LiquidHaskell:
Liquid Types for Haskell

Niki Vazou
UC San Diego

Haskell

A purely-functional programming language.

Haskell is strongly typed

Haskell is strongly typed

```
λ 42 `div` True
Couldn't match expected type `Int'
with actual type `Bool’
```

```
div  :: Int -> Int -> Int
42   :: Int
True :: Bool
```

```
42 `div` True :: ???
```

Haskell

Haskell is strongly typed

“If it compiles, it works”

“Well typed programs cannot go wrong”

“Well typed programs can go wrong”

```
λ 42 `div` 0
* Exception: divide by zero
```

```
λ [2, 3, 1] !! 4
* Exception: index too large
```

```
λ sort [2, 3, 1]
  [2, 3, 1]
```

"Well typed programs can go wrong"

```
λ 42 `div` 0
* Exception: divide by zero
```

Haskell Types Reason About **Structure** of Values

`True :: Bool` *V.S.* `0 :: Int`

We want to Reason About **Semantics** of Values

`0` *V.S.* `2`

Liquid Types

```
{v:Int | 0 < v }
```

Haskell Type

Predicate*

* From a decidable language

Liquid Types

```
div :: Int -> {v:Int |0 < v} -> Int
```

Haskell Type

Predicate*

* From a decidable language

"Well typed programs can go wrong"

```
div :: Int -> {v:Int | 0 < v} -> Int
```

Haskell Types Reason About **Structure** of Values

`True :: Bool` *V.S.* `0 :: Int`

Refinement Types Reason About **Semantics** of Values

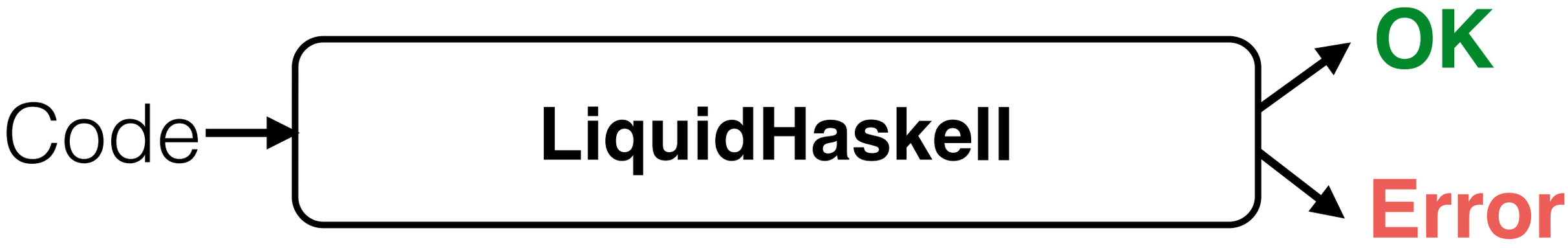
`0 :: {v:Int | v == 0}` *V.S.* `2 :: {v:Int | 0 < v}`

1 Motivation

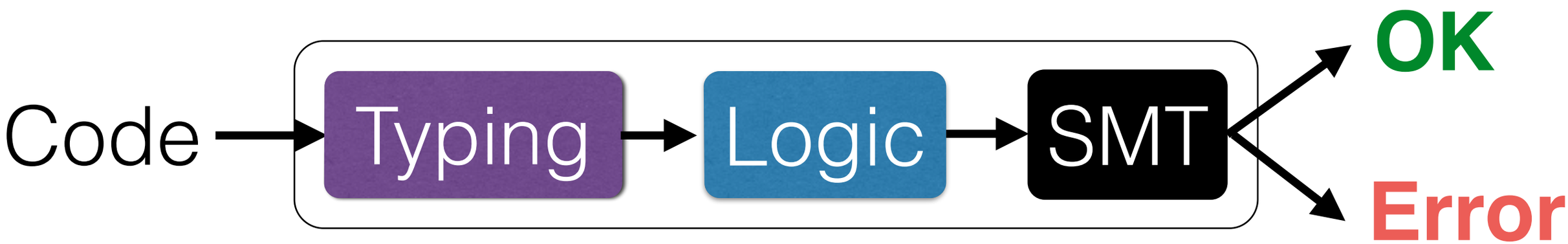
How to reason about semantics of values?

