Refinement Types for Haskell

Niki Vazou
University of Maryland

Software bugs are everywhere

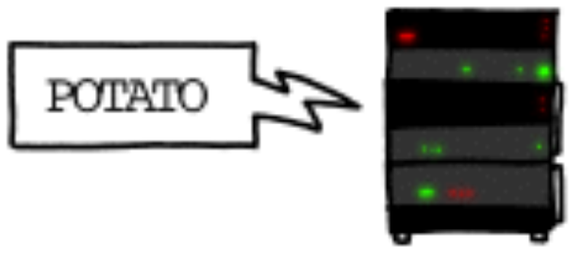
Airbus A400M crashed due to a software bug.
— May 2015

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 "POTATO" (6 LETTERS).



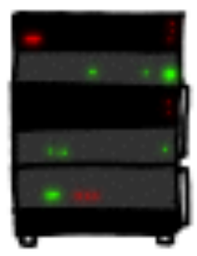
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

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

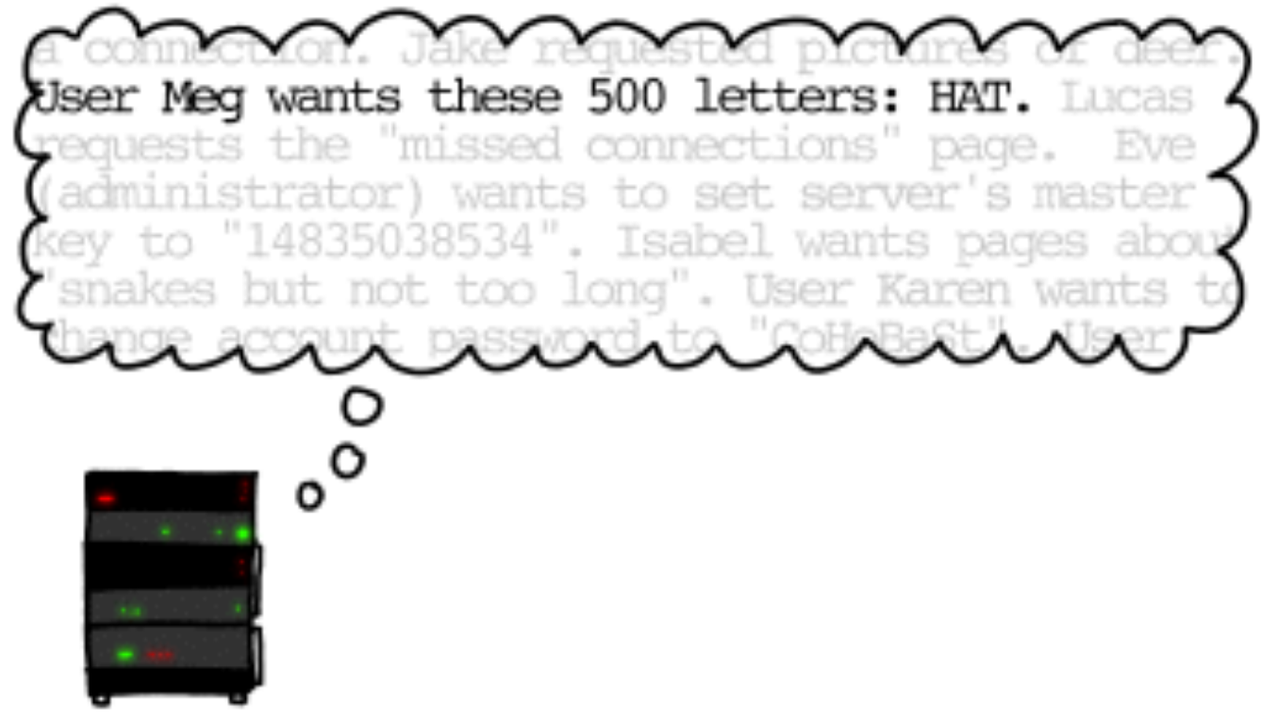
BIRD

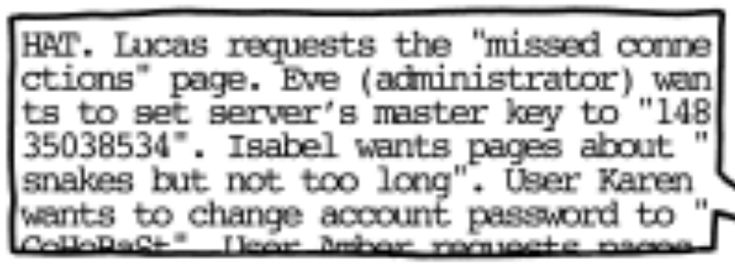


