

# Liquid Haskell

## Haskell as a Theorem Prover

Niki Vazou  
UC San Diego

# Software bugs are everywhere



On a more serious note, software bugs can lead to deaths, which is what happened in May 2015, when Airbus crashed in Spain due to a bug in the software control engine.

4 deaths, Spain, software control

Airbus A400M crashed due to a software bug.

— May 2015

# Software bugs are everywhere



Heartbleed is another bug that got publicity last year, partially because of its pretty logo.

that due to buffer overflow attackers could get access to encrypted information.

Information protected by the SSL protocol is stored next to unprotected information. Due to buffer overflow attackers can steal encrypted information, like user passwords.

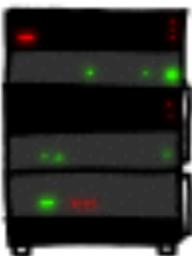
The Heartbleed Bug.  
Buffer overflow in OpenSSL. 2015

# HOW THE HEARTBLEED BUG WORKS:

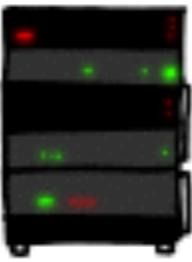
SERVER, ARE YOU STILL THERE?  
IF SO, REPLY "POTATO" (6 LETTERS).



User 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



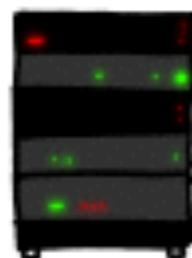
POTATO



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



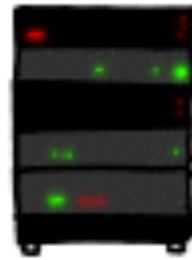
User Olivia from London wants pages about "new bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 345 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bfff84)



HMM...



BIRD



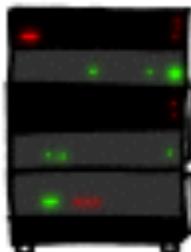
SERVER, ARE YOU STILL THERE?

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).

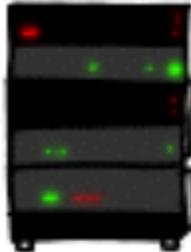


a connection. Jake requested pictures of deer.  
User Meg wants these 500 letters: HAT. Lucas  
requests the "missed connections" page. Eve  
(administrator) wants to set server's master  
key to "14835038534". Isabel wants pages about  
"snakes but not too long". User Karen wants to  
change account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



# Make bugs difficult to express

Using Modern Programming Languages

F#, Ocaml, Scala, Haskell

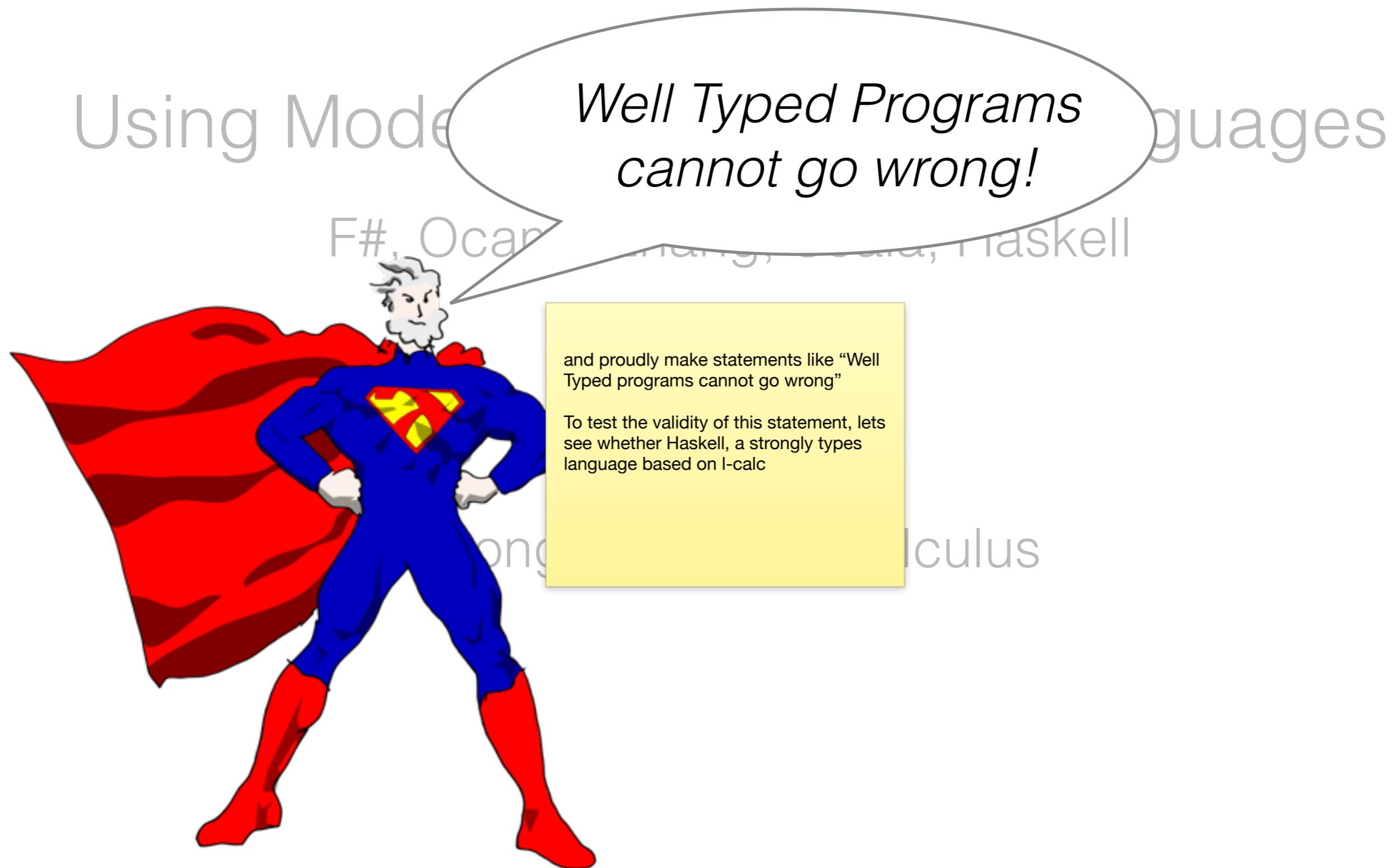
Because of  
Strong Types +  $\lambda$ -Calculus

The goal of research in programming languages is not only to track these bugs, but rather make such kind of software bugs difficult to express.

So, these days we use fancy FPL like that make heavy use

Towards this goal, PL researchers have built fancy functional programming languages such as ... that make heavy use of the well studied theories of strong typing and lambda calculus

# Make bugs difficult to express





can be used to encode Heartbleed





can be used to encode Heartbleed



I am importing data text, a library fir  
unicode mania  
I am creating a text niki  
and will use a take function to index  
pack .

Note that take comes from a library  
marked as unsafe, so safety relies only  
on type system and not on runtime  
checks.

```
λ> :m +Data.Text Data.Text.Unsafe
```

```
λ> let pack = "hat"
```

```
λ> :t takeWord16
```

```
takeWord16 :: Text -> Int -> Text
```



VS.



If I ask for the True first characters  
lambda man is smart enough to tell me  
that True is an invalid argument for take

```
λ> :m +Data.Text Data.Text.Unsafe
```

```
λ> let pack = "hat"
```

```
λ> takeWord16 pack True
```

Type Error: Cannot match **Bool** vs **Int**



VS.



```
λ> :m +Data.Text Data.Text.U  
λ> let pack = "hat"  
  
λ> takeWord16 pack 500  
“hat\58456\2594\SOH\NUL...
```

But, If I ask for the 8 first carachets I encoded HD take will return the 4 first carachets and 4 more characters that are stored in the memory next to Niki

My goal is to use types and statically catch this error



VS.



# Valid Values for takeWord16?

the type of takes states that all ints are valid indexes

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

**All Ints**

`..., -2, -1, 0, 1, 2, 3, ...`

# Valid Values for takeWord16?

what i want is to split ints and specify  
tats the valid arguments for take are  
only ints that are less than the size of t

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

**Valid Ints**

`0, 1 ... , len t`

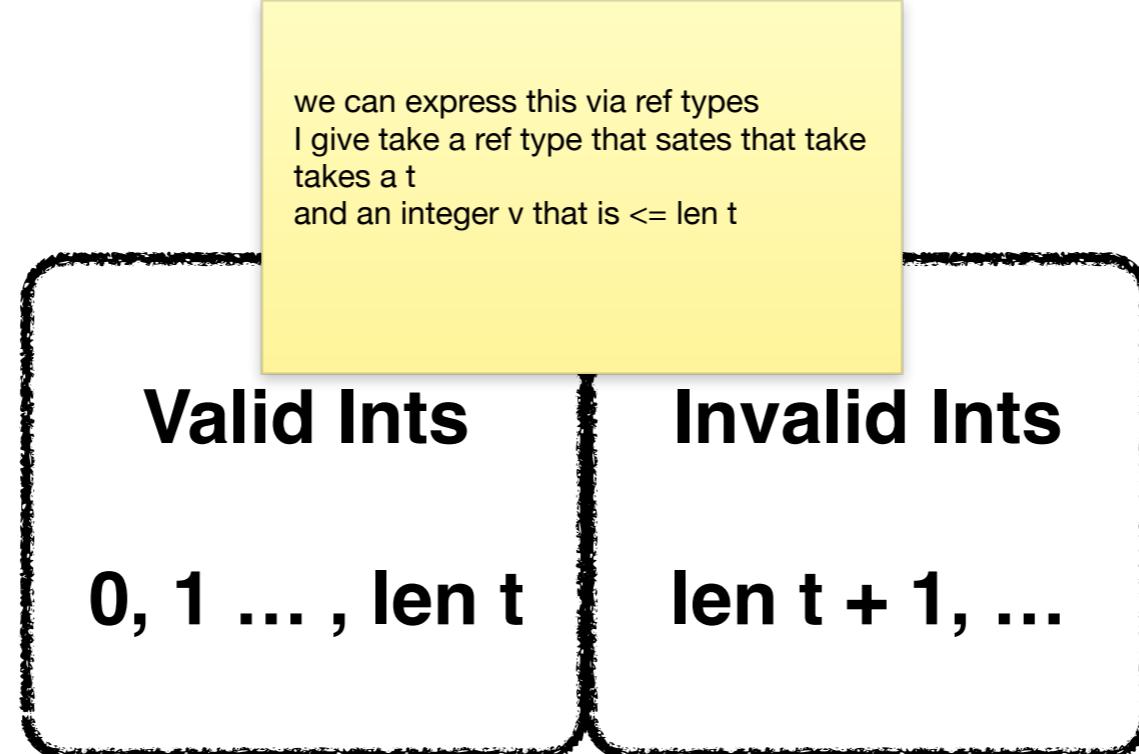
**Invalid Ints**

`len t + 1, ...`

# Refinement Types

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

we can express this via ref types  
I give take a ref type that states that take  
takes a t  
and an integer v that is  $\leq \text{len } t$



# Refinement Types

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

```
λ> :m +Data.Text Data.Text.
```

```
λ> let pack = "hat"
```

```
λ> take pack 500
```

```
Refinement Type Error
```

going back to the code, with the refined type for take, asking 8 elemns of Niki will return a refinement type error,

My goal is to allow this reasoning by embedding refinements into haskell's type system

## Goal: Refinement Types for Haskell

# **Goal: Refinement Types for Haskell**

specifically during my PhD I developed a sound and expressive refinement type system for Haskell programs and what is important is that I achieved both of these under the requirement of decidable and predictable verification