2 Liquid Typing 101

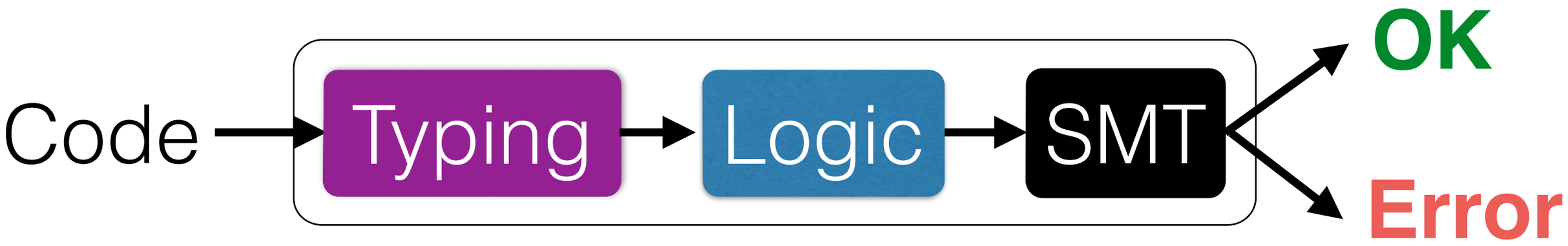
Liquid Typing 101



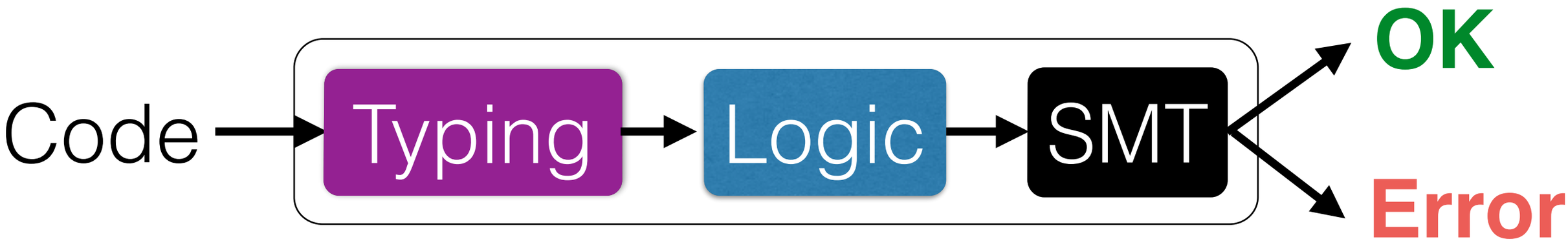
Liquid Typing 101



Liquid Typing 101



1. Source Code to **Typing constraints**
2. **Typing Constraints** to **Logical VC**
3. Check **VC validity** with **SMT Solver**


Code
Typing
Logic
SMT
OK
Error

Code → Typing

```
div    :: Int -> {v|0<v} -> Int

ex x = let y = 0
       in  x `div` y
```

```
y:{v|v=0} |- {v|v=0} <: {v|0<v}
```

Code → Typing

```
div    :: Int -> {v|0<v} -> Int

ex x = let y = 0
       in  x `div` y
```

```
y:{v|v=0} |- {v|v=0} <: {v|0<v}
```

Code → Typing

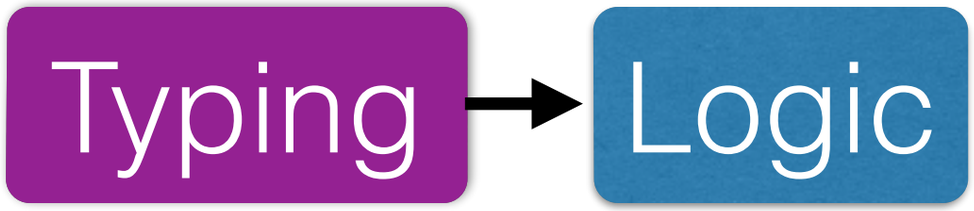
```
div    :: Int -> {v|0<v} -> Int

ex x = let y = 0
       in  x `div` y
```