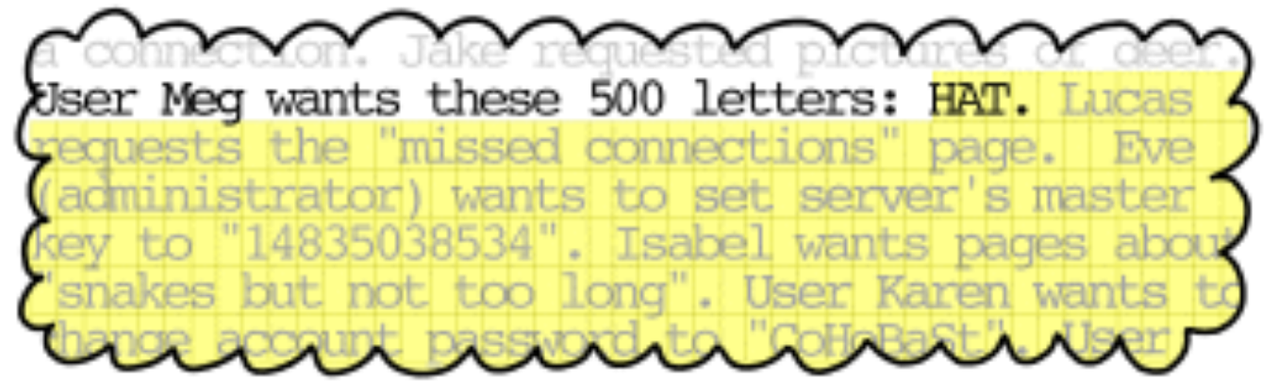
SERVER, ARE YOU STILL THERE?

User Meg wants these 500 letters: HAT. Lucas

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







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, …

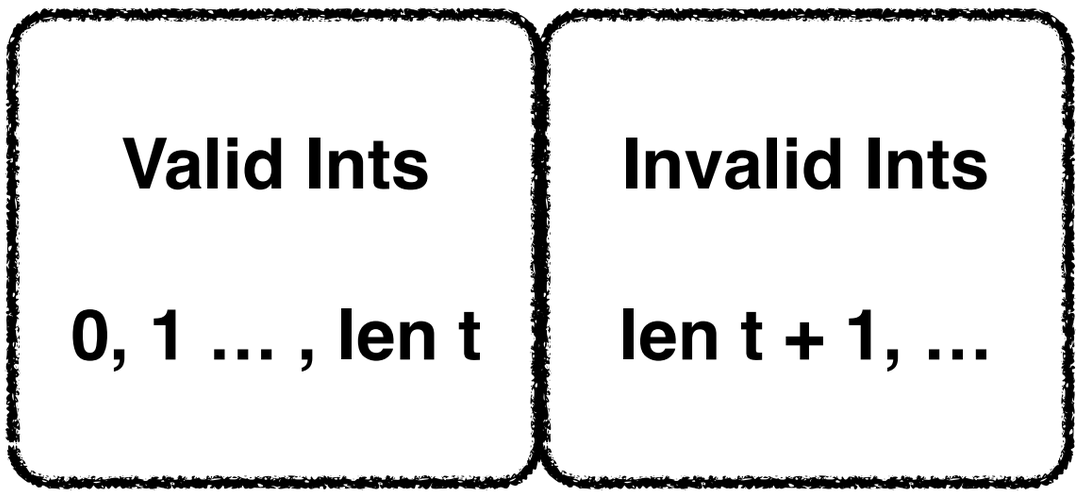
Valid Values for `takeWord16`?

```
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

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.

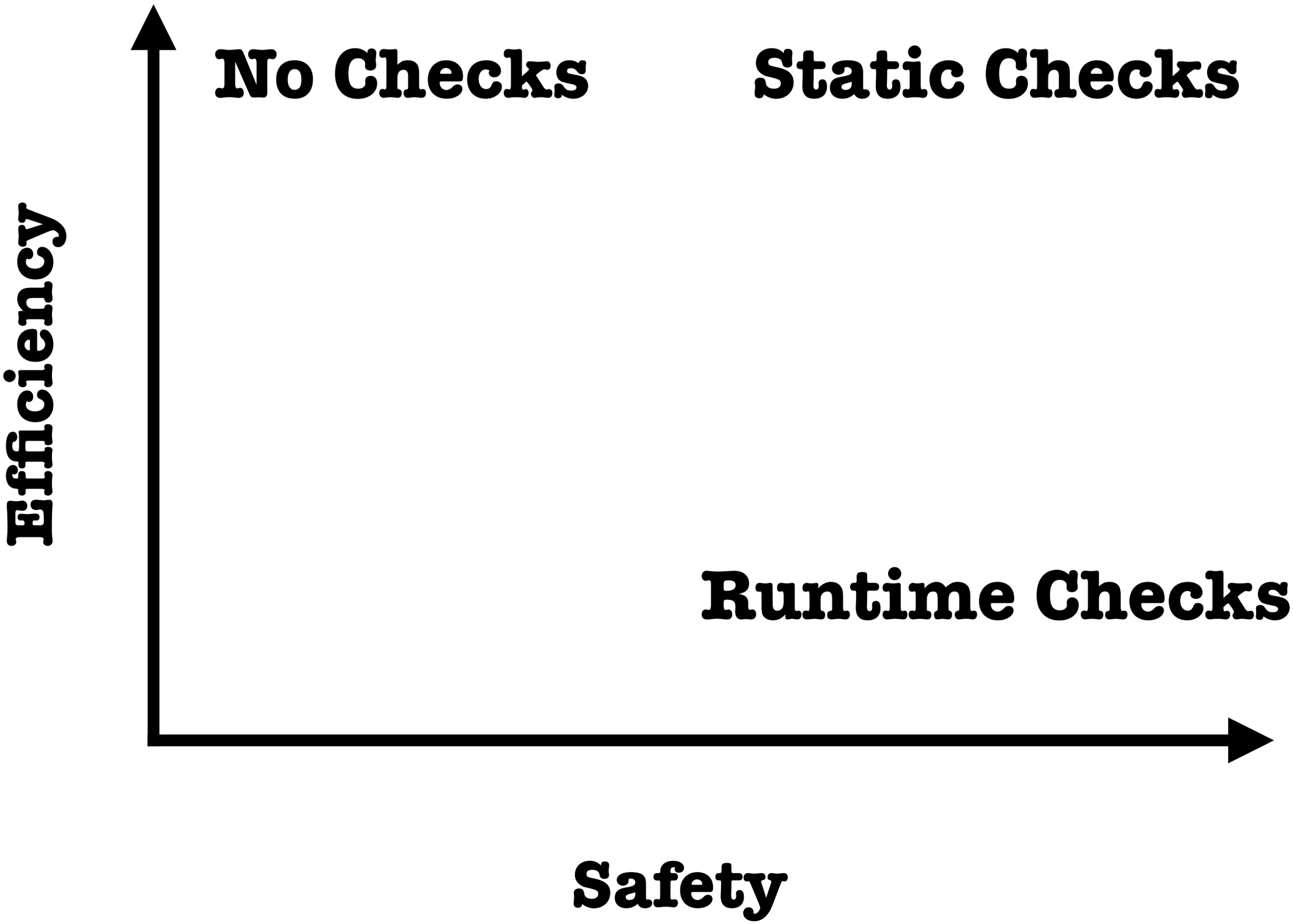
```
take :: t:Text->{v | v < len t}->Text