## **1. Soundness**

Under Lazy Evaluation

## **2. Expressiveness**

Modular Higher-Order Specifications

Refinement Reflection

**With automatic & predictable verification.**

Lets start by building a refinement type checkers that is sound unders

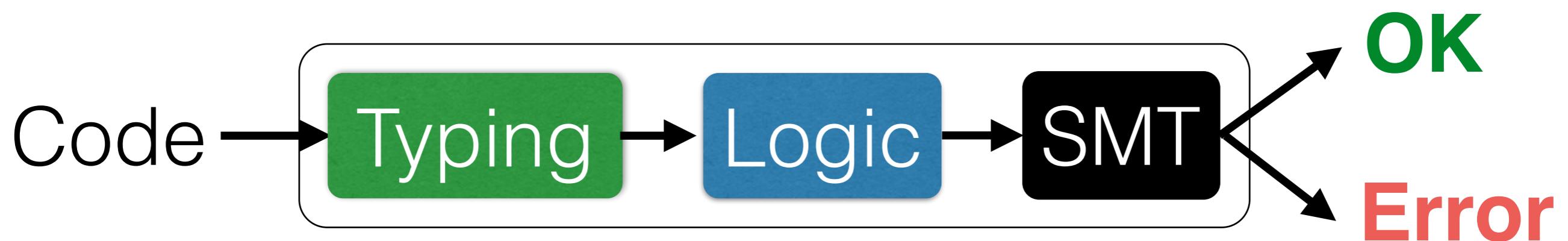
## 1. Soundness

Under Lazy Evaluation

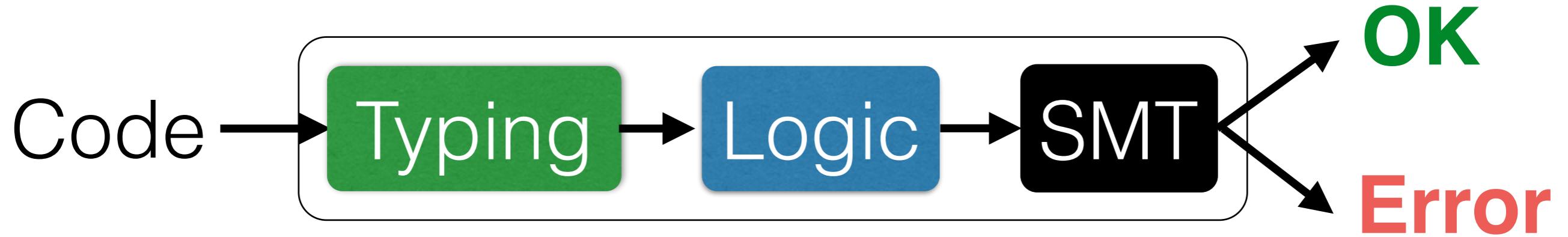
# Refinement Typing 101



# Refinement Typing 101



1. Source Code to **Type constraints**
2. **Type Constraints** to **Verification Condition (VC)**
3. Check **VC validity** with **SMT Solver**



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

Code → Typing

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

x:{v|len v=3} |- {v|v=500} <: {v|v<=len x}

Code → Typing

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

x: {v | len v = 3} |- {v | v = 500} <: {v | v <= len x}

Code → Typing

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

$x : \{v \mid \text{len } v = 3\} \dashv \{v \mid v = 500\} \lhd \{v \mid v \leq \text{len } x\}$

Code → Typing

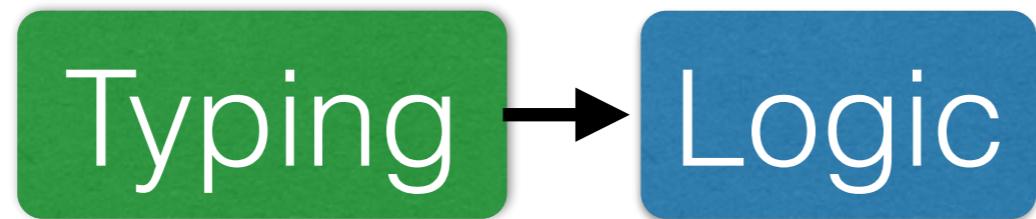
```
take :: t:Text -> {v | v <= len t} -> Text
heartbleed = let x = "hat"
            in take x 500
```

$x : \{v \mid \text{len } v = 3\} \dashv \{v \mid v = 500\} \lhd \{v \mid v \leq \text{len } x\}$

Code → Typing

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

$x : \{v \mid \text{len } v = 3\} \dashv \{v \mid v = 500\} \lhd \{v \mid v \leq \text{len } x\}$



## Encode Subtyping as Logical VC

If VC valid then Subtyping holds

Clearly state that on the left I have my environment



## Encode Subtyping as Logical VC

$$x : \{v \mid \text{len } v = 3\} \dashv \vdash \{v \mid v = 500\} \lhd \{v \mid v \leq \text{len } x\}$$

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

$\{v \mid p\} \lhd \{v \mid q\}$





to encode an environment binding we use flanagan;s meaning that says

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

**Means\***: if  $x$  reduces to a value then  $p[x/v]$

**Encoded as**: “ $x$  has a value”  $\Rightarrow p[x/v]$

\* Flanagan “Hybrid Type Checking” (POPL ’06)



$$\{\textcolor{blue}{v \mid p}\} \lhd \{\textcolor{blue}{v \mid q}\}$$

**Means:** if  $y : \{\textcolor{blue}{v \mid p}\}$  then  $y : \{\textcolor{blue}{v \mid q}\}$

**Encoded as:**  $p \Rightarrow q$

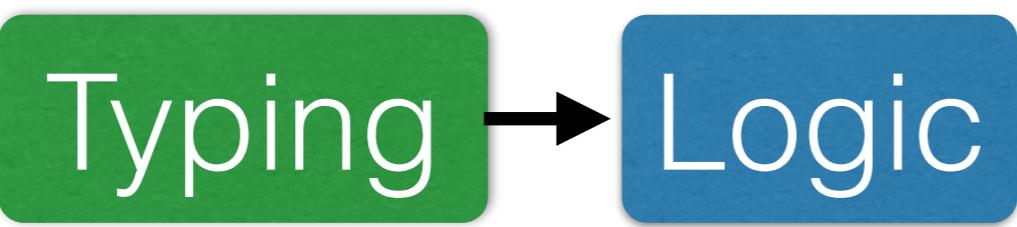


## Encode Subtyping ...

```
x:{v|len v=3} |- {v|v=500} <: {v|v<=len x}
```

... as Logical VC

```
(“x has a value” => len x = 3)  
=> (v = 500) => (v <= len x)
```



## Encode Subtyping ...

$x : \{v \mid \text{len } v = 3\}$   $\vdash$   $\{v \mid v = 500\} <: \{v \mid v \leq \text{len } x\}$

... as Logical VC

(“ $x$  has a value”  $\Rightarrow$   $\text{len } x = 3$ )  
 $\Rightarrow (v = 500) \Rightarrow (v \leq \text{len } x)$

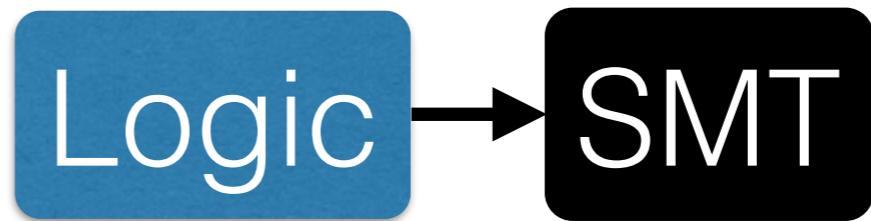


## Encode Subtyping ...

$x : \{v \mid \text{len } v = 3\} \dashv \vdash \{v \mid v = 500\} <: \{v \mid v \leq \text{len } x\}$

... as Logical VC

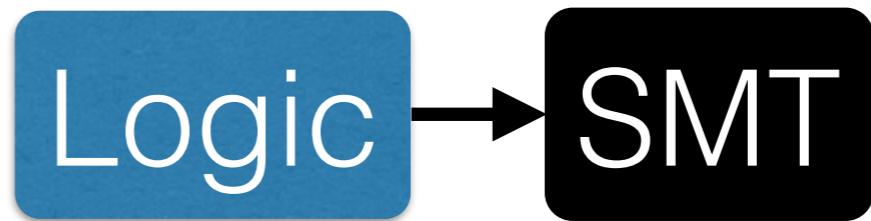
(“ $x$  has a value”  $\Rightarrow \text{len } x = 3$ )  
 $\Rightarrow (v = 500) \Rightarrow (v \leq \text{len } x)$



(“`x` has a value”  $\Rightarrow$  `len x = 3`)  
 $\Rightarrow (\text{v} = 500) \Rightarrow (\text{v} \leq \text{len x})$

**How to encode “`x` has a value” ?**

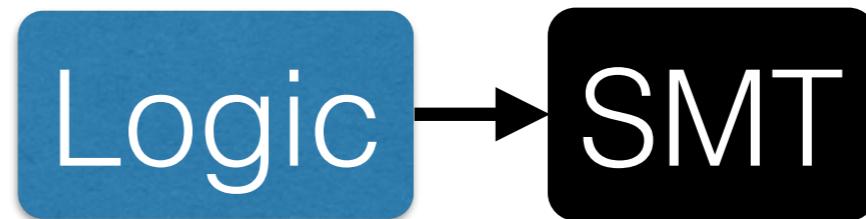
(In a decidable manner)



(“`x` has a value”  $\Rightarrow \text{len } x = 3$ )  
 $\Rightarrow (v = 500) \Rightarrow (v \leq \text{len } x)$

## **Assumption: Binders Are Values!**

i.e. `x` is guaranteed to be a value



len x = 3  
=> (v = 500) => (v <= len x)

## Assumption: Binders Are Values!

i.e. x is guaranteed to be a value



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

Checker soundly reports **Error**

# **Assumption: Binders must be values**

Not true under Haskell's lazy semantics

**Ignoring “has a value” is unsound!**

Under Haskell’s lazy semantics

# Ignoring “has a value” is unsound!

```
spin :: Int -> Int  
spin x = spin x
```

# Ignoring “has a value” is unsound!

```
spin :: Int -> {v:Int | false}  
spin x = spin x
```

OK

As `spin` does not return any value

# Ignoring “has a value” is unsound!

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

```
heartbleed = let x = "hat"
              y = spin 0
            in take x 500
```

OK? or Error?

# Ignoring “has a value” is unsound!

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

heartbleed = let x = "hat"
            y = spin 0
            in take x 500
```

**OK** under **CBV** evaluation

# Ignoring “has a value” is unsound!

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

heartbleed = let x = "hat"
            y = spin 0
            in take x 500
```

Error under Lazy evaluation

**Ignoring “has a value” is unsound!**

Reports **Erroneous** code as **OK**

**Ignoring “has a value” is unsound!**

**How to encode “has a value” ?**

# **How to encode “has a value” ?**

Most expressions provably reduce to a value

# How to encode “has a value” ?

Most expressions provably reduce to a value

**If**      x reduces to a value,

**Then** encode “x has a value” by **true**

# Solution: Stratified Types

$x : \{v : \text{Int} \mid p\}$

**Must** reduce to a Value

$x : \{v : \text{Int}^{\uparrow} \mid p\}$

**May-not** reduce to a Value

# Stratified Types to Logic

$x : \{v : \text{Int} \mid p\}$  encoded as  $p[x/v]$

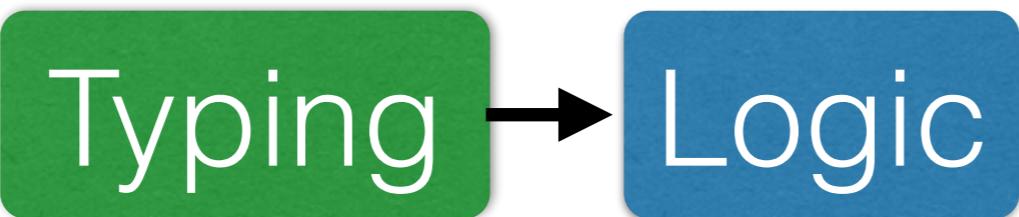
$x : \{v : \text{Int}^{\uparrow} \mid p\}$  encoded as  $\text{true}$

Code → Typing

```
take :: t:Text -> {v | v <= len t} -> Text
spin :: Int -> {v:Int↑ | false}
```

```
heartbleed = let x = "hat"
              y = spin 0
            in take x 500
```

x:{v | len v = 3}  
y:{v:Int<sup>↑</sup> | false} |- {v | v = 500} <: {v | v <= len x}



```
x:{v|len v=3}
y:{v:Int↑|false} |- {v|v=500} <: {v|v<=len x}
```

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



len x=3

true

$\Rightarrow v = 500 \Rightarrow v \leq \text{len } x$



```
take :: t:Text -> {v | v <= len t} -> Text
spin :: Int -> {v:Int↑ | false}

heartbleed = let x = pack "hat"
            y = spin 0
            in take x 500
```

## How to enforce stratification?

# How to enforce stratification?

$x : \{v : \text{Int} \mid p\}$

**Must** have a Value

Terminating expressions must have a value

**Solution**

Check termination with Refinement Types!