y:{v|v=0} |- {v|v=0} <: {v|0<v}

Code → Typing

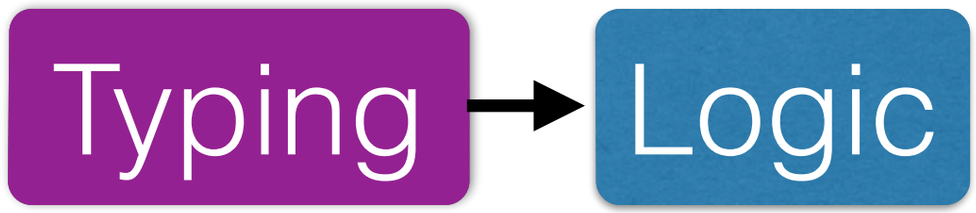
```
div    :: Int -> {v|0<v} -> Int

ex x = let y = 0
       in  x `div` y
```

```
y:{v|v=0} |- {v|v=0} <: {v|0<v}
```



Encode Subtyping as Logical VC

If VC valid then Subtyping holds



Encode Subtyping ...

```
y:{v|v=0} |- {v|v=0} <: {v|v>0}
```

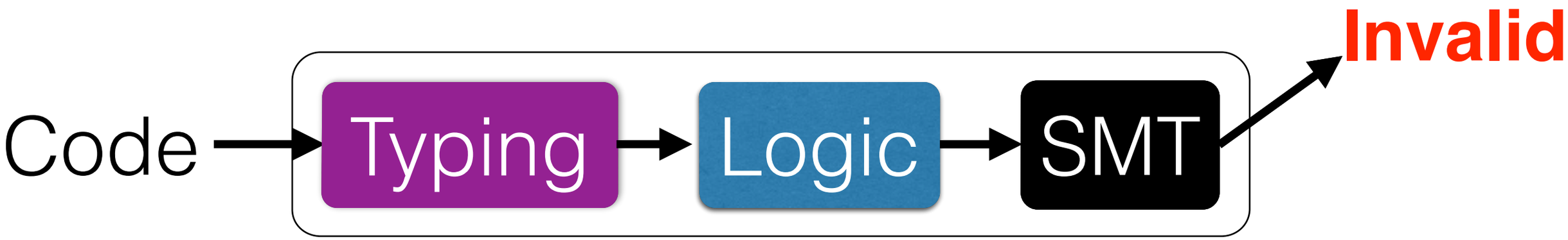
Typing → Logic

Encode Subtyping …

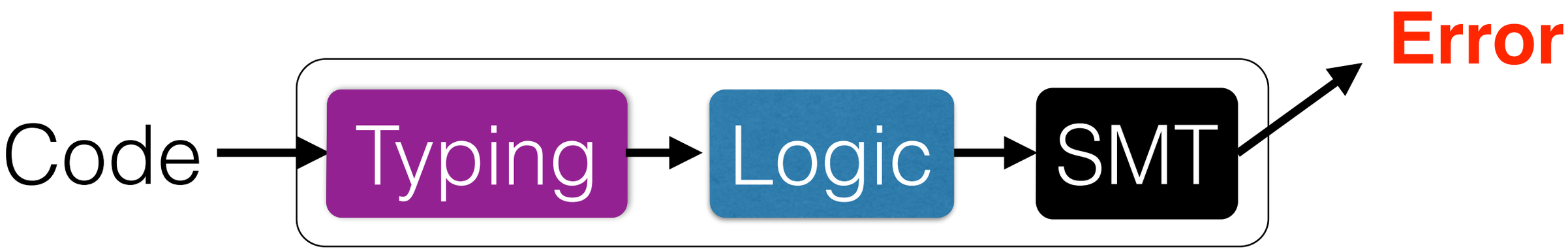
```
y:{v|v=0} |- {v|v=0} <: {v|v>0}
```

… as Logical VC

```
y=0  =>  v=0  =>  v>0
```




```
y=0   =>   v=0   =>   v>0
```




```
div    :: Int -> {v|0<v} -> Int

ex x = let y = 0
       in  x `div` y
```

Reason about (linear) arithmetic

How to reason about data structures?

1 Motivation

How to reason about semantics of values?

2 Liquid Typing 101

How to reason about data structures?

How to reason about data structures?

```
!! :: xs:[a]-> i:Int -> a

(x:_) !!0 = x
(_:xs)!!n = xs!!(n-1)
_     !!_ = error "index too large"
```

For the error not to occur, we want the index `i` to be `0 <= i && i < len x`.

How to talk about list length in logic?