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

Checker reports **Error**

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 & Fast Code!

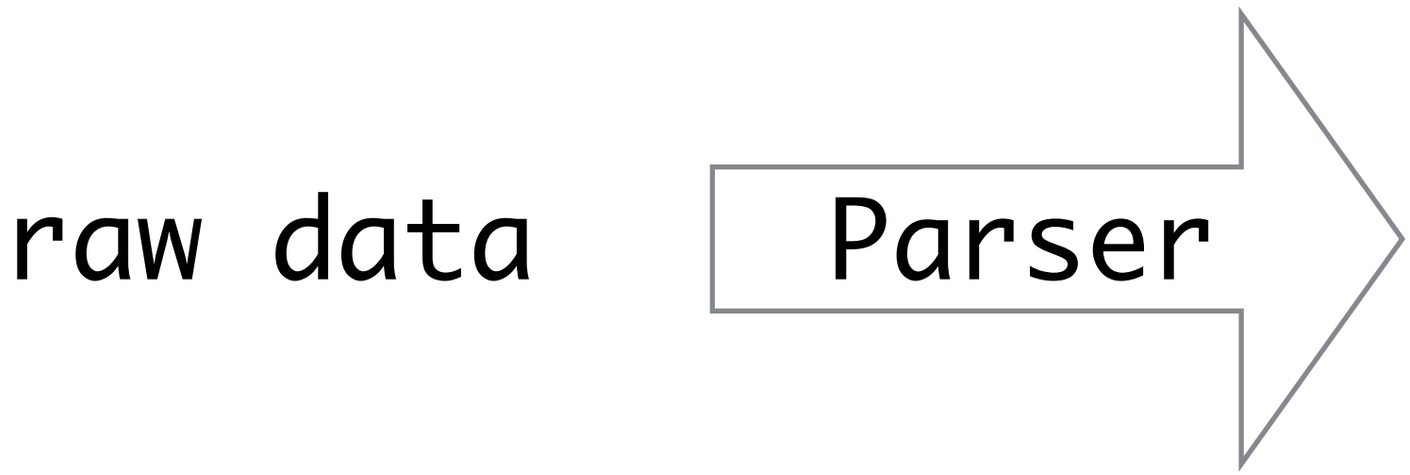
Example: Parsing UDP

Example: Parsing UDP*

User Datagram Protocol

*Gabriel Gonzalez (Awake Networks)