We saw how regain soundness under  
lazy evaluation using type stratification

# 1. Soundness Under Lazy Evaluation

# Goal: Refinement Types for ~~Tool~~~1~11

What is important is that this technique allows for decidable and predictable verification

## 1. Soundness

Under Lazy Evaluation

## 2. Expressiveness

Modular Higher-Order Specifications

Refinement Reflection

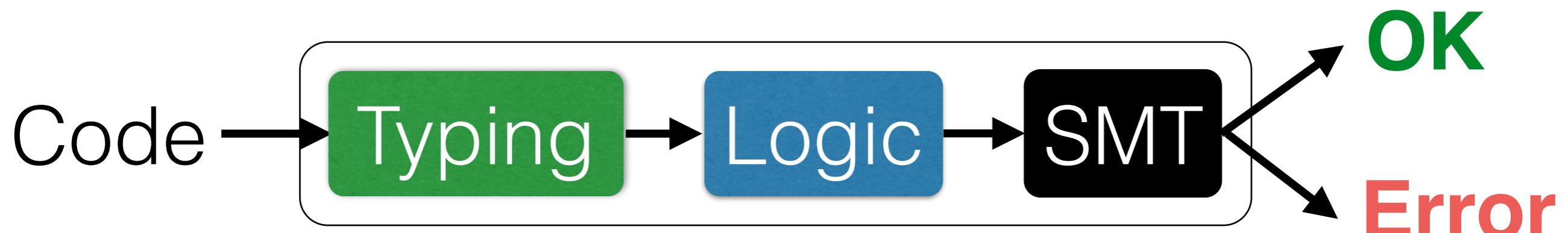
**With automatic & predictable verification.**

What is important is that this technique allows for decidable and predictable verification

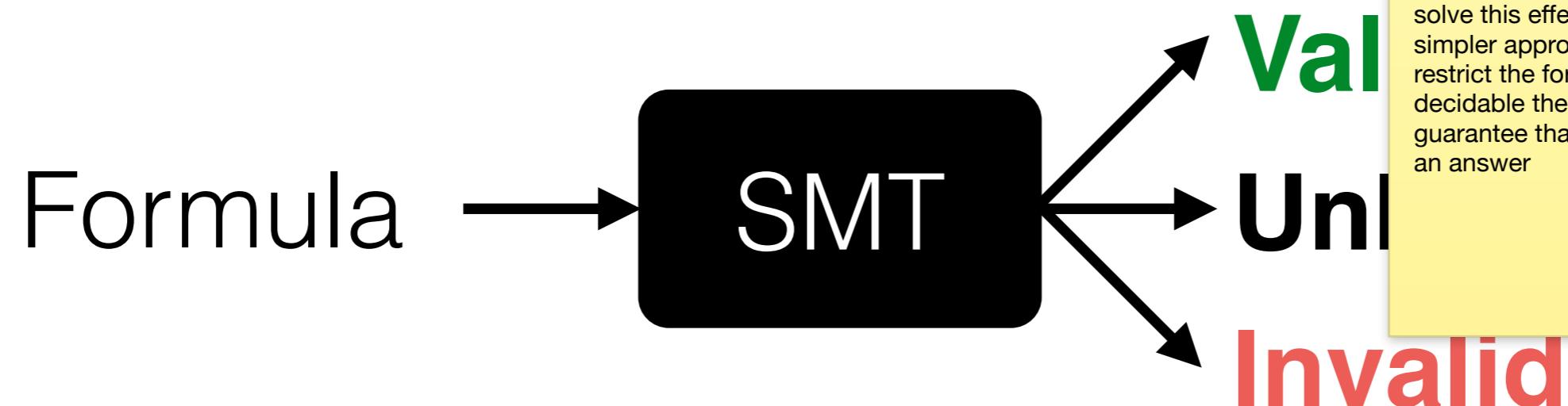
# With automatic & predictable verification.

# With automatic & predictable error detection.

Recall the structure of the type checker  
that extracts the VC from types and  
checks the verification of VC using an  
SMT solver



# With automatic & predictable verification.



An SMT solver is a black box that given a formula it can

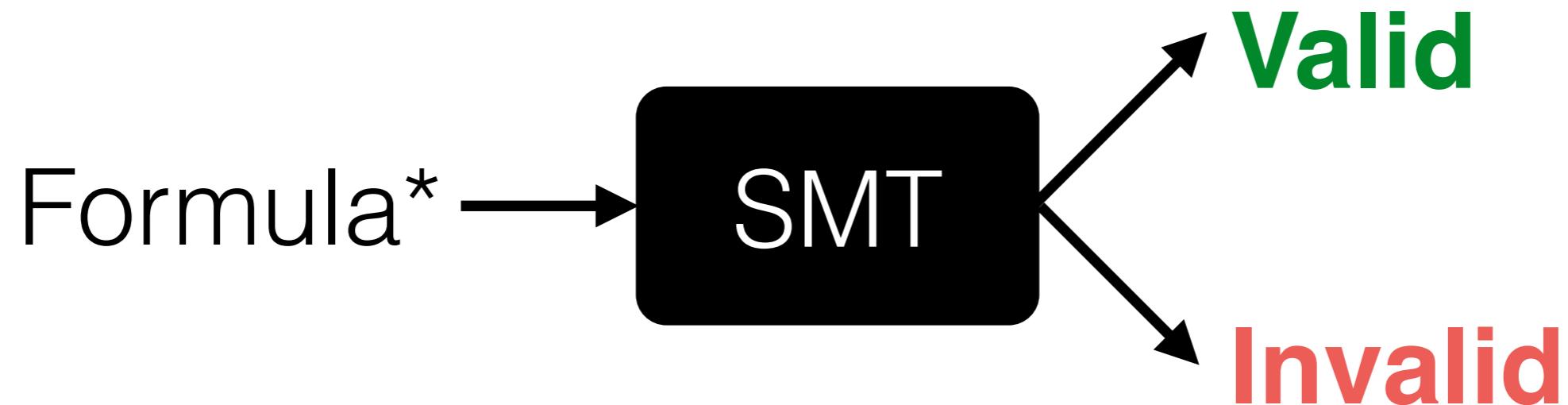
1. return valid if the formula is always true,
2. return unsat if the formula is always false, or
3. timeout with unknown

Timeouts lead to unpredictable behaviours. what Rustan calls the butterfly event: minor modification at one part of your programs can change the verification of another part

Rustan carefully selects triggers to solve this effect, while we chose a simpler approach that says that if we restrict the formula to efficient and decidable theories, then we have the guarantee that the SMT will return with an answer

# With automatic & predictable verification.

How do we prove it is correct?



\*From Decidable Logic



The VC comes via conjunctions and implications from the refinements of types, thus

## Encode Subtyping ...

$$x_1 : \{v \mid p_1\}, \dots, x_n : \{v \mid p_n\} \vdash \{v \mid q_1\} <: \{v \mid q_2\}$$

... as Logical Verification Condition

$$p_1 \wedge \dots \wedge p_n \Rightarrow q_1 \Rightarrow q_{12}$$

**For predictability, p from decidable theories**

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

**... as Logical Verification Condition**

$$p_1 \wedge \dots \wedge p_n \Rightarrow q_1 \Rightarrow q_{12}$$

where dec theories are theories for which the SMT has dec & efficient procedures, like ...

# For predictability, p from decidable theories

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

Boolean Logic

(QF) Linear Arithmetic

Uninterpreted Functions ...

# For automatic & predictable verification.

where dec theories are theories for which the SMT has dec & efficient procedures, like ...

# For predictability, p from decidable theories

{v:a | p }

Boolean Logic

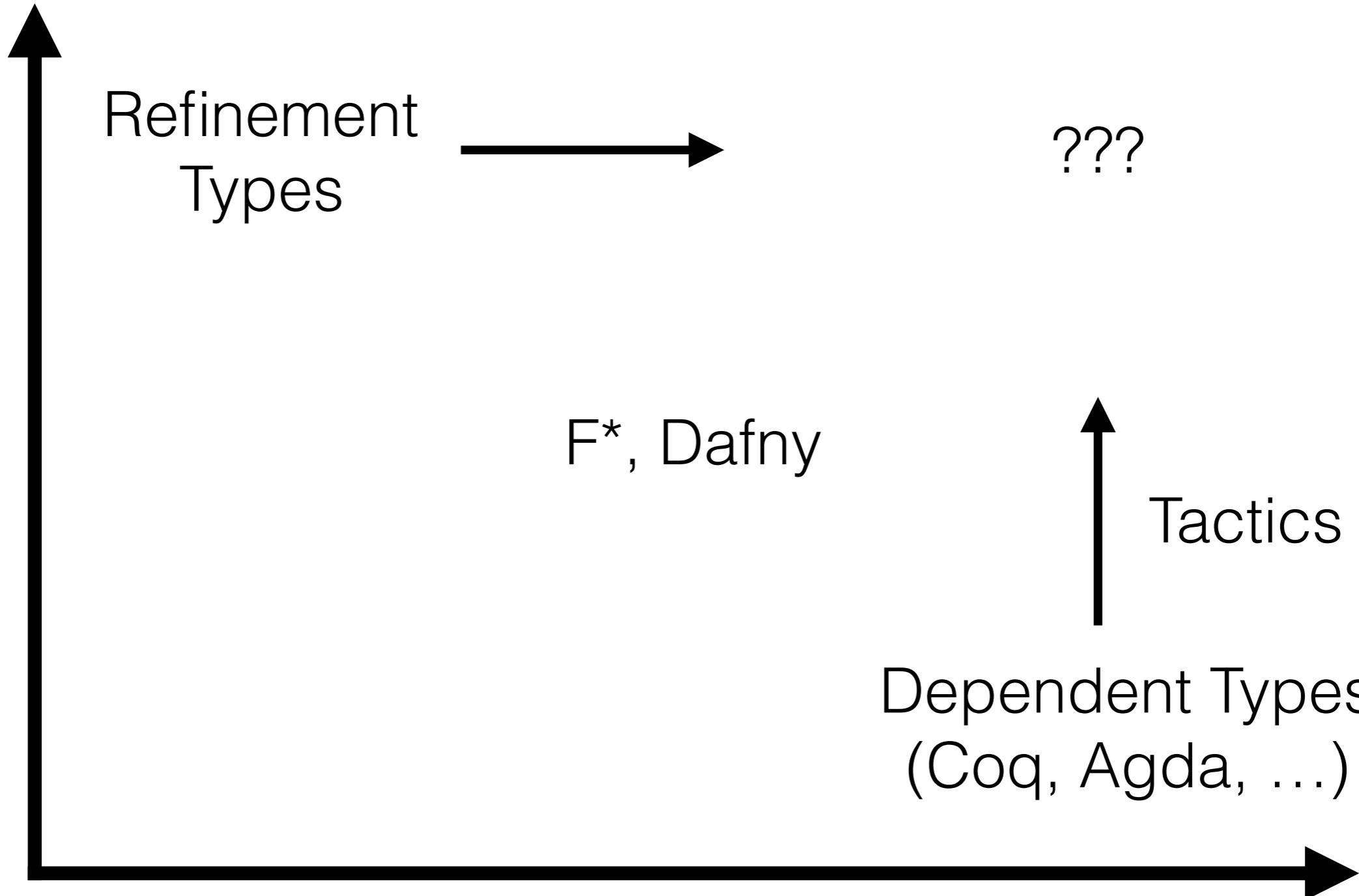
(QF) Linear Arithmetic

Uninterpreted Functions ...

