Refinement Types for Haskell

Niki Vazou
University of Maryland

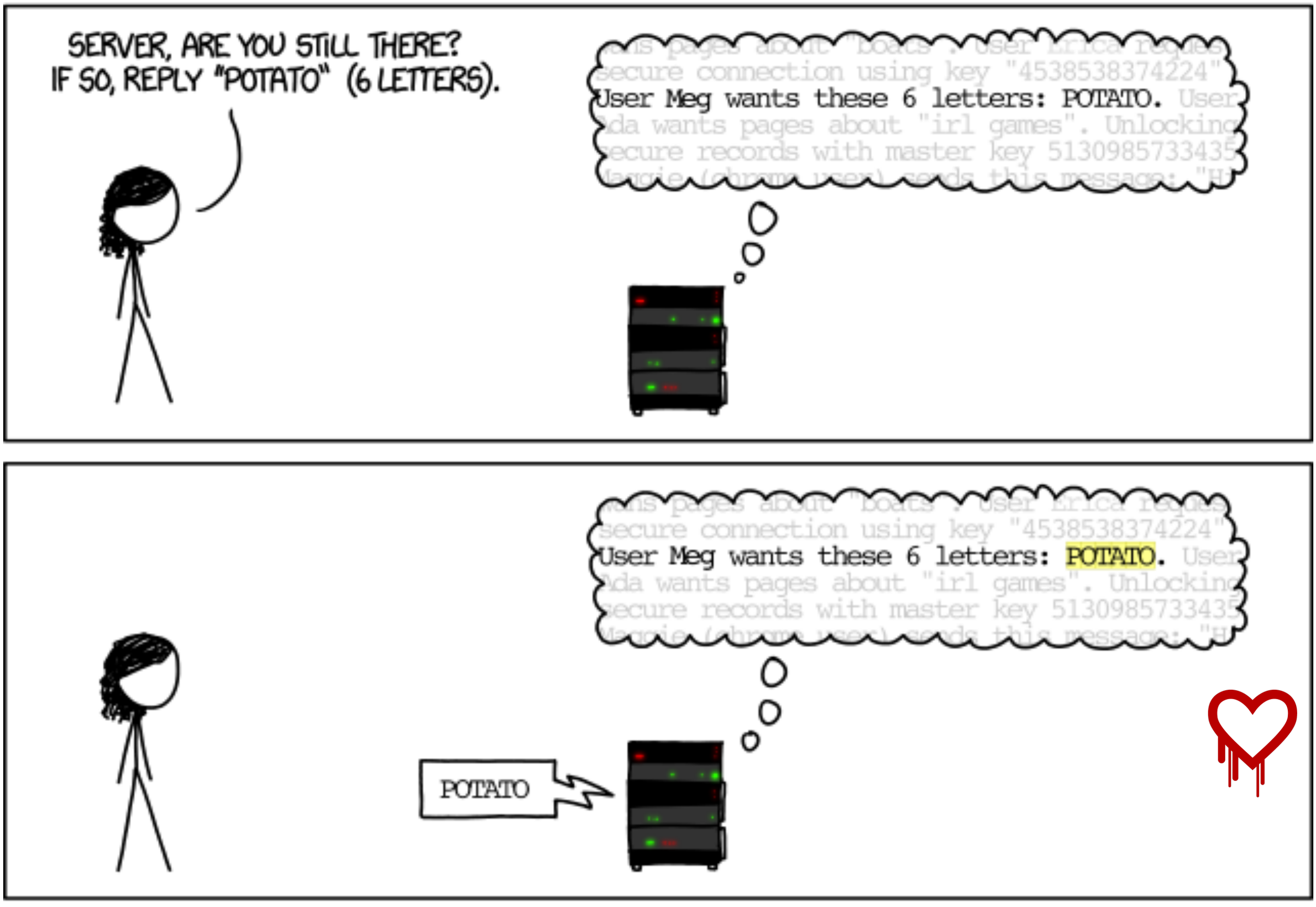
Software bugs are everywhere

Airbus A400M crashed due to a software bug.
— May 2015

Software bugs are everywhere

The Heartbleed Bug.
Buffer overflow in OpenSSL. 2015

HOW THE HEARTBLEED BUG WORKS:



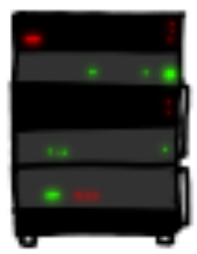
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

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).

User Meg wants these 500 letters: HAT.