Example: Parsing UDP*



*Gabriel Gonzalez (Awake Networks)

Example: Parsing UDP

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

Example: Parsing UDP

```
data Parser a = Parser {
    runP :: Text -> Maybe (a, Text) }
```

Example: Parsing UDP

```haskell
data Parser a = Parser {
    runP :: Text -> Maybe (a, Text) }
```

```haskell
takeP :: Int -> Parser Text
takeP n = Parser (\t ->
    if length t < n
     then Nothing
     else Just (US.splitAt n t))
```

Example: Parsing UDP

```
udp :: Parser UDP
udp = do
    udpSrcPort  <- takeP 2
    udpDestPort <- takeP 2
    udpLength   <- takeP 2
    udpChecksum <- takeP 2
    return (UDP {..})
```

Safe but Slow (4 runtime checks)

Solution: Inline takeP

Example: Parsing UDP

```
udp2 :: Parser UDP
udp2 = Parser (\bs0 ->
  if length bs0 < 8
   then Nothing
   else do
     let (udpSrcPort , bs1) = US.splitAt 2 bs0
     let (udpDestPort, bs2) = US.splitAt 2 bs1
     let (udpLength  , bs3) = US.splitAt 2 bs2
     let (udpChecksum, bs4) = US.splitAt 2 bs3
     return (Just (UDP {..}, bs4))
```

Fast & Safe

Example: Parsing UDP

```
udp2 :: Parser UDP
udp2 = Parser (\bs0 ->
  if length bs0 < 8
   then Nothing
   else do
     let (udpSrcPort , bs1) = US.splitAt 2 bs0
     let (udpDestPort, bs2) = US.splitAt 4 bs1
     let (udpLength  , bs3) = US.splitAt 2 bs2
     let (udpChecksum, bs4) = US.splitAt 2 bs3
     return (Just (UDP {..}, bs4))
```

Fast & Safe, bur error prone

Example: Parsing UDP

```
udp2 :: Parser UDP
udp2 = Parser (\bs0 ->
  if length bs0 < 8
   then Nothing
```

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

```
    let (udpLength  , bs3) = US.splitAt 2 bs2
    let (udpChecksum, bs4) = US.splitAt 2 bs3
    return (Just (UDP {..}, bs4))
```

Enforce Static Checks!

Example: Parsing UDP

UNSAFE

```
udp2 :: Parser UDP
udp2 = Parser (\bs0 ->
  if length bs0 < 8
   then Nothing
   else do
    let (udpSrcPort , bs1) = US.splitAt 2 bs0
    let (udpDestPort, bs2) = US.splitAt 4 bs1
    let (udpLength  , bs3) = US.splitAt 2 bs2
    let (udpChecksum, bs4) = US.splitAt 2 bs3
    return (Just (UDP {..}, bs4))
```

Enforce Static Checks!

Example: Parsing UDP

SAFE

```
udp2 :: Parser UDP
udp2 = Parser (\bs0 ->
  if length bs0 < 8
   then Nothing
   else do
     let (udpSrcPort , bs1) = US.splitAt 2 bs0
     let (udpDestPort, bs2) = US.splitAt 2 bs1
     let (udpLength  , bs3) = US.splitAt 2 bs2
     let (udpChecksum, bs4) = US.splitAt 2 bs3
     return (Just (UDP {..}, bs4))
```