## For automatic & predictable verification.

## What about expressiveness?

**Automation**



**Expressiveness**

# **Goal: Refinement Types for Haskell**

## **1. Soundness**

Under Lazy Evaluation

## **2. Expressiveness**

Modular Higher-Order Specifications

Refinement Reflection

With automatic & predictable verification.

# Goal: Modular & Decidable Specification

As a first example, consider a function maximum that takes as input a list of ints and returns the maximum element of the list

```
maximum      :: [Int] -> Int
maximum [x]    = x
maximum (x:xs) = let y = maximum xs in
                  if x < y then y else x
```

# Client Verification

```
p = maximum [1, 2, 3]
assert (p > 0)

n = maximum [-1, -2, -3]
assert (n < 0)
```

# Goal: Modular & Decidable Specifications

maximum [x] = x

maximum (x:xs) = let y = maximum xs in  
if x < y then y else x

Specification?

# Goal: Modular & Decidable Specifications

maximum  
maximum

[x]  
(x:xs)

= x  
Let y = maximum xs in  
if x < y then y else x

Modular  
First-Order Specification

maximum : [ {v:Int | 0 < v} ] -> {v:Int | 0 < v}

# Goal: Modular & Decidable Specifications

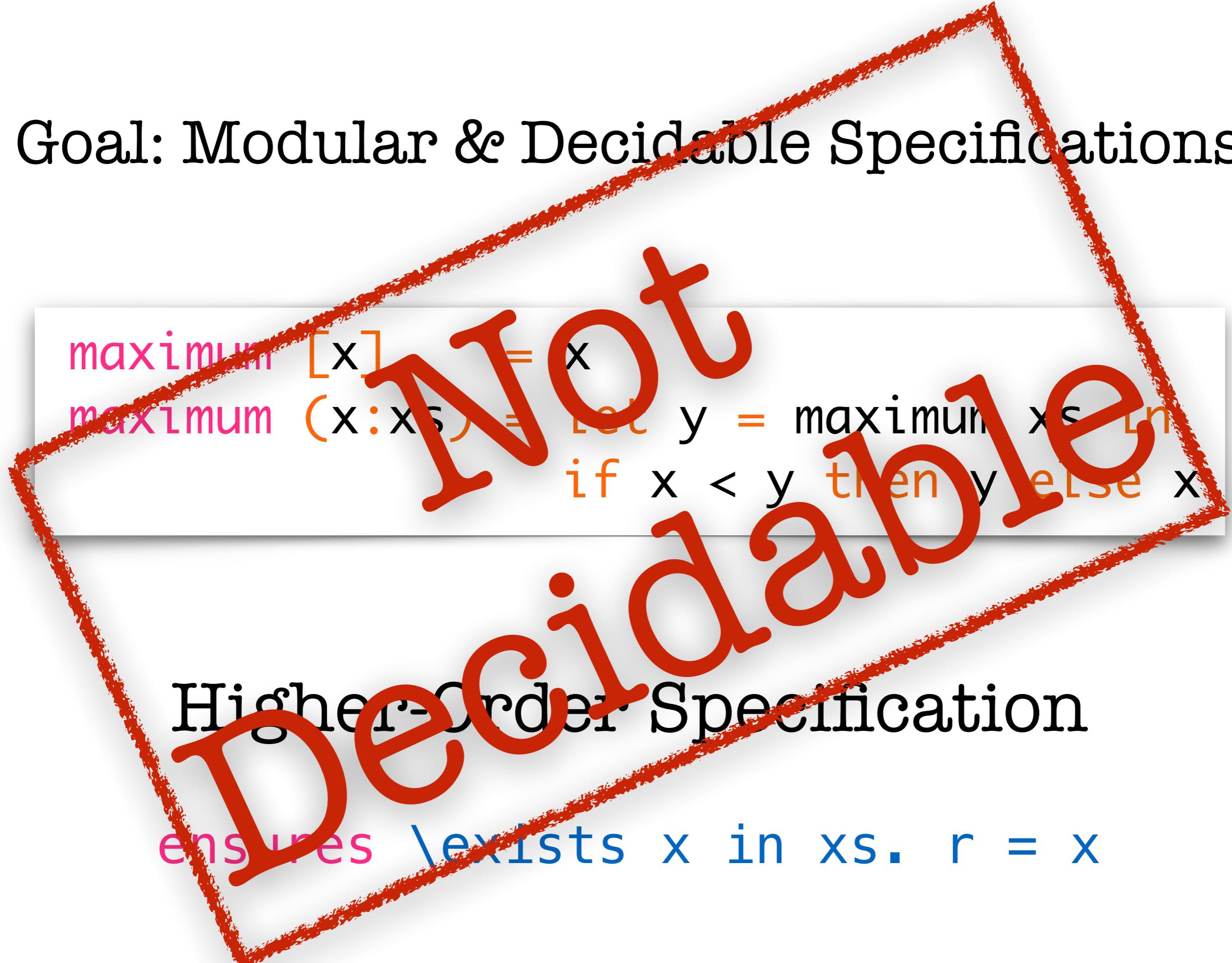
maximum  
maximum

[x]  
(x : xs)

=  
Let y = maximum xs in  
if x < y then y else x

Higher-Order Specification  
ensures \exists x in xs. r = x

Decidable



**Observation:**

maximum returns one of the input elements

## **Observation:**

**maximum** returns one of the input elements

```
maximum :: [{v:Int | 0 < v}] -> {v:Int | 0 < v}
```

**if** all input elements are **positive**

**then** the result is **positive**

## Observation:

maximum returns one of the input elements

```
maximum :: [{v:Int | v<0}] -> {v:Int | v<0}
```

**if** all input elements are negative  
**then** the result is negative

## Observation:

`maximum` returns one of the input elements

```
maximum :: [{v:Int | p v}] -> {v:Int | p v}
```

**if** all input elements satisfy `p`

**then** the result satisfies `p`

## Observation:

maximum returns one of the input elements

maximum::  $\forall p. [\{v: \text{Int} \mid p v\}] \rightarrow \{v: \text{Int} \mid p v\}$

**for all**  $p$

**if** all input elements satisfy  $p$

**then** the result satisfies  $p$

## Observation:

maximum returns one of the input elements

maximum::  $\forall p. [\{v:\text{Int} \mid p v\}] \rightarrow \{v:\text{Int} \mid p v\}$

In SMT  $p$  is “Uninterpreted Function”

$$\forall \bar{x}, \bar{y}. \bar{x} = \bar{y} \Rightarrow (p \bar{x}) = (p \bar{y})$$

## Observation:

maximum returns one of the input elements

maximum::  $\forall p. [\{v: \text{Int} \mid p v\}] \rightarrow \{v: \text{Int} \mid p v\}$

Is Verification Decidable?

# Is Verification Decidable?

```
maximum ::= ∀p. [{v:Int | p v}] -> {v:Int | p v}
maximum [x]      = x
maximum (x:xs) = let y = maximum xs in
                  if x < y then y else x
```

# Is Verification Decidable?

```
maximum ::= ∀p. [{v:Int | p v}] -> {v:Int | p v}
maximum [x]      = x
maximum (x:xs) = let y = maximum xs in
                  if x < y then y else x
```

Assume

Prove

# Is Verification Decidable?

maximum ::=  $\forall p. [\{v: \text{Int} \mid p v\}] \rightarrow \{v: \text{Int} \mid p v\}$

maximum  $[x]$  = x

maximum  $(x: xs)$  = let y = maximum xs in  
if  $x < y$  then y else x

Assume

Prove

$x: \{x: \text{Int} \mid p x\}$

$xs: [\{v: \text{Int} \mid p v\}]$

# Is Verification Decidable?

```
maximum ::= ∀p. [{v:Int | p v}] -> {v:Int | p v}
maximum [x]      = x
maximum (x:xs) = let y = maximum xs in
                  if x < y then y else x
```

Assume

$x : \{x : \text{Int} \mid p x\}$   
 $xs : [\{v : \text{Int} \mid p v\}]$   
 $y : \{y : \text{Int} \mid p y\}$

Prove

# Is Verification Decidable?

```
maximum ::= ∀p. [{v:Int | p v}] -> {v:Int | p v}
maximum [x]      = x
maximum (x:xs) = let y = maximum xs in
                  if x < y then y else x
```

Assume

$x: \{x: \text{Int} \mid p x\}$

$xs: [\{v: \text{Int} \mid p v\}]$

$y: \{y: \text{Int} \mid p y\}$

Prove

$y: \{x: \text{Int} \mid p y\}$

# Is Verification Decidable?

```
maximum ::= ∀p. [{v:Int | p v}] -> {v:Int | p v}
maximum [x]      = x
maximum (x:xs) = let y = maximum xs in
                  if x < y then y else x
```

Assume

$x: \{x: \text{Int} \mid p x\}$   
 $xs: [\{v: \text{Int} \mid p v\}]$   
 $y: \{y: \text{Int} \mid p y\}$

Prove

$y: \{x: \text{Int} \mid p y\}$   
 $x: \{x: \text{Int} \mid p x\}$

# Is Verification Decidable?

```
maximum ::= ∀p. [{v:Int | p v}] -> {v:Int | p v}
maximum [x]      = x
maximum (x:xs) = let y = maximum xs in
                  if x < y then y else xOK
```

Are SMT Valid ? Yes!

$$\begin{aligned} p x \wedge p y &\Rightarrow p y \\ p x \wedge p y &\Rightarrow p x \end{aligned}$$

# Is Verification Decidable?

```
maximum ::= ∀p. [{v:Int | p v}] -> {v:Int | p v}
maximum [x]      = x
maximum (x:xs) = let y = maximum xs in
                  if x < y then y else x
```

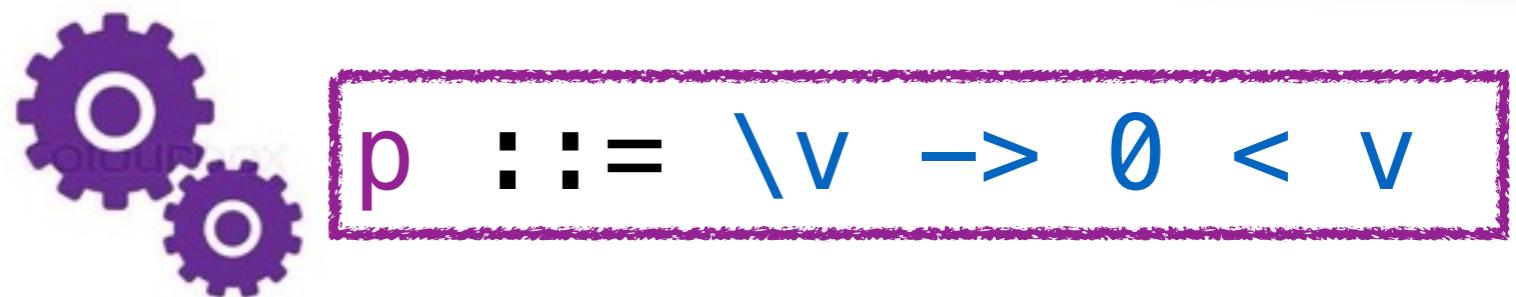
Verification is **Decidable** ...

