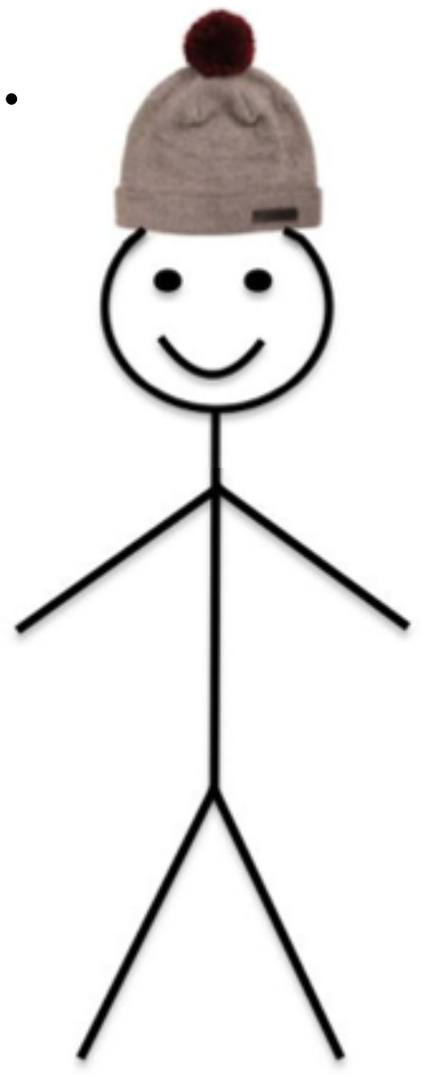
Liquid Haskell in Education

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

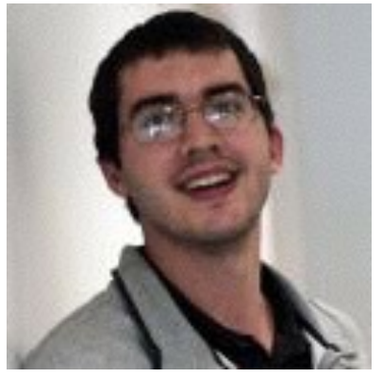
Education

Will Kunkel,
undergrad @UMD

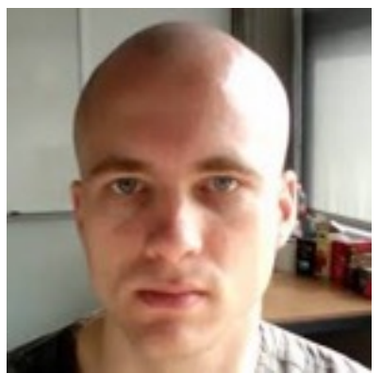
Rachel Xu & Xinyue Zhang,
undergrads @Bryn Mawr College

Niki Vazou & Ranjt Jhala,
Liquid Haskell dev team

Industry



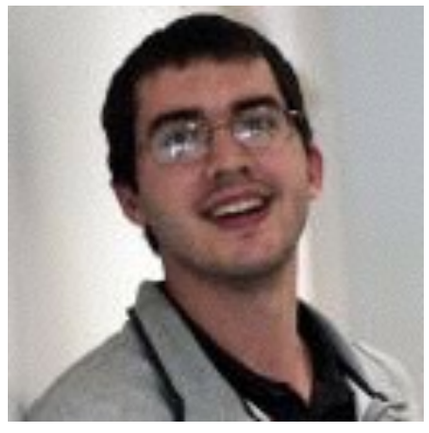
Gabriel Gonzalez
Awake Security



Edsko de Vries
Well-Typed

Liquid Haskell in Industry

Gabriel Gonzalez



AWAKE

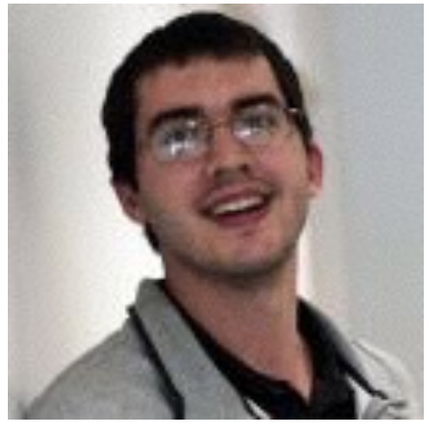
Static checks during parsing.

Provably Correct & Faster (x6) Code!

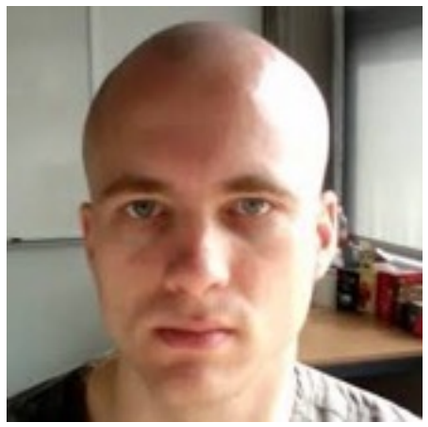
Huge speedup for internet traffic parsing.