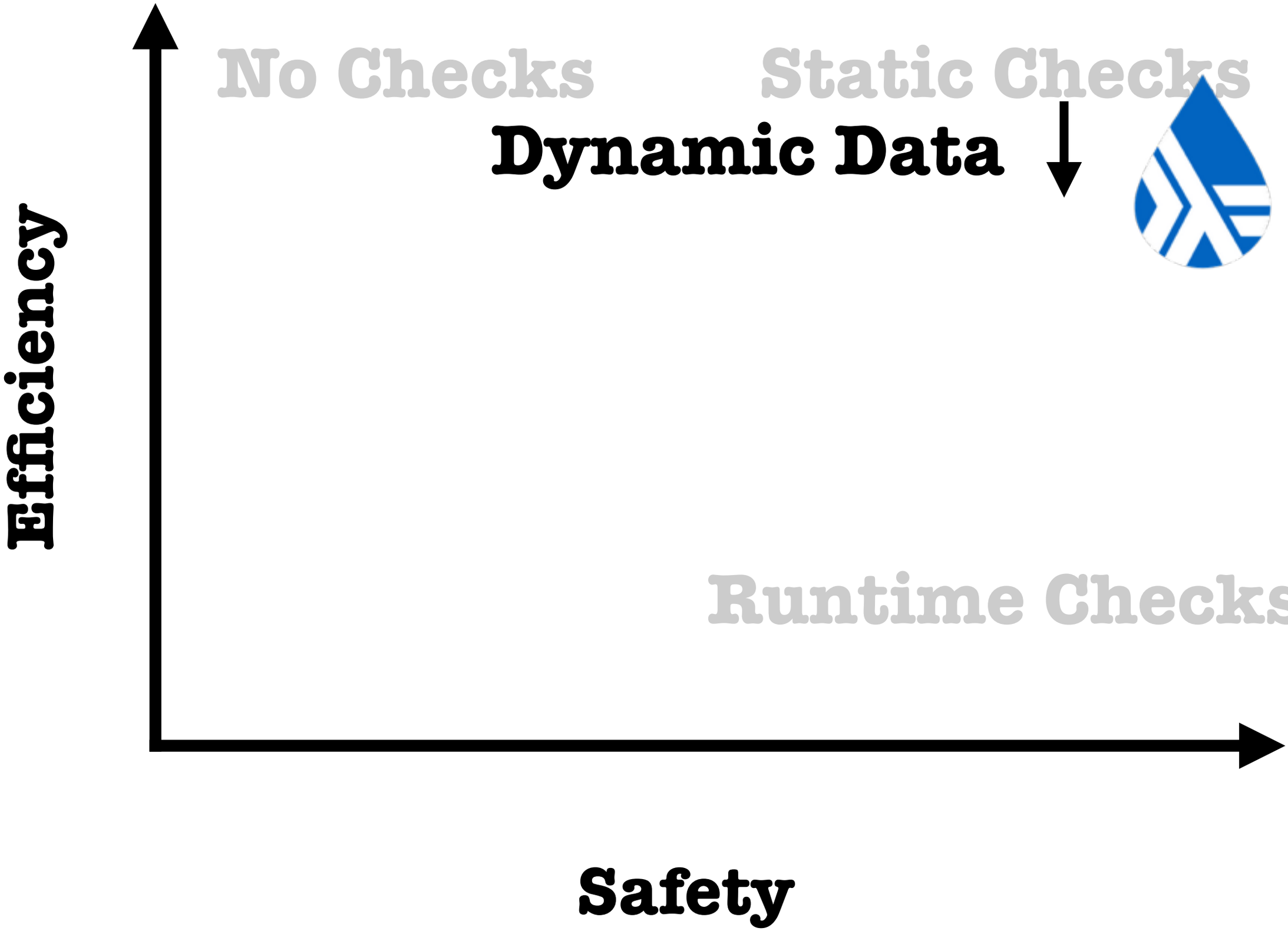
Provably Correct & Faster (x6) Code!