... but is Specification **Modular**?

add the “negative example”

# Client Verification

**maximum**::  $\forall p. [\{v:\text{Int} \mid p v\}] \rightarrow \{v:\text{Int} \mid p v\}$



**maxPos**::  $[\{v:\text{Int} \mid 0 < v\}] \rightarrow \{v:\text{Int} \mid 0 < v\}$

**maxPos** xs = maximum xs

OK

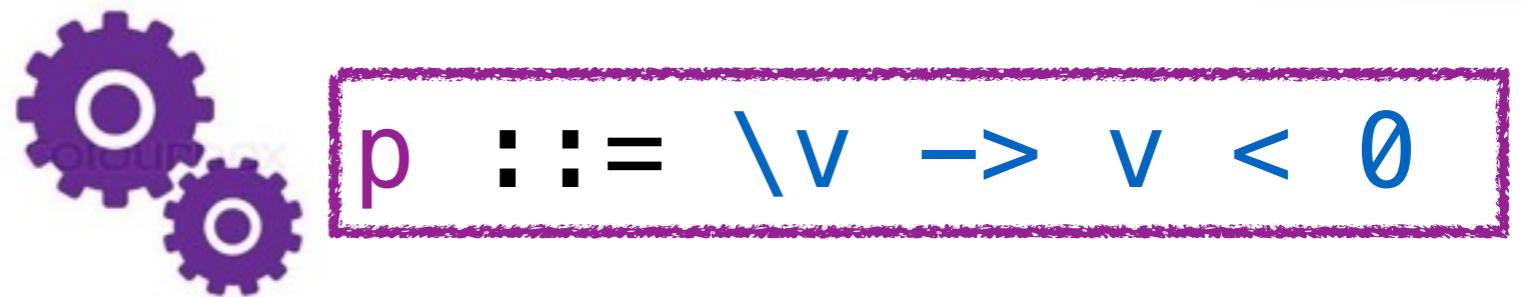
## Automatic Instantiation via Liquid Types\*

\* Rondon, Kawaguchi, Jhala. “Liquid Types” (PLDI ’08)

add the “negative example”

# Client Verification

**maximum**::  $\forall p. [\{v: \text{Int} \mid p v\}] \rightarrow \{v: \text{Int} \mid p v\}$



**maxNeg**::  $[\{v: \text{Int} \mid v < 0\}] \rightarrow \{v: \text{Int} \mid v < 0\}$

**maxNeg** xs = maximum xs

OK

# Modular & Decidable Specifications

**Step 1:** Abstract Specification Refinements @Lib

Higher Order VS Modular

**Step 2:** Installation Requirements @Clt

# **Applications**

## **Higher-Order Functions**

# **Higher-Order Functions**

## **Examples**

compose, foldr, filter, ...

# Higher-Order Functions

## Examples

compose, **foldr**, filter, ...

`foldr f b []` = b

`foldr f b (x:xs)` = f x (foldr f b xs)

# Specification

`foldr f b [] = b`

`foldr f b (x:xs) = f xs x (foldr f b xs)`

`inv xs b` := list `xs` satisfies `inv` at value `b`

**Goal:**

Specify, by induction, that the result satisfies `inv`

Specify, by induction, that the result satisfies `inv`

## Inductive Hypothesis:

```
f: (xs: [a] -> x:a -> {b:b | inv xs b}
      -> {r:b | inv (x:xs) r})
```

Specify, by induction, that the result satisfies `inv`

### **Inductive Hypothesis:**

$$\begin{aligned} f: (\textcolor{blue}{xs}: [\text{a}]) \rightarrow \textcolor{blue}{x}: \text{a} \rightarrow & \{ \textcolor{blue}{b}: \text{b} \mid \text{inv } \textcolor{purple}{xs} \text{ } \textcolor{blue}{b} \} \\ \rightarrow & \{ \textcolor{blue}{r}: \text{b} \mid \text{inv } (\textcolor{blue}{x}: \textcolor{blue}{xs}) \text{ } \textcolor{blue}{r} \} \end{aligned}$$

### **Base Case:**

$$b: \{ \textcolor{blue}{b} \mid \text{inv } [] \text{ } \textcolor{blue}{b} \}$$

Specify, by induction, that the result satisfies `inv`

### **Inductive Hypothesis:**

$$\begin{aligned} f: (\text{xs}: [\text{a}]) \rightarrow & \text{x: a} \rightarrow \{\text{b: b} \mid \text{inv xs b}\} \\ \rightarrow & \{\text{r: b} \mid \text{inv (x: xs) r}\} \end{aligned}$$

### **Base Case:**

$$b: \{\text{b} \mid \text{inv [] b}\}$$

### **Conclusion:**

$$\{\text{v: b} \mid \text{inv xs v}\}$$

Specify, by induction that the result satisfies `inv`

$$\begin{aligned}\text{foldr } f \ b \ [] &= b \\ \text{foldr } f \ b \ (x:xs) &= f \ xs \ x \ (\text{foldr } f \ b \ xs)\end{aligned}$$

```
foldr :: forall inv.  
         -> f:(xs:[a] -> x:a -> {b:b|inv xs b})  
              -> {r:b|inv (x:xs) r})  
         -> b:{b|inv [] b}  
         -> xs:[a]  
         -> {v:b|inv xs v}
```

**Hack:** `xs` as extra argument on `f`

Specify, by induction that the result satisfies `inv`

$$\begin{aligned}\text{foldr } f \ b \ [] &= b \\ \text{foldr } f \ b \ (x:xs) &= f \ x \ (\text{foldr } f \ b \ xs)\end{aligned}$$

$$\begin{aligned}\text{foldr} &:: \text{forall } \text{inv}. \\ \rightarrow &f: (xs: [a] \rightarrow x:a \rightarrow \{b:b \mid \text{inv } xs \ b\}) \\ &\quad \rightarrow \{r:b \mid \text{inv } (x:xs) \ r\}) \\ \rightarrow &b: \{b \mid \text{inv } [] \ b\} \\ \rightarrow &xs: [a] \\ \rightarrow &\{v:b \mid \text{inv } xs \ v\}\end{aligned}$$

**Problem:** type mismatch

**Solution:** helper abstract refinement

# Solution: helper abstract refinement

```
f: (xs:[a] -> x:a -> {b:b|inv xs b})  
    -> {r:b|inv (x:xs) r})
```

Simplifies to

```
f: (x:a -> b:b -> {r:b|stp x b r})
```

where

forall x xs b r.

$$\begin{aligned} \text{inv } xs \ b &\Rightarrow \text{stp } x \ b \ r \\ &\Rightarrow \text{inv } (x:xs) \ r \end{aligned}$$

# Solution: helper abstract refinement

```
f: (xs:[a] -> x:a -> {b:b|inv xs b})  
    -> {r:b|inv (x:xs) r})
```

Simplifies to

```
f: (x:a -> b:b -> {r:b|stp x b r})
```

where

```
bound Ind inv stp = \x xs b r ->  
    inv xs b => stp x b r  
    => inv (x:xs) r
```

# Specification

```
foldr :: (Ind inv stp)
        => f:(x:a -> b:b -> {r:b|stp x b r})
        -> {b:b|inv [] b}
        -> xs:[a]
        -> {v:b|inv xs v}
```

```
bound Ind inv stp = \x xs b r ->
    inv xs b -> stp x b r
    -> inv (x:xs) r
```

# Higher-Order Functions

## Examples

compose, **foldr**, filter, ...

# **Applications**

**Higher-Order Functions**

**Recursive Invariants**

**Floyd-Hoare Logic (in ST)**

**Capability-Safe IO Monad**

## **2. Expressiveness**

Modular Higher-Order Specifications

## **2. Expressiveness**

Refinement Reflection

# Specifications of functions

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

# Specifications of functions

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

# Specifications of functions

```
fib :: {i:Int | 0<=i} -> {v:Int | i<=v}
fib i
| i <= 1      = i
| 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 express **theorems** about functions?

```
fibCon :: i:Nat -> j:Nat  
        -> {v:() | i = j => fib i = fib j}  
fibCon _ _ = ()                                OK
```

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

# How to express **theorems** about functions?

```
fibCon :: i:Nat -> j:Nat  
        -> {i = j => fib i = fib j}  
fibCon _ _ = ()                                OK
```

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

# How to express **theorems** about functions?

```
fibOne :: {fib 1 = 1}  
fibOne = ()
```

Error

How to connect logic fib with target fib?

# How to connect logic fib with target fib?

~~$\text{fib} :: i:\text{Nat} \rightarrow \text{Nat}$~~

~~$\text{fib } i$~~

~~$| \quad 1 \leq 1$~~

~~$| \quad \text{otherwise} = \text{fib } (i-1) + \text{fib } (i-2)$~~

**Not decidable**

~~$\forall i.$~~

~~$\text{if } i \leq 1 \text{ then } \text{fib } i = i$~~

~~$\text{else } \text{fib } i = \text{fib } (i-1) + \text{fib } (i-2)$~~

**SMT Axiom**

# How to connect logic fib with target fib?

```
fib :: i:Nat -> Nat
fib i
| i <= 1      = i
| otherwise    = fib (i-1) + fib (i-2)
```

## Refinement Reflection

```
fib :: i:Nat -> {v:Nat | v = fib i &&
if i <= 1 then fib i = i
else fib i = fib (i-1) + fib (i-2)
}
```

# Refinement Reflection

**Step 1:** Definition

**Step 2:** Reflection

```
fib :: i:Nat -> {v:Nat | v = fib i &&
if i <= 1 then fib i = i
else fib i = fib (i-1) + fib (i-2)
}
```

# Refinement Reflection

**Step 1:** Definition

**Step 2:** Reflection

```
fib :: i:Nat -> {v:Nat | v = fib i &&
if i <= 1 then fib i = i
else fib i = fib (i-1) + fib (i-2)
}
```

**Step 3:** Application

```
fib 1 :: {v:Nat | v = fib 1 && fib 1 = 1}
```

# Refinement Reflection

```
fibOne :: {fib 1 = 1}  
fibOne = let _ = fib 1  
        in ()
```

OK

```
fib 1 :: {v:Nat | v = fib 1 && fib 1 = 1}
```

# Refinement Reflection

```
fibTwo :: {fib 2 = 2}
fibTwo = let _ = fib 2
         _ = fib 1
         _ = fib 0
       in ()
```

OK

**Problem:** Proof is Unreadable!

**Solution:** Proof Combinators!

# Refinement Reflection

```
fibTwo :: {fib 2 = 2}
fibTwo
  =  fib 2
  ==. fib 1 + fib 0
*** QED
```

OK

**Problem:** Proof is Unreadable!

**Solution:** Proof Combinators!

# Refinement Reflection

```
fibTwo :: {fib 2 = 2}
```

```
fibTwo
```

```
= fib 2
```

```
==. fib 1 + fib 0
```

```
==. 1 + 1
```

```
==. 2
```

```
*** QED
```

OK

Can we reuse existing proofs?

# Refinement Reflection

```
fibThree :: {fib 3 = 3}
fibThree
= fib 3
==. fib 2 + fib 1
==. 2 + 1           ? fibTwo      OK
==. 3
*** QED
```

Combining Proofs

# Refinement Reflection

Prove **theorems** about functions.

**Step 1:** Definition

**Step 2:** Reflection

**Step 3:** Application

# Refinement Reflection

Turns Haskell into a theorem prover.

# Refinement Reflection

Turns Haskell into a theorem prover.

Express **theorems** via refinement types.

Express **proofs** via functions.

**Check** that functions prove theorems.

# Refinement Reflection

```
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-1) + fib (i-2)
<=. fib i      + fib (i-2) ? fibUp (i-1)
<=. fib i      + fib (i-1) ? fibUp (i-2)
<=. fib (i+1)
*** QED
```

# Refinement Reflection

## Higher Order Theorems

```
fMono :: f:(Nat -> Int)
        -> fUp:(z:Nat -> {f z <= f (z+1)})
        -> x:Nat
        -> y:{Nat | x < y}
        -> {f x <= f y}
```

```
fibMono :: x:Nat -> y:{Nat | x < y}
          -> {fib x <= fib y}
fibMono = fMono fib fibUp
```

# **Applications**

## **Arithmetic Properties**

**(fib, ackermann)**

# **Applications**

**Arithmetic Properties**

(fib, ackermann)

**Class Laws**

(monoid, functor, monad)

**Parallelization Equivalence**

(of monoid morphisms)

# **Parallelization Equivalence**

MapReduce: Apply  $f \ x$  in parallel

# Parallelization Equivalence

MapReduce: Apply  $f \ x$  in parallel

1. Chunk input  $x$  in  $i$  chunks (`chunk i x`)
2. Apply  $f$  to each chunk in parallel (`map f`)
3. Reduce all results op (`foldr op (f [])`)

# Parallelization Equivalence

MapReduce: Apply  $f \ x$  in parallel

1. Chunk input  $x$  in  $i$  chunks ( $\text{chunk } i \ x$ )
2. Apply  $f$  to each chunk in parallel ( $\text{map } f$ )
3. Reduce all results op ( $\text{foldr } op \ (f \ [])$ )

```
mapReduce i f op xs  
= foldr op (f []) (map f (chunk i xs))
```

# Example: Parallel Sums

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

```
psum :: Int -> xs:[Int] -> Int
psum i xs = mapReduce i sum (+) xs
```

Question: is parallel sum equivalent to sum?

# Question: is parallel sum equivalent to sum?