Liquid Haskell in Industry

Gabriel Gonzalez

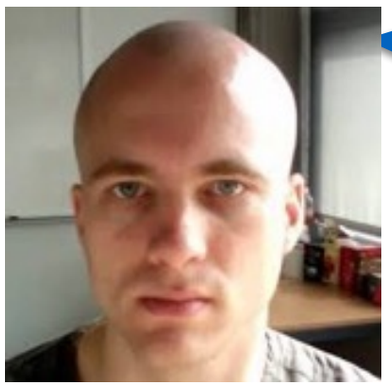


AWAKE

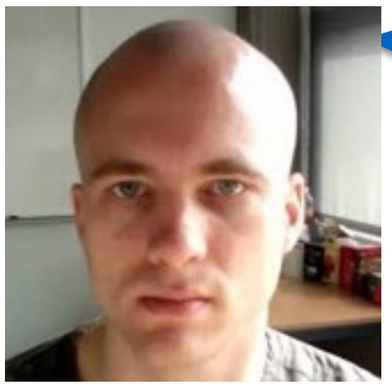


Well-Typed

Edsko de Vries

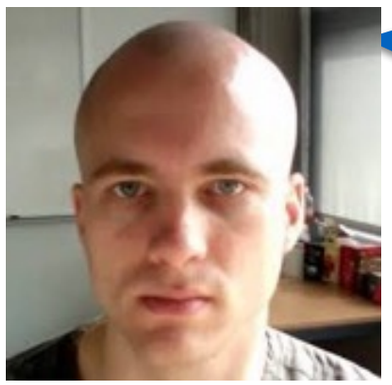


I am a Haskell consultant at a **cryptocurrency/blockchain** company.
We have a blockchain algorithm written **in paper** & a **Haskell implementation**.
We want Liquid Haskell to connect them.



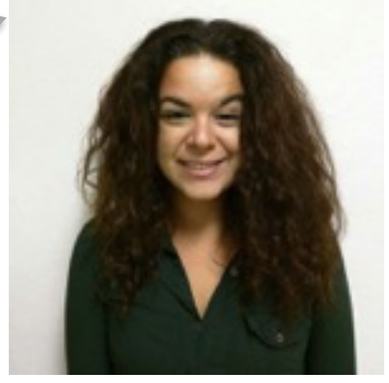
I am a Haskell consultant at a **cryptocurrency/blockchain** company. We have a blockchain algorithm written **in paper** & a **Haskell implementation**. We want Liquid Haskell to connect them.

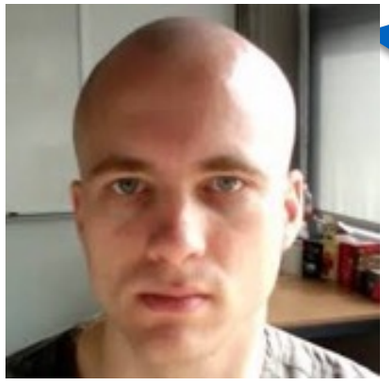
Awesome! Lmk if you need anything!



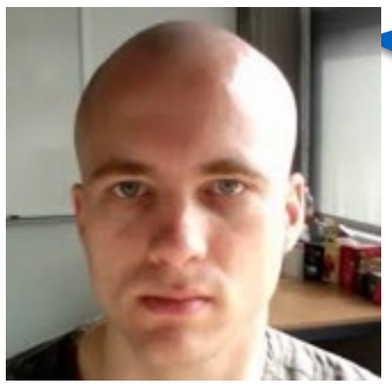
I am a Haskell consultant at a **cryptocurrency/blockchain** company. We have a blockchain algorithm written **in paper** & a **Haskell implementation**. We want Liquid Haskell to connect them.

Awesome! Lmk if you need anything!



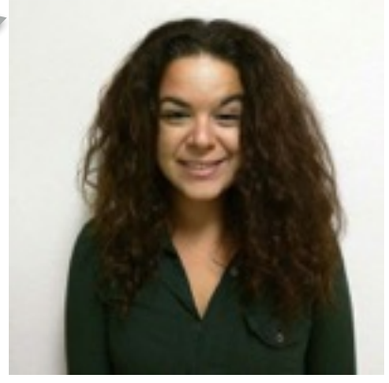


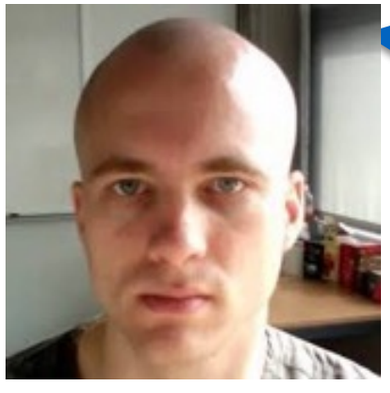
I want **interactive proof generation**!