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 "

Make bugs difficult to express

Using Modern Programming Languages

F#, Ocaml, Erlang, Scala, Haskell

Because of

Strong Types + λ-Calculus

Make bugs difficult to express

Using Mode guages

F#, Ocam , Haskell

Because of

ong Types + λ-Calculus

VS.

VS.

```
λ>  :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"

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

VS.

```
λ>  :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"

λ> takeWord16 pack True
    Type Error: Cannot match Bool vs Int
```

VS.

```
λ>  :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"

λ> takeWord16 pack 500
   “hat\58456\2594\SOH\NUL…
```


VS.

Valid Values for `takeWord16`?

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

All Ints

…, -2, -1, 0, 1, 2, 3, …

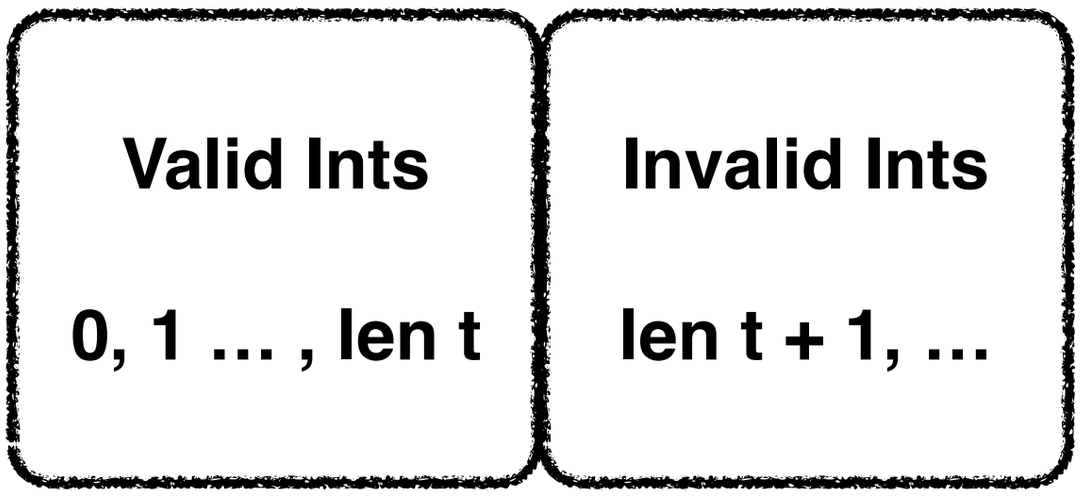
Valid Values for `takeWord16`?

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

Valid Ints	Invalid Ints
0, 1 … , len t	len t + 1, …

Refinement Types

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



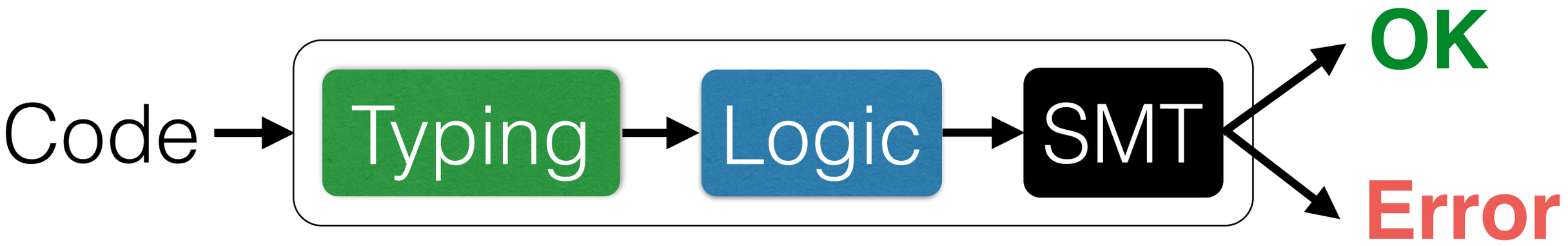
Refinement Types

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

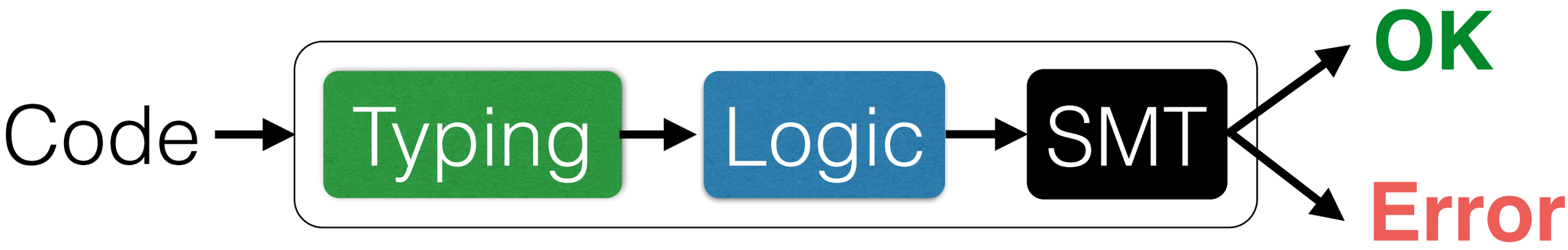
```
λ>  :m +Data.Text Data.Text.Unsafe
λ> let pack = "hat"