```
sumEq :: i:Int->xs:[Int]->\{sum xs = psum i xs}
sumEq i xs
  =
  psum i xs
  ==.
  sum xs
    ? mapReduceEq i sum (+) rightId distr xs
*** QED
```

# Question: is parallel sum equivalent to sum?

```
sumEq :: i:Int->xs:[Int] -> {sum xs = psum i xs}
sumEq i xs
  =
  psum i xs
==.
  sum xs
    ? mapReduceEq i sum (+) rightId distr xs
*** QED
```

```
rightId :: xs:[Int] -> ys:[Int]
          -> {sum xs + sum [] = sum xs}
```

# Question: is parallel sum equivalent to sum?

```
sumEq :: i:Int->xs:[Int]->\{sum xs = psum i xs}
sumEq i xs
  =
  psum i xs
  ==.
  sum xs
    ?
    mapReduceEq i sum (+) rightId distr xs
*** QED
```

```
rightId :: xs:[Int] -> ys:[Int]
          -> \{sum xs + sum [] = sum xs}
```

```
distr   :: xs:[Int] -> ys:[Int]
          -> \{sum (xs++ys) = sum xs + sum ys}
```

# Parallelization Equivalence

```
mapReduceEq
  :: i:Int
  -> f:([a] -> b)
  -> op:(b -> b -> b)
  -> rId:(xs:[a] -> {f xs `op` f [] = f xs})
  -> dis:(xs:[a] -> ys:[a]
           -> {f (xs++ys) = f xs `op` f ys})
  -> xs:[a]
  -> { f x = mapReduce i f op xs }
```

# Parallelization Equivalence

mapReduceEq

```
:: i:Int  
-> f:( [a] -> b)  
-> op:(b -> b -> b)  
-> rId:(xs:[a] -> {f xs `op` f [] = f xs})  
-> dis:(xs:[a] -> ys:[a]  
       -> {f (xs++ys) = f xs `op` f ys})  
-> xs:[a]  
-> { f x = mapReduce i f op xs }
```

Generalization:

Let  $b$  be a monoid with an assoc operator

# Parallelization Equivalence

```
mapReduceEq :: (Monoid b)
  => n:Int
  -> f:([a] -> b)
  -> rId:(xs:[a] -> {f xs `op` f [] = f xs})
  -> dis:(xs:[a] -> ys:[a]
           -> {f (xs++ys) = f xs `op` f ys})
  -> xs:[a]
  -> { f x = mapReduce n f xs }
```

Generalization:

Let b be a **verified** monoid with right identity

# Parallelization Equivalence

```
mapReduceEq :: (VerifiedMonoid b)
=> n:Int
-> f:([a] -> b)
-> dis:(xs:[a] -> ys:[a]
      -> {f (xs++ys) = f xs `op` f ys})
-> xs:[a]
-> { f x = mapReduce n f xs }
```

Generalization:

Let  $b$  be a **verified** monoid with right identity

# Parallelization Equivalence

```
mapReduceEq :: (VerifiedMonoid b)
  => n:Int
  -> f:([a] -> b)
  -> dis:(xs:[a] -> ys:[a]
    -> {f(xs++ys) = f xs `op` f ys})
  -> xs:[a]
  -> { f x = mapReduce n f xs }
```

Generalization:

Let  $f$  be a morphism among  $a$  and  $[b]$

# Parallelization Equivalence

```
mapReduceEq :: (VerifiedMonoid b)
=> n:Int
-> f:Morphism [a] b
-> xs:[a]
-> { f x = mapReduce n f xs }
```

# **Applications**

**Arithmetic Properties**

(fib, ackermann)

**Class Laws**

(monoid, functor, monad)

**Parallelization Equivalence**

(of monoid morphisms)

**Parallelization of String Matching**

# **2. Expressiveness**

## Refinement Reflection

# **Goal: Refinement Types for Haskell**

## **1. Soundness**

Under Lazy Evaluation

## **2. Expressiveness**

Modular Higher-Order Specifications

Refinement Reflection

**With decidable & predictable verification.**

# Implementation

LiquidHaskell

# LiquidHaskell

**Liquid Types \***

Decidable Type Inference

\* Rondon, Kawaguchi, Jhala. “Liquid Types” (PLDI ’08)

# LiquidHaskell

**Liquid Types**

+

**Termination Checker [ICFP 14]**

Soundness under Laziness

# LiquidHaskell

**Liquid Types**

+

**Termination Checker [ICFP 14]**

+

**Abstract Refinement Types [ESOP 13]**

Desugar to Ghost Variables

# LiquidHaskell

**Liquid Types**

+

**Termination Checker [ICFP 14]**

+

**Abstract Refinement Types [ESOP 13]**

+

**Bounded Refinement Types [ICFP 15]**

Desugar to Ghost Functions

# LiquidHaskell

**Liquid Types**

+

**Termination Checker [ICFP 14]**

+

**Abstract Refinement Types [ESOP 13]**

+

**Bounded Refinement Types [ICFP 15]**

+

**Refinement Reflection [Submitted 16]**

Liquid Haskell is a theorem prover

# LiquidHaskell

**Termination, Totality, Functional Correctness**

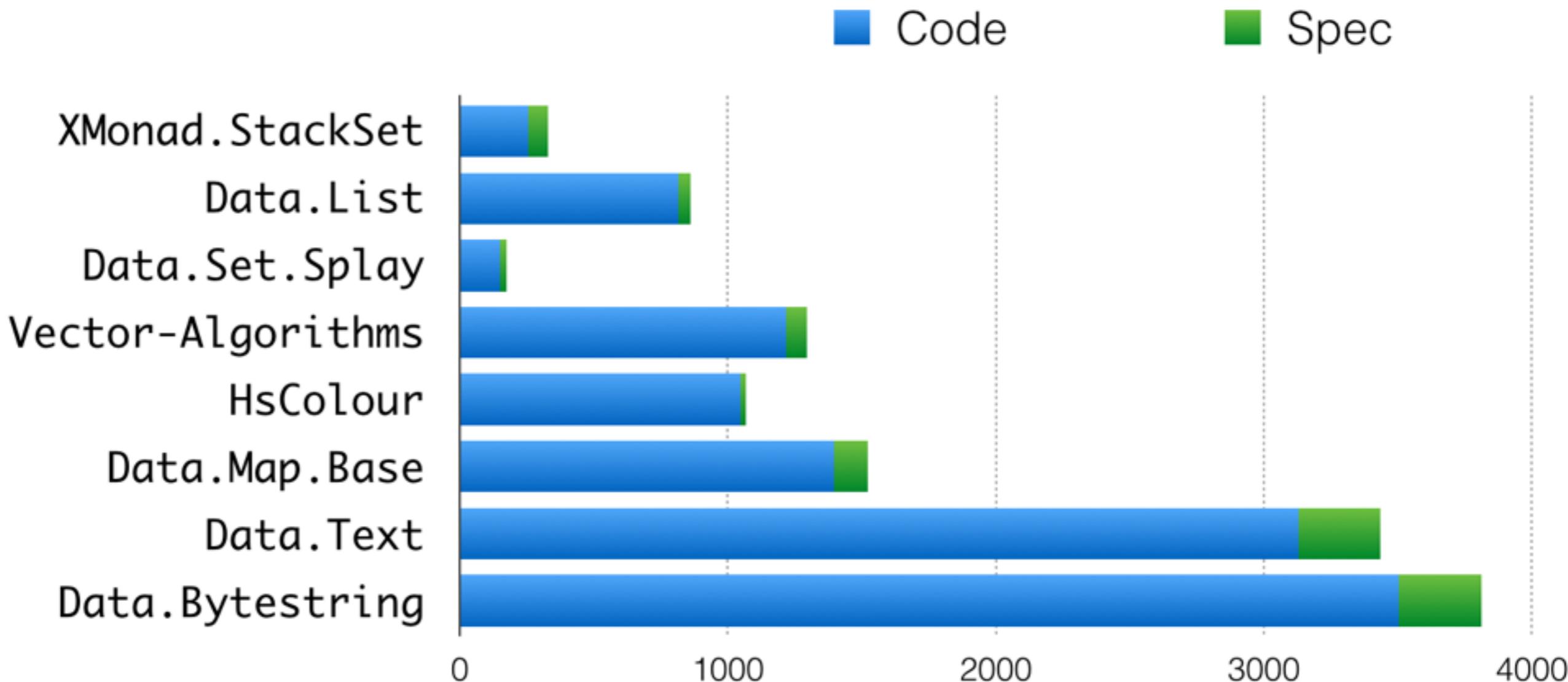
Data.List: list sortedness, safe indexing

Data.Map: binary-search tree preservation

Text, ByteString: low-level memory safety

XMonad: uniqueness of windows

# LiquidHaskell



specs: 1/10 LOC

**Modest Annotations**

time: 0.5s/10 LOC

# LiquidHaskell

as a theorem prover

**Modest Annotations BUT Verbose Proofs**

# LiquidHaskell

## as a theorem prover

Case study: String matcher parallelization

**200LoC** “runtime” code

**100LoC** specs on “runtime” code

**1300LoC** proof terms

**200LoC** theorem specs

proofs = 8 x code

**Verbose Proofs**

# Refinement Types for Haskell

## 1. Soundness

Under Lazy Evaluation

Thanks!

## 2. Expressiveness

Modular Higher-Order Specifications

Refinement Reflection

## 3. Implementation

LiquidHaskell

With decidable & predictable verification.

**END**

My research goal is to integrate verification into code development chain

## **Envision:**

# Integrate Verification into Code Development

# LiquidHaskell

## The crew

Alexander Bakst

Ranjit Jhala

Simon Peyton-Jones

Eric Seidel

Dimitrios Vytiniotis

...

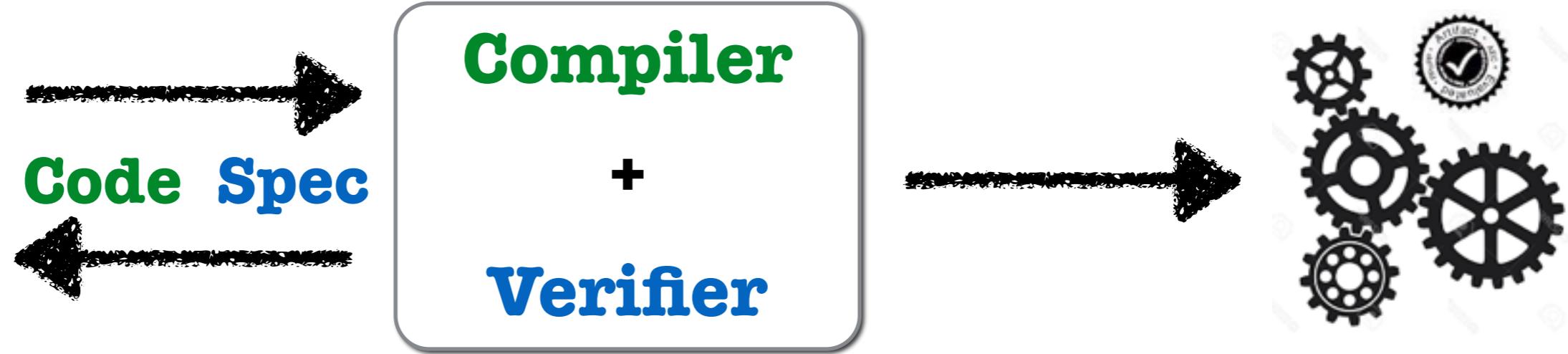
My research goal is to integrate verification into code development chain

## **Envision:**

# Integrate Verification into Code Development

# Goal: Verification into Code Development

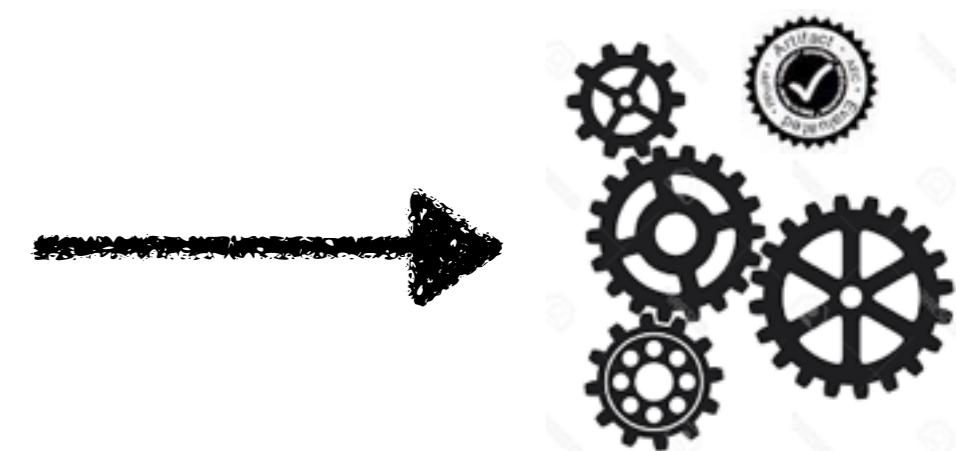
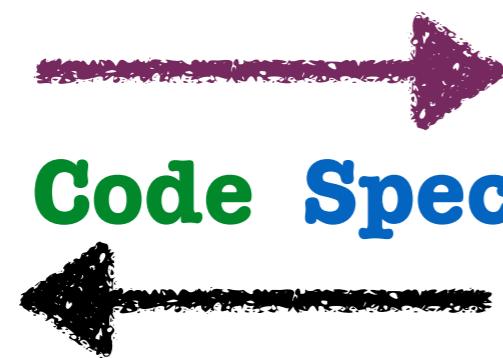
What I envision is that in this process the programmer also provides some code specifications that are automatically checked by a verifier ensuring that the final application satisfies the specifications



To achieve this, we need to persuade  
the programmer to write specifications

# Goal: Verification into Code D

User should **naturally** write Specifications ...



and the verifier should provide useful feedback

# Goal: Verification into Code D

User should **naturally** write Specifications ...



... and get **useful** feedback.

# User should **naturally** write Specifications ...

## Natural Integration

## Expressiveness

I believe that refinement types are in the right direction of naturally integrating specs into the language, as they simply refine the existing type system with logical predicates.

So the user does not need to learn a new language to express code of specification.

A second requirement for natural specification, is that spec language should be arbitrary expressive and this is a requirement that LH fails to satisfy

# Expressiveness

LiquidHaskell:  
Functions cannot appear in spec:

```
fib_increasing :: x:Nat -> y:Nat  
                  -> { x < y => fib x < fib y }  
fib_increasing x y = AutoProof
```

Right now LH does not allow arbitrary functions to appear inside the specifications which crucially restricts expressiveness.

As an example if I want to prove that fib is increasing I need to refer to the fib function inside the spec and say...

This is my current work, How to allow arbitrary functions appear inside specifications, while retaining decidability and predictability,

# Expressiveness

LiquidHaskell:  
Functions cannot appear in spec:

```
fib_increasing :: x:Nat -> y:Nat  
                  -> { x < y => fib x < fib y }  
fib_increasing x y = AutoProof
```

Right now LH does not allow arbitrary functions to appear inside the specifications which crucially restricts expressiveness.

As an example if I want to prove that fib is increasing I need to refer to the fib function inside the spec and say...

This is my current work, How to allow arbitrary functions appear inside specifications, while retaining decidability and predictability,

# Expressiveness

Once I achieve that we could express interesting higher order properties like that the monadic list implementation satisfies the left identity list monad, just by providing a functions that satisfies the specification that forall m:pa[ if I bind m to return I get m

LiquidHaskell:  
Functions cannot appear in specifications.

```
left_id    :: m:[a] -> { m >>= return == m }
left_id m = AutoProof
```

Then I am going to describe what I see  
as useful feedback by an example

# Goal: Verification into Code D

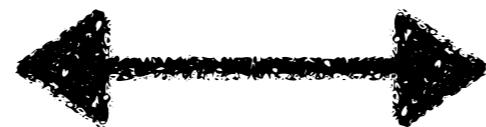
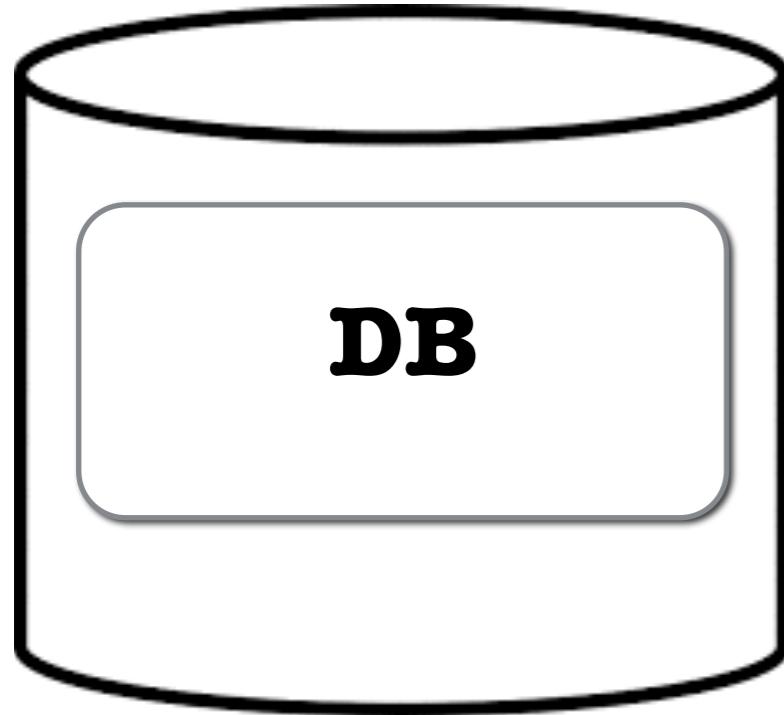
Natural Integration

Expressiveness



... and get **useful** feedback.

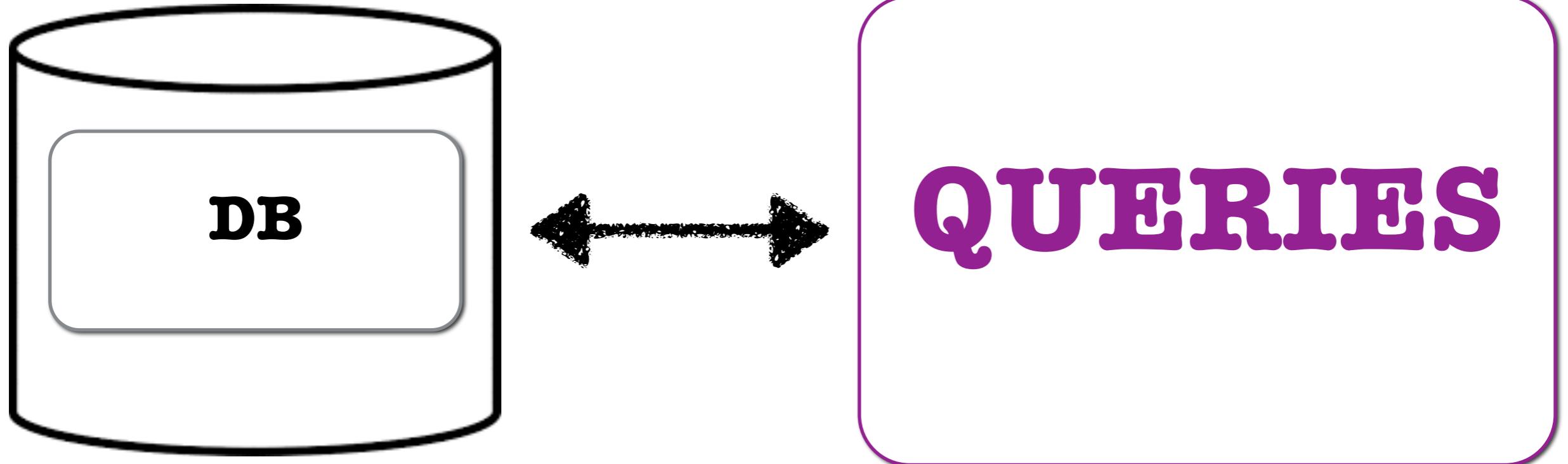
# ... and get useful feedback



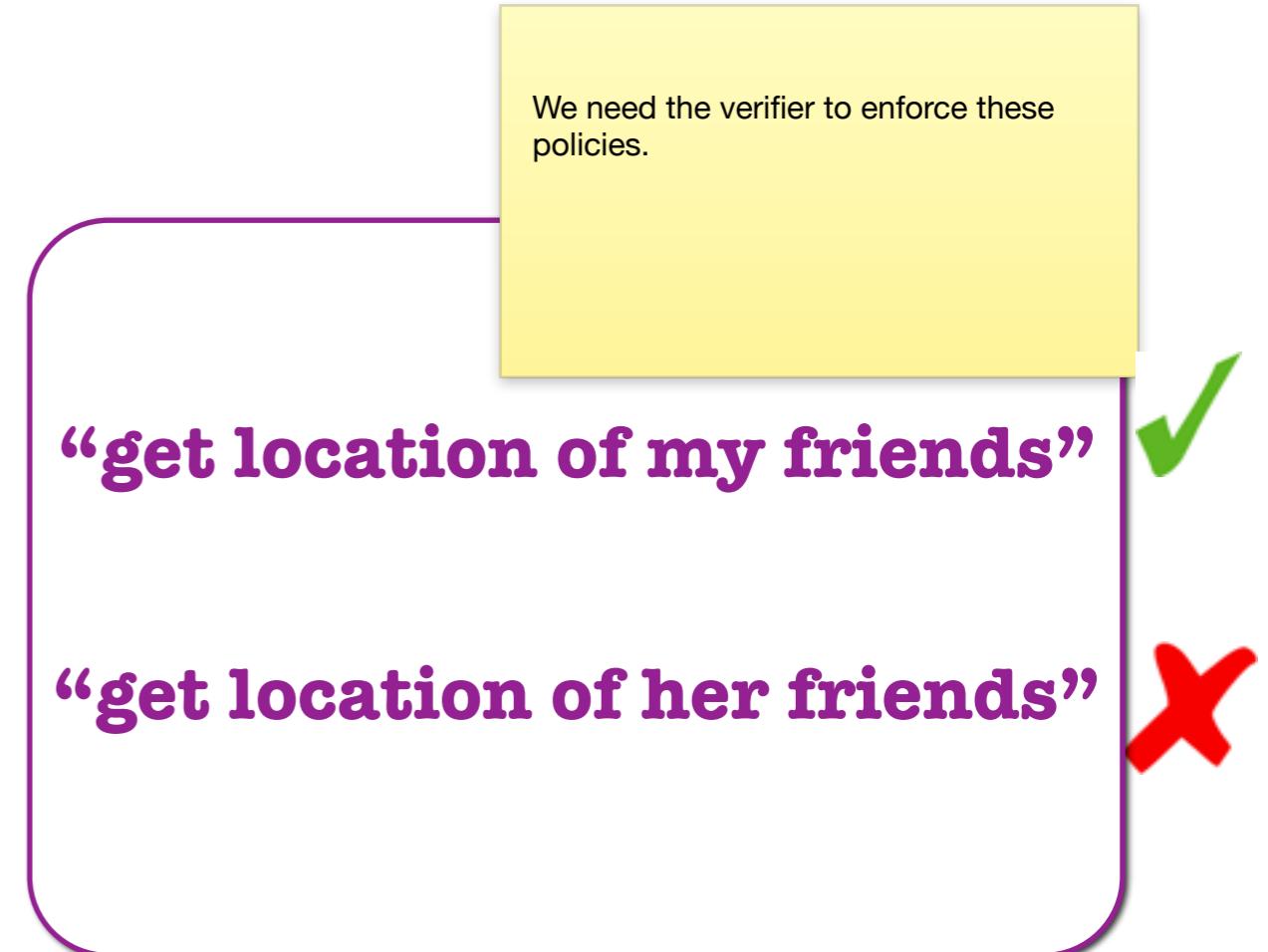
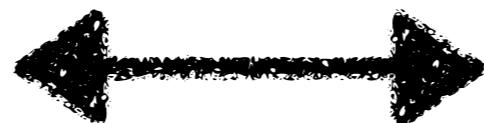
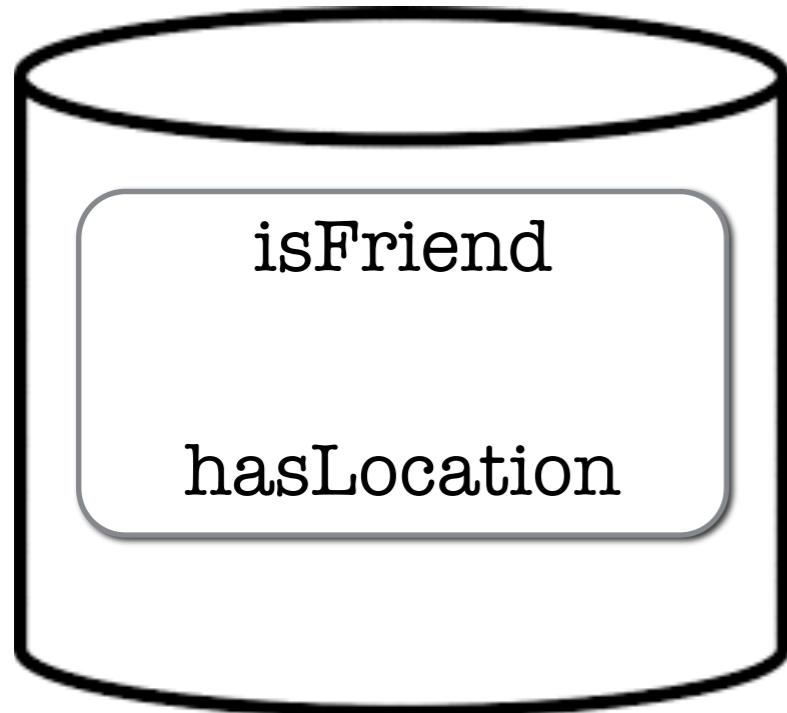
## QUERIES

Consider a db and an API that makes queries. it is not important what the query language is, but to be concrete I am going to use HAXI

... and get useful feedback.



# ... and get useful feedback.



**P1: “Anyone can look at my friend list.”**

**P2: “Only my friends can get my location.”**

# ... and get useful feedback.