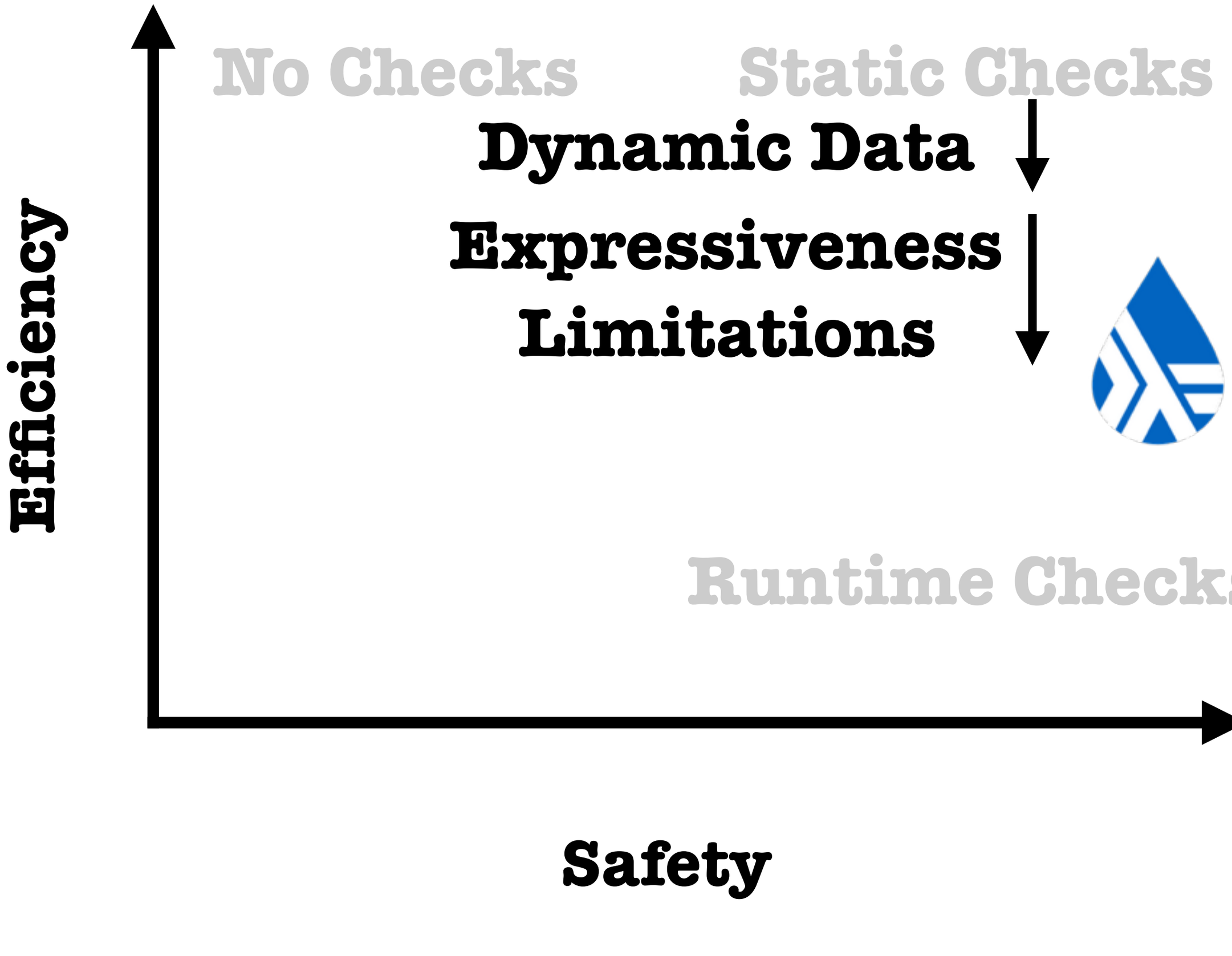
Example: Parsing UDP

Provably Correct & Faster (x6) Code!

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 ...