λ> take pack 500
   Refinement Type Error
```

Refinement Types

Refinement Types



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 8
```

Code → Typing

```
take :: t:Text -> {v | v <= len t} -> Text
heartbleed = let x = "hat"
             in  take x 8
```

```
x:{v|len v=3} |- {v|v=8} <: {v|v<=len x}
```

Code → Typing

```
take :: t:Text->{v | v <= len t}->Text
heartbleed = let x = "hat"
             in  take x 8
```

`x:{v|len v=3}` `|-` `{v|v=8}` `<:` `{v|v<=len x}`

Code → Typing

```
take :: t:Text -> {v | v <= len t} -> Text
heartbleed = let x = "hat"
             in take x 8
```

```
x:{v|len v=3} |- {v|v=8} <: {v|v<=len x}
```

Code → Typing

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

heartbleed = let x = "hat"
             in  take x 8
```

x:{v|len v=3} |- {v|v=8} <: {v|v<=len x}

Code → Typing

```
take :: t:Text->{v | v <= len t}->Text
heartbleed = let x = "hat"
             in  take x 8
```

```
x:{v|len v=3} |- {v|v=8} <: {v|v<=len x}
```

Encode Subtyping ...

```
x:{v|len v=3} |- {v|v=8} <: {v|v<=len x}
```

... as Logical VC

```
len x = 3 =>  (v = 8)  => (v <= len x)
```

Encode Subtyping ...

```
x:{v|len v=3} |- {v|v=8} <: {v|v<=len x}
```

... as Logical VC

```
len x = 3 =>  (v = 8)  => (v <= len x)
```

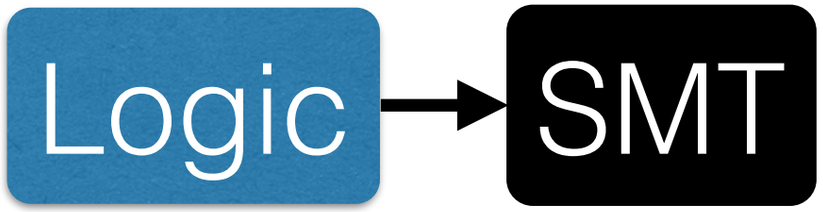
Typing → Logic

Encode Subtyping ...

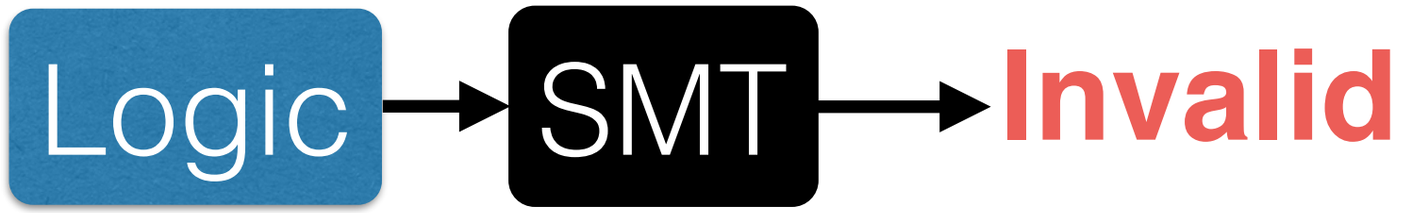
```
x:{v|len v=3} |- {v|v=8} <: {v|v<=len x}
```

... as Logical VC

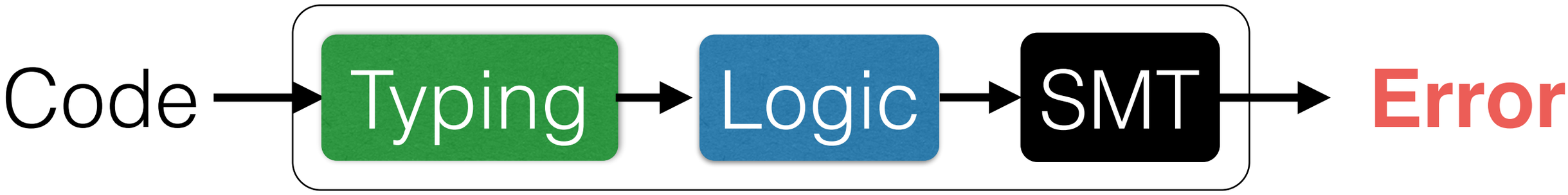
```
len x = 3 => (v = 8) => (v <= len x)
```