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!



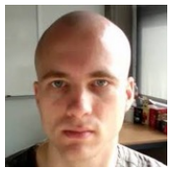


I want **interactive proof generation**!

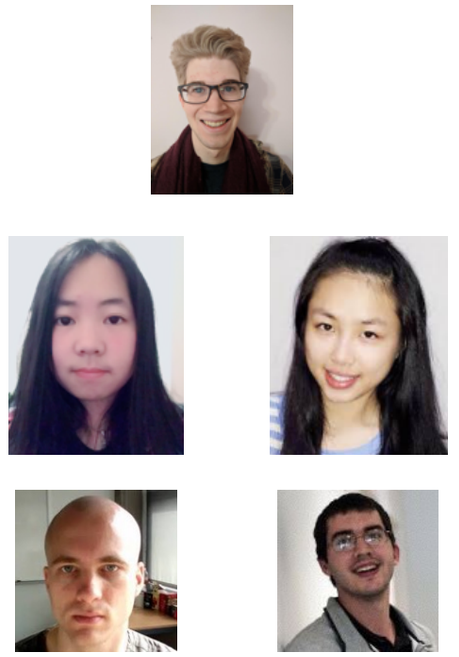
Liquid Haskell has many users

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.

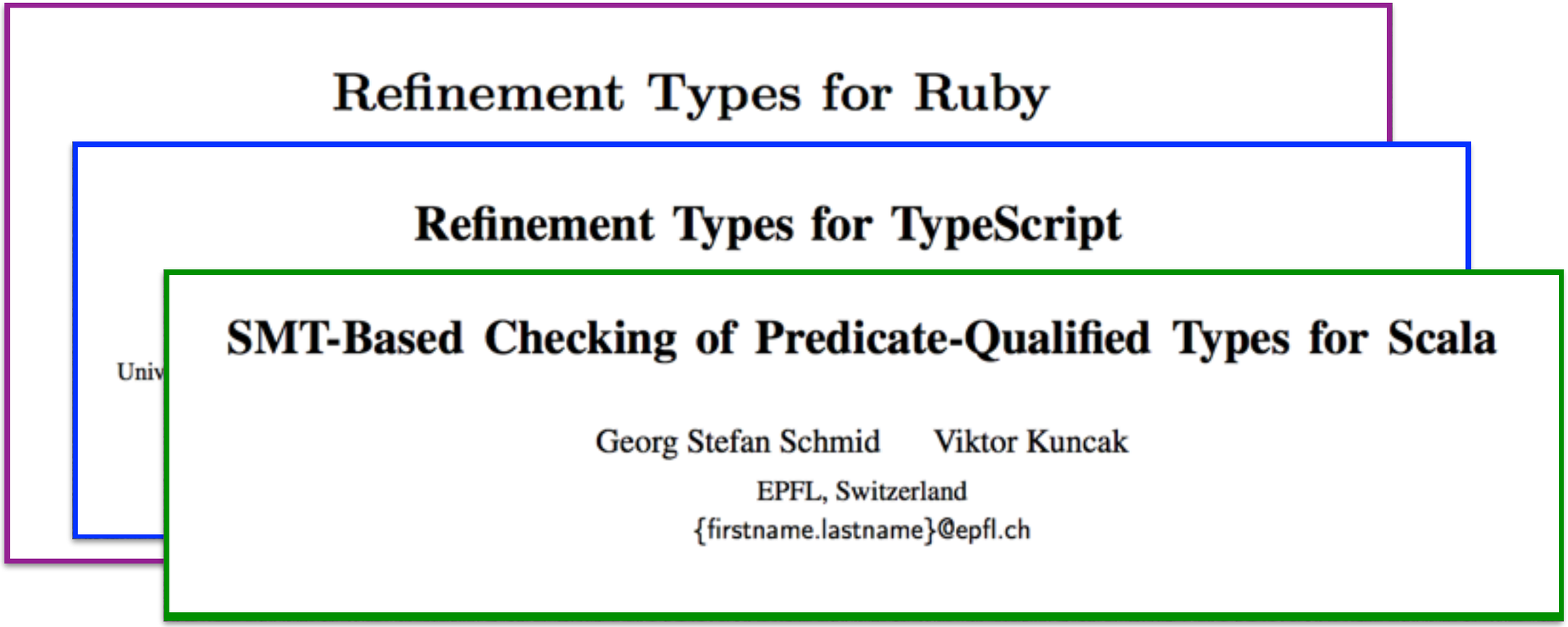
Semi-**automatically** machine checked, via SMTs.

For runtime **optimizations**, by the user or the compiler.

Vision:
Embed Verification in ~~Haskell~~ Programming

mainstream

Ruby, **JavaScript**, **Scala**, ...



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!