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

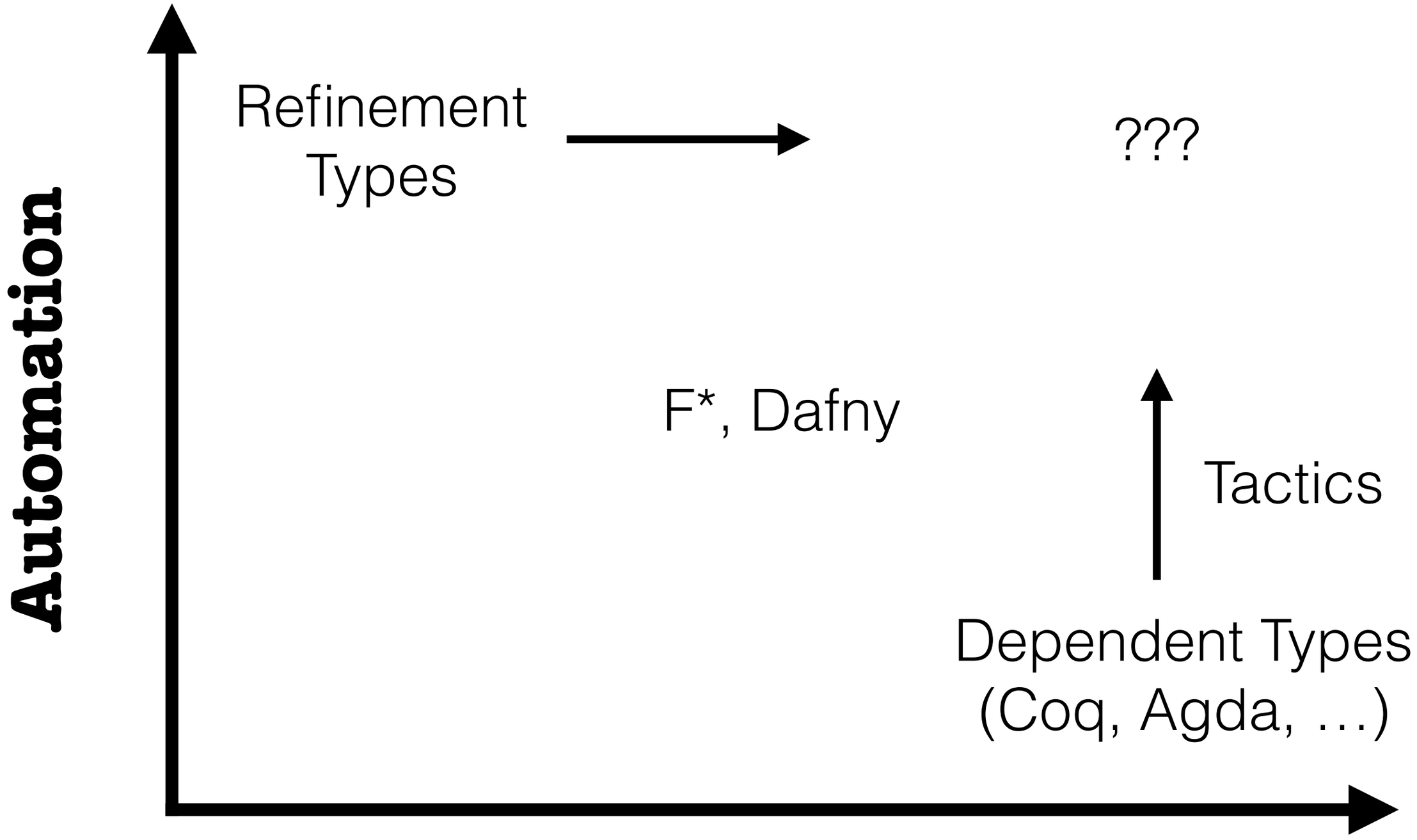
Boolean Logic

(QF) Linear Arithmetic

Uninterpreted Functions ...

... then SMT-automatic verification.

What about expressiveness?


Automation
Refinement Types
???
F*, Dafny
Tactics
Dependent Types (Coq, Agda, …)
Expressiveness

Expressiveness

Functional Correctness

Totality

Termination

Theorem Proving

Functional Correctness

```
type SList a = [a]<{\hd tl -> hd <= tl}>
```

Abstract Refinement Types

Functional Correctness

```
type SList a = [a]<{\hd tl -> hd <= tl}>
```

```
ups :: SList Int
ups = [1, 2, 3, 4]
```

SAFE

Functional Correctness

```
type SList a = [a]<{\hd tl -> hd <= tl}>
```

```
insert :: (Ord a) => a -> SList a ->  SList a
insert x [] = [x]
insert x (y:ys)
  | x <= y     = x:y:ys
  | otherwise = y:insert x ys
```

SAFE

Functional Correctness

```
type SList a = [a]<{\hd tl -> hd <= tl}>
```

```
insert :: (Ord a) => a -> SList a ->  SList a
insert x [] = [x]
insert x (y:ys)
  | x <= y     = x:y:ys
  | otherwise = y:insert x ys
```

```
insertSort :: (Ord a) => [a] -> SList a
insertSort = foldr insert []
```

SAFE

Expressiveness

Functional Correctness

Totality

Termination

Theorem Proving

Totality

```haskell
head :: [a] -> a
head (x:_) = x
```

Totality

```
head :: [a] -> a
head (x:_) = x
head _     = internalError “Empty List”
```

Totality

UNSAFE

```
head :: [a] -> a
head (x:_) = x
head _     = internalError “Empty List”
```

```
internalError :: {String | false} -> a
```

Totality

SAFE

```
head :: {v:[a] | 0 < len v} -> a
head (x:_) = x
head _     = internalError “Empty List”
```

```
internalError :: {String | false} -> a
```

Totality

SAFE

```
head :: {v:[a] | 0 < len v} -> a
head (x:_) = x
head _     = internalError “Empty List”
```

UNSAFE

```
foldl1 :: _ -> [a] -> a
foldl1 f xs = foldl f (head xs) (tail xs)
```

Totality

SAFE

```
head :: {v:[a] | 0 < len v} -> a
head (x:_) = x
head _     = internalError “Empty List”
```

SAFE

```
foldl1 :: _ -> {v:[a] | 0 < len v} -> a
foldl1 f xs = foldl f (head xs) (tail xs)
```

Expressiveness

Functional Correctness

Totality

Termination

Theorem Proving

Termination

UNSAFE

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

Sized Types

Termination

SAFE

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

Sized Types

Termination

UNSAFE

```
merge :: xs:SList a -> ys:SList a
      -> SList a / [len xs + len ys]
merge [] ys   = ys
merge xs []   = xs
merge (x:xs) (y:ys)
  | x < y       = x:merge xs (y:ys)
  | otherwise = y:merge (x:xs) ys
```

Termination

SAFE

```
merge :: xs:SList a -> ys:SList a
      -> SList a / [len xs + len ys]
merge [] ys   = ys
merge xs []   = xs
merge (x:xs) (y:ys)
  | x < y       = x:merge xs (y:ys)
  | otherwise = y:merge (x:xs) ys
```

Expressiveness

Functional Correctness

Totality

Termination

Theorem Proving

Expressiveness

Functional Correctness

Totality

Termination

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?

Refinement Reflection

Theorem Proving

Refinement Reflection

Express **theorems** via refinement types.