```
len x = 3 =>  (v = 8)  => (v <= len x)
```

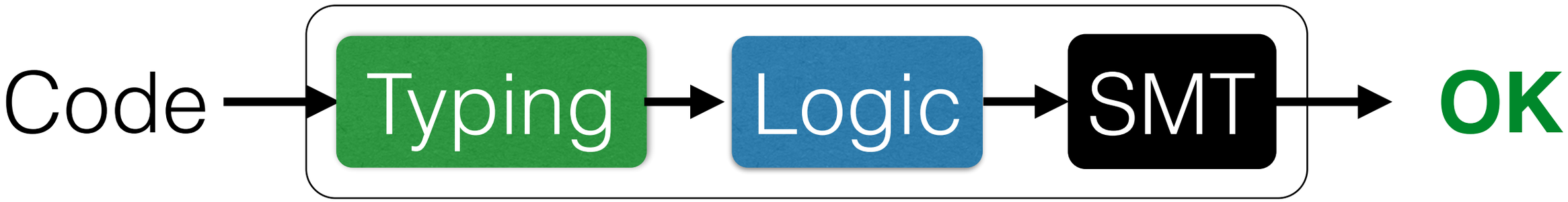



```
len x = 3 =>  (v = 8)  => (v <= len x)
```




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

heartbleed = let x = "hat"
             in  take x 8
```

```
len x = 3 =>  (v = 8)  => (v <= len x)
```

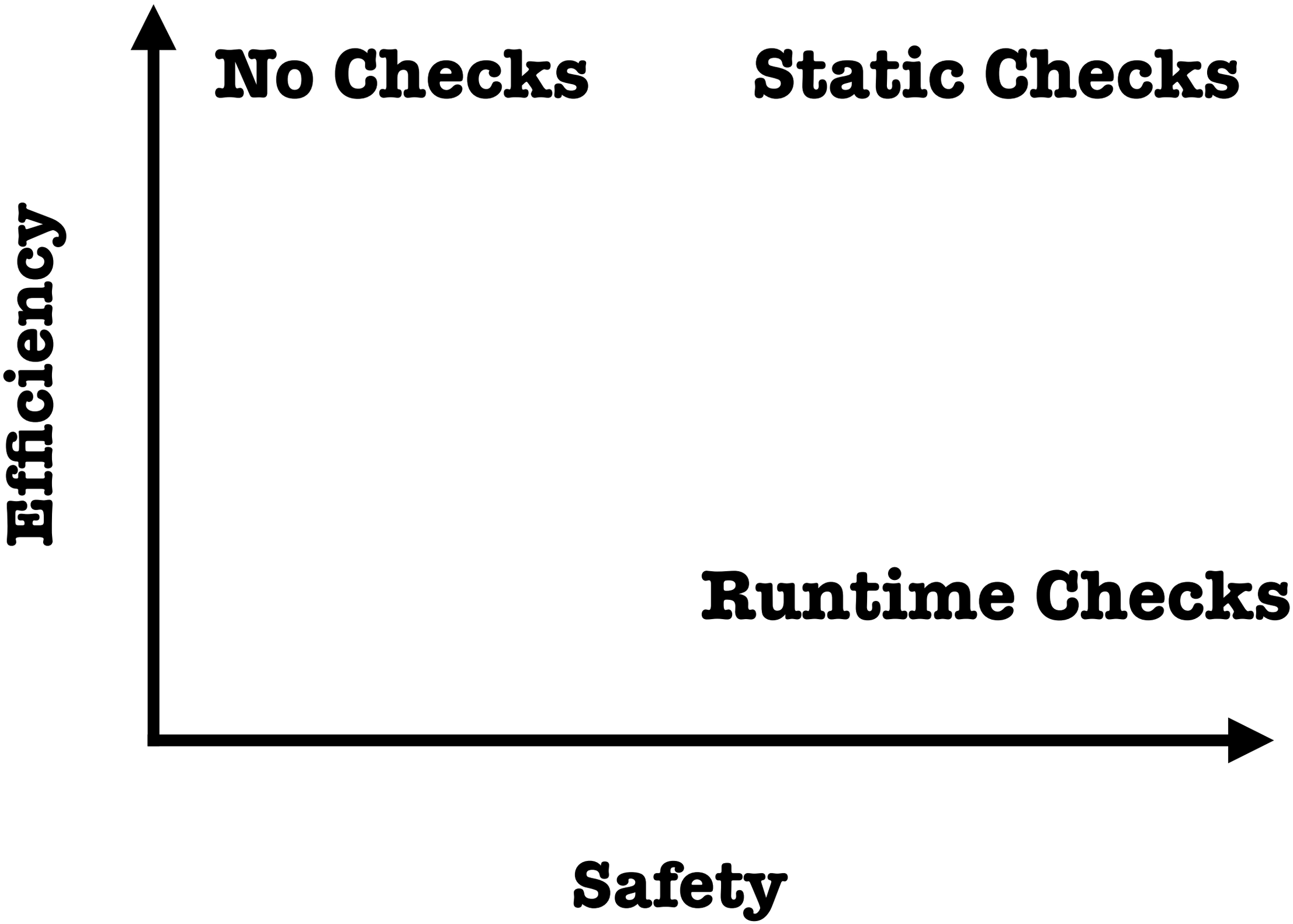



```
take :: t:Text->{v | v <= len t}->Text

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

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

Checks valid arguments, under facts.

Static Checks


No Checks
Static Checks
Efficiency
Runtime Checks
Safety

No Checks