```
main :: IO ()  
main = do
```

- me and her are two random users

```
let (myid, herid) = (42, 24)
```

- initialize the environment with my credentials

```
env <- initEnv $ UserState { me = myid }
```

- fetch the locations of MY friends

```
locs <- runHaxl env (getFriendsLocations myid)
```

**P1: “Anyone can look at my friend list.”**

**P2: “Only my friends can get my location.”**

Haxl\*: A Haskell library that accesses remote data

\*Simon Marlow et al. There is no Fork. (ICFP ’14)

# ... and get useful feedback.

```
main :: IO ()
```

```
main = do
```

- me and her are two random users

```
let (myid, herid) =
```

- initialize the environment

```
env <- initEnv $
```

**P1: “Anyone can look at my friend list.”**

**P2: “Only my friends can get my location.”**

If my query is SAFE,

Verification should succeed  
automatically

- fetch the locations of MY friends

```
locs <- runHaxl env (getFriendsLocations myid)
```



## Automation

# ... and get useful feedback.

```
main :: IO ()  
main = do  
    -- me and her are two random users  
    let (myid, herid) =  
        -- initialize the environment  
        env <- initEnv $  
            -- fetch the locations of MY friends  
            locs <- runHaxl env (getFriendsLocations herid)
```

- P1: “Anyone can look at my friend list.”
- P2: “Only my friends can get my location.”

If my query is UNSAFE,  
Verification should fail immediately

Speed

Automation

# ... and get useful feedback.

```
main :: IO ()
```

```
main = do
```

- me and her are two random users

```
let (myid, herid) = (42, 24)
```

- initialize the environment with my credentials

```
env <- initEnv $ UserState { me = myid }
```

- fetch the locations of MY friends

```
locs <- runHaxl env (getFriendsLocations herid) ✗
```

Speed

Automation

Error Diagnosis

**P1: “Anyone can look at my friend list.”**

**P2: “Only my friends can get my location.”**

Error: Violation of P2

# ... and get useful feedback.

```
main :: IO ()
```

```
main = do
```

- me and her are two random users

```
let (myid, herid) = (42, 24)
```

- initialize the environment with my credentials

```
env <- initEnv $ UserState { me = myid }
```

- fetch the locations of MY friends

```
locs <- runHaxl env (getFriendsLocations !)
```

Speed

Automation

Error Diagnosis

Program Synthesis

**P1: “Anyone can look at my friend list.”**

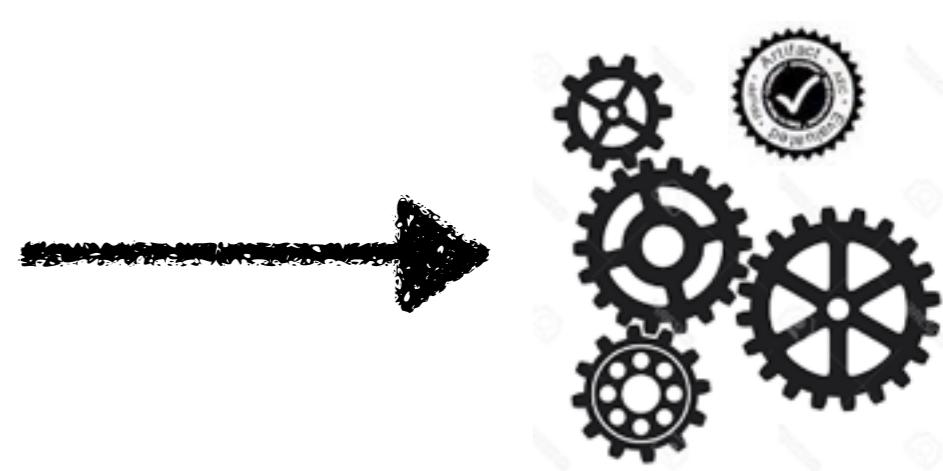
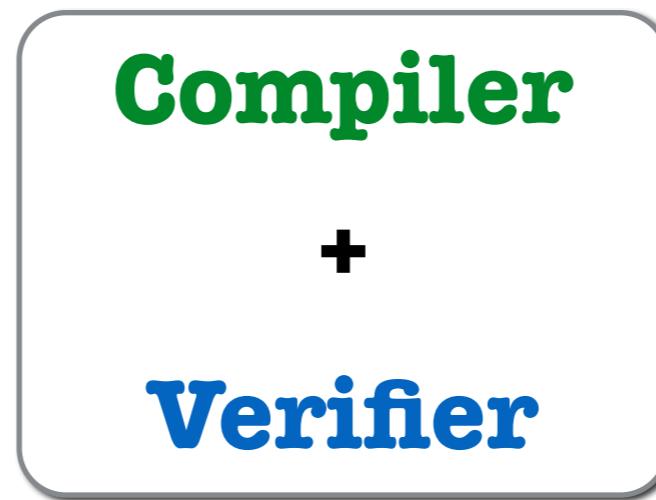
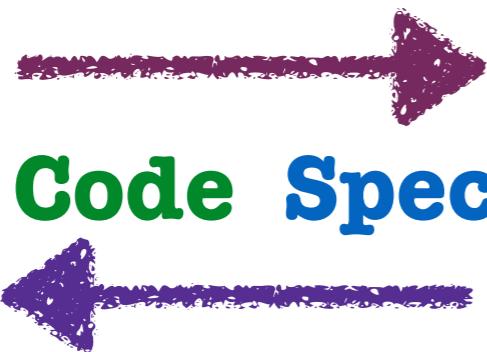
**P2: “Only my friends can get my location.”**

myid	p1, p2
herid	p1, p2

# Goal: Verification into Code Development

Natural Integration

Expressiveness



Speed

Automation

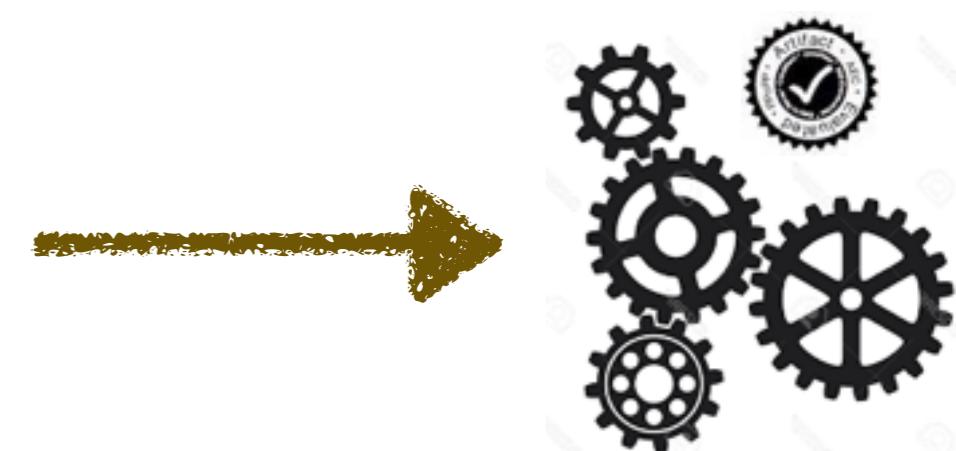
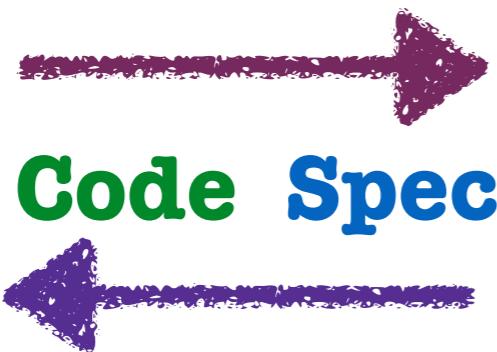
Error Diagnosis

Program Synthesis

# Goal: Verification into Code Development

Natural Integration  
Expressiveness

Data Privacy  
Web Security



Speed  
Automation  
Error Diagnosis  
Program Synthesis

***Thanks!***

**END**

# The Haxl Monad

For this specific example, HAxl provides a run function that says that if you give me an env that stores my credentials and a computation that decries a query, I will give you back the result of the query.

```
runHaxl :: env:Env      - the environment
          -> haxl:Haxl a - the computation
          -> IO a         - fetched data
```

# Refine the Haxl Monad

```
runHaxl :: (Valid Env (Haxl a))  
          => env:Env  
          -> {haxl:Haxl a | valid env haxl}  
          -> IO a
```

# The Haxl Monad

Haxl is concurrency monad that performs requests of data `a`

```
newtype Haxl a = Haxl
{ unHaxl :: Env           – environment reader
  -> IORef (RequestStore) – cached fetches reader
  -> IO (Result a) }      – data fetched
```

```
runHaxl :: env:Env       – the environment
          -> haxl:Haxl a – the computation
          -> IO a         – fetched data
```



# LiquidHaskell

Alexander Bakst

Ranjit Jhala

Simon Peyton-Jones

Eric Seidel

Michael Smith

Christopher Tetrault

Dimitrios Vytiniotis

...

# Usable Program Verification

## Expressiveness

Abstract Refinement Types

**N. Vazou**, P. M. Rondon, and R. Jhala. ESOP ‘13

Bounded Refinement Types

**N. Vazou**, A. Bakst, and R. Jhala. ICFP ‘15

## Error Reporting & Diagnosis

Type Targeted Testing

E. Seidel, **N. Vazou**, and R. Jhala. ESOP ‘15

## Prototype: LiquidHaskell

Refinement Types for Haskell

**N. Vazou**, E. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. ICFP ‘15

Experience with Refinement Types in the Real World

**N. Vazou**, E. Seidel, R. Jhala. Haskell ‘15