Express **proofs** via functions.

Check that functions prove theorems.

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
```

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 []))`

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

Example: Parallel Sums

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

```haskell
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 }
```

Theorem Proving

Refinement Reflection

Express **theorems** via refinement types.

Express **proofs** via functions.

Check that functions prove theorems.

Expressiveness

Functional Correctness

Totality

Termination

Theorem Proving

Liquid Types

+

Functional Correctness

+

Totality

+

Termination

+

Theorem Proving

as a theorem prover

Case study: String matcher parallelization

200LoC "runtime" code

100LoC specs on "runtime" code

1300LoC proof terms

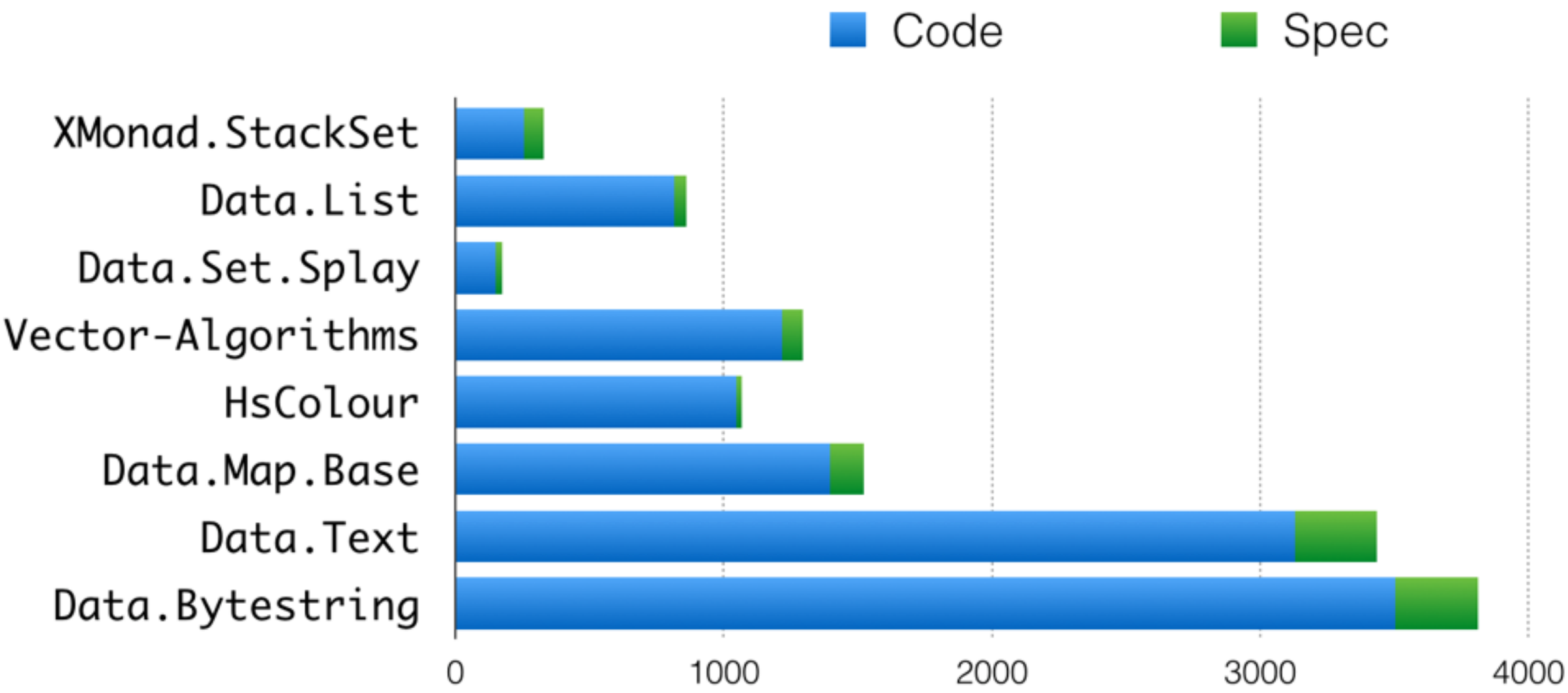
200LoC theorem specs

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

Abstract Refinements
Refinement Reflection

Modest Annotations

SMT Automation

Fast & Safe Code

Static Checks

Expressiveness

Abstract Refinements
Refinement Reflection

Modest Annotations

SMT Automation

Refinement Types in Scala

Refined

Scala Library for Refinement Types

LiquidTyper

Liquid Types in Dotty

Challenges

Mutability & Object Orientation

Interaction of Logic & Types

Fast & Safe Code

Static Checks

Expressiveness

Abstract Refinements
Refinement Reflection

Modest Annotations

SMT Automation

Thanks!

END