```
take :: t:Text -> i:Int -> Text
take i t
  = Unsafe.takeWord16 i t
```

No Checks

```
take :: t:Text -> i:Int -> Text
take i t
 = Unsafe.takeWord16 i t
```

```
heartbleed = take "hat" 500
```

OK

No Checks

```
take :: t:Text->i:Int->Text
take i t
 = Unsafe.takeWord16 i t
```

```
heartbleed = take "hat" 500
```

OK

```
λ> heartbleed
λ> “hat\58456\2594\SOH\NUL…
```

UNSAFE

Runtime Checks

```
take :: t:Text -> i:Int -> Text
take i t | i < len t
  = Unsafe.takeWord16 i t
take i t
  = error “Out Of Bounds!”
```

Runtime Checks

```
take :: t:Text -> i:Int -> Text
take i t | i < len t
  = Unsafe.takeWord16 i t
take i t
  = error “Out Of Bounds!”
```

Runtime Checks

```
take :: t:Text -> i:Int -> Text
take i t | i < len t
  = Unsafe.takeWord16 i t
take i t
  = error “Out Of Bounds!”
```

```
heartbleed = take "hat" 500
```

OK

Runtime Checks

```
take :: t:Text->i:Int->Text
take i t | i < len t
 = Unsafe.takeWord16 i t
take i t
 = error “Out Of Bounds!”
```

OK

```
heartbleed = take "hat" 500
```

SAFE

```
λ> heartbleed
λ> *** Exception: Out Of Bounds!
```

Runtime Checks are expensive

```
take :: t:Text -> i:Int -> Text
take i t | i < len t
  = Unsafe.takeWord16 i t
take i t
  = error “Out Of Bounds!”
```

Static Checks

```
take :: t:Text->i:{i < len t}->Text
take i t | i < len t
  = Unsafe.takeWord16 i t
take i t
  = error “Out Of Bounds!”
```

Static Checks

```
take :: t:Text -> i:{i < len t} -> Text
take i t | i < len t
  = Unsafe.takeWord16 i t
  = take i t
  = error “Out Of Bounds!”
```

Static Checks

```
take :: t:Text->i:{i < len t}->Text
take i t
  = Unsafe.takeWord16 i t
```

Static Checks

```
take :: t:Text->i:{i < len t}->Text
take i t
 = Unsafe.takeWord16 i t
```

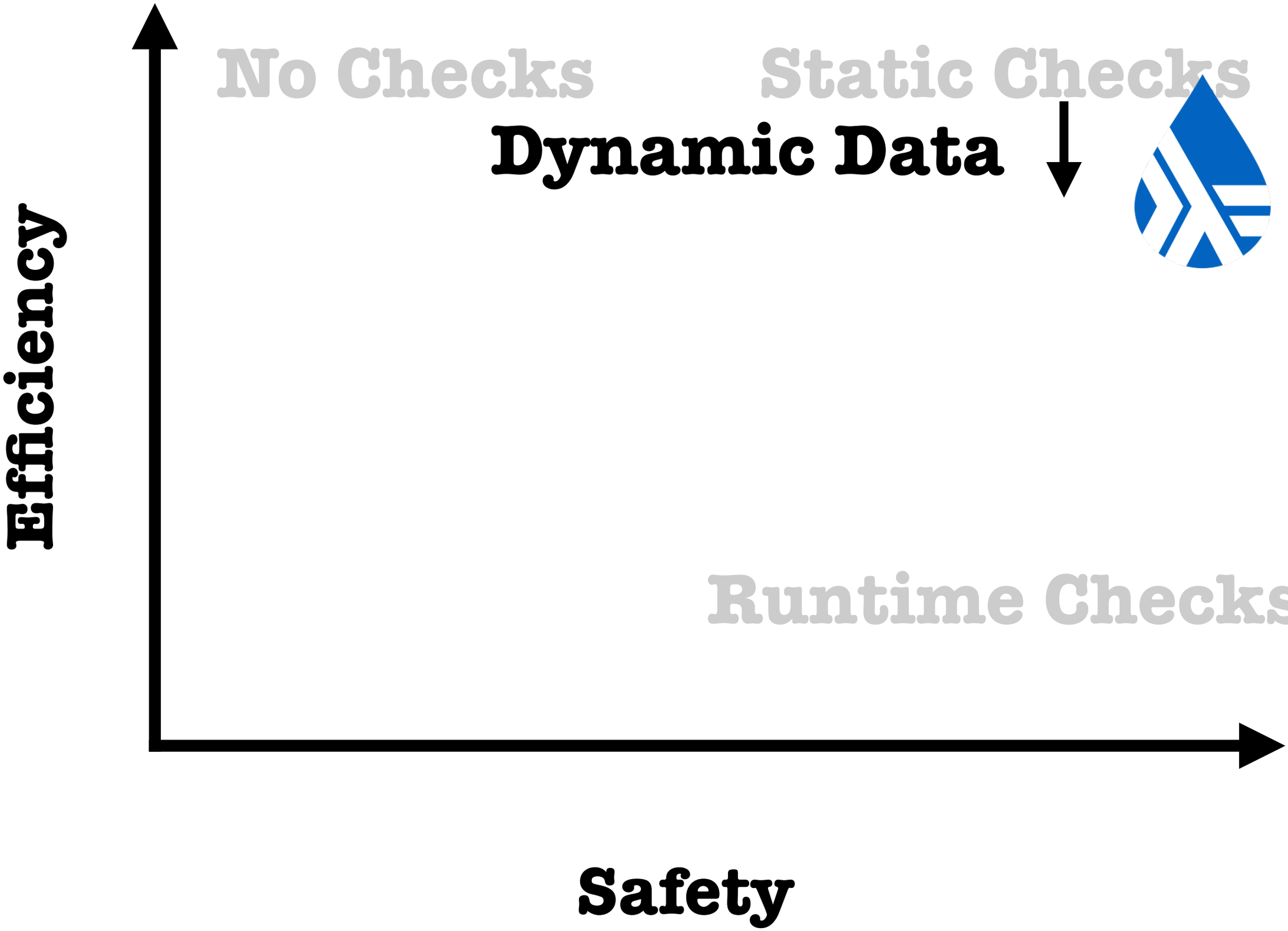
```
heartbleed = take "hat" 500
```

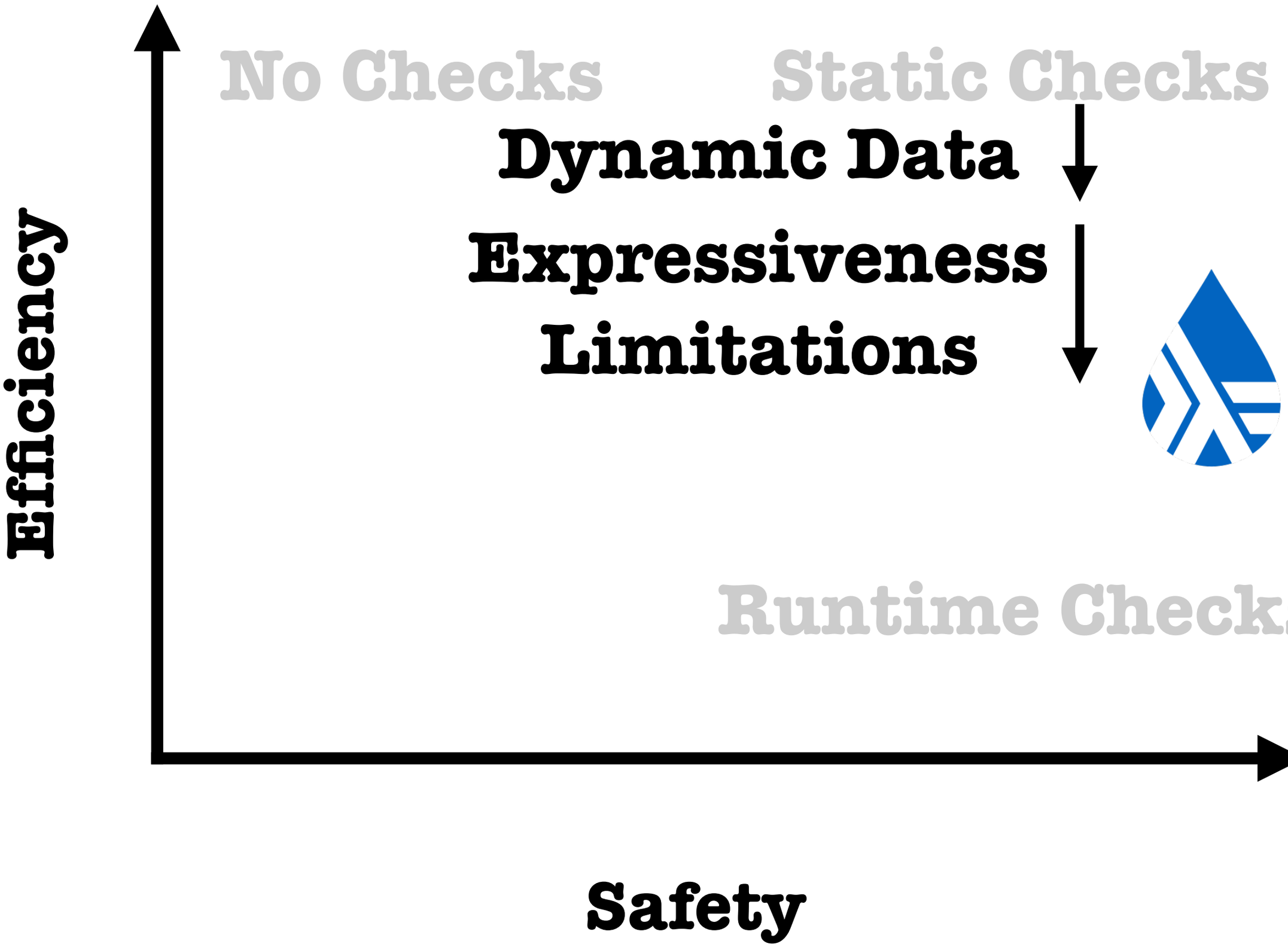
UNSAFE

Static Checks

Safe & Efficient Code!


No Checks
Static Chec
Runtime Checks
Efficiency
Safety


No Checks
Static Checks
Dynamic Data
Runtime Checks
Efficiency
Safety


No Checks
Static Checks
Dynamic Data
Expressiveness
Limitations
Runtime Checks
Efficiency
Safety

Expressiveness

What properties can be expressed in types?

Expressiveness vs. Automation

Expressiveness vs. Automation

If p is safe indexing ...

```
{t:Text | i < len t }
```

... then SMT-automatic verification.

Expressiveness vs. Automation

If p from decidable theories ...

{t:a | p}

... then SMT-automatic verification.

Expressiveness vs. Automation

If p from decidable theories ...

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

Boolean Logic

(QF) Linear Arithmetic

Uninterpreted Functions ...

... then SMT-automatic verification.

What about expressiveness?

Automation

Refinement Types

Friday @13:40

F*, Dafny

Tactics

Dependent Types
(Coq, Agda, …)

Expressiveness

Friday @13:40

Refinement Types for Theorem Proving

Theorem Proving

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

Theorem Proving

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

Theorem Proving

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

Can we prove **theorems** about functions?

```
\forall i. 0 <= i => fib i <= fib (i+1)
```

Theorem Proving

Can we prove **theorems** about functions?

Theorem Proving

Express **theorems** via refinement types.

Express **proofs** via functions.

Check that functions prove theorems.

Known as
Curry-Howard correspondence

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

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

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

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

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

Theorem Proving

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
```

Theorem Proving

Refinement Reflection

Express **theorems** via refinement types.

Express **proofs** via functions.

Check that functions prove theorems.

as a theorem prover

Case study: String matcher parallelization

200LoC "runtime" code

100LoC specs on "runtime" code

1300LoC proof terms

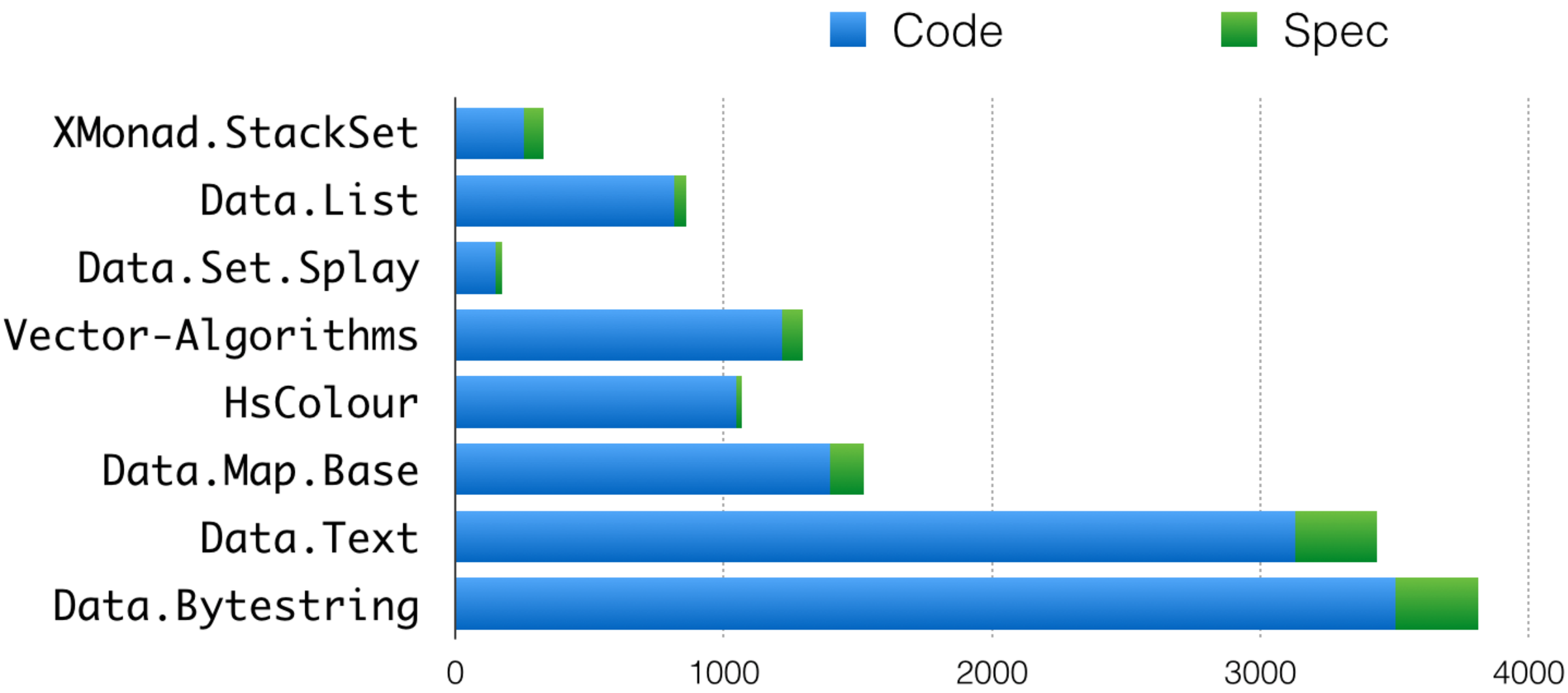
200LoC theorem specs

proofs = 8 x code

Verbose Proofs

Verbose Proofs BUT Modest Annotations

LiquidHaskell



specs: 1/10 LOC

time: 0.1s/10 LOC

Modest Annotations

Modest Annotations

SMT Automation

Expressiveness

Refinement Reflection

Modest Annotations

SMT Automation

Fast & Safe Code

Static Checks

Expressiveness

Refinement Reflection

Modest Annotations

SMT Automation

Thanks!

END