How to talk about list length in logic?

```
measure len

len          :: [a] -> Int
len (_:xs) = 1 + len xs
len []     = 0
```

LiquidHaskell **automatically strengthens** the types of data constructors

```
data [a] where
  []  :: {v:[a]|len v = 0}
  (:) :: x:a -> xs:[a] -> {v:[a]|len v = 1 + len xs}
```

How to talk about list length in logic?

```
!! :: xs:[a]-> {v|0<=v && v< len xs } -> a

(x:_) !!0 = x
(_:xs)!!n = xs!!(n-1)
_     !!_ = error "index too large"
```

Now, the error line is **provably** dead code

Another measure on lists

```
measure hasZero

hasZero          :: [Int] -> Int
hasZero (x:xs) = x == 0 || hasZero xs
hasZero []     = False
```

LiquidHaskell **automatically strengthens** the types of data constructors

```
data [a] where
  []  :: {v:[a]|not (hasZero v)}
  (:) :: x:a -> xs:[a]
      -> {v:[a]|hasZero v <=> x == 0 || hasZero xs}
```

Multiple measures on lists

```
data [a] where
  []  :: {v:[a]|len v = 0}
  (:) :: x:a -> xs:[a] -> {v:[a]|len v = len xs + 1}
```

+

```
data [a] where
  []  :: {v:[a]|not (hasZero v)}
  (:) :: x:a -> xs:[a]
      -> {v:[a]|hasZero v <=> x == 0 || hasZero xs}
```

=

```
data [a] where
  []  :: {v:[a]|not (hasZero v) && len v = 0}
  (:) :: x:a -> xs:[a]
      -> {v:[a]|hasZero v <=> x == 0 || hasZero xs
                && len v = len xs + 1                }
```

Refine List Data Constructors

```
data [a] =
  []  :: [a]
  (:) :: x:a -> xs:[{v:a|x <= v}] -> [a]
```

LiquidHaskell

- proves *invariant* any time `(:)` is used
- assumes *invariant* any time `(:)` is opened

Specify Increasing Lists

Specify Increasing Lists

```
data [a] =
  []  :: [a]
  (:) :: x:a -> xs:[{v:a|x <= v}] -> [a]
```

```
insert :: Ord a => a -> [a]->[a]

insert y []     = [y]
insert y (x:xs) | x <= y    = x:insert y xs
                | otherwise = y:x:xs
```

Can Increasing and Simple Lists Coexist?

1 Motivation
How to reason about semantics of values?

2 Liquid Typing 101
How to reason about data structures?

3 Measures
Can Increasing and Simple Lists Coexist?

Can Increasing and Simple Lists Coexist?

```
data [a]<p :: a -> a -> Bool> =
  []   :: [a]
  (:) :: x:a -> xs:[{v:a|  p v }] -> [a]<p>
```

Increasing Lists

```
p |-> \x v-> x<=v
```

```
type Incr a = [a] <\x v -> x <= v>
```

Simple Lists

```
p |-> \x v-> true
```

```
type List a = [a] <\x v -> true> = [a]
```

Can Increasing and Simple Lists Coexist?

```
sort = mergeAll . sequences
 where
  sequences (a:b:xs)
    | b < a          = descending b [a] xs
  sequences [x]      = [[x]]
  sequences []       = [[]]

  descending a as (b:bs)
    | b < a          = descending b (a:as) bs
  descending a as bs = (a:as): sequences bs


ascending a as (b:bs)
    | a <= b         = ascending b (\ys -> as (a:ys)) bs
  ascending a as bs  = as [a]: sequences bs
```

```
sort :: Ord a => [a] -> Incr a
```

Can Increasing and Simple Lists Coexist?

```
sort :: Ord a => [a] -> Incr a
sort = mergeAll . sequences
 where
  sequences (a:b:xs)
    | b < a           = descending b [a]  xs
    | otherwise       = ascending  b (a:) xs
  sequences [x]       = [[x]]
  sequences []        = [[]]

descending :: x:a -> Incr {v|x<v} -> [a] ->  [Incr a]
 descending a as (b:bs)
   | b < a            = descending b (a:as) bs
 descending a as bs = (a:as): sequences bs

ascending :: x:a -> (Incr {v|x<=v} -> Incr a) -> [a] -> [Incr a]
ascending a as (b:bs)
   | a <= b           = ascending b (\ys -> as (a:ys)) bs
 ascending a as bs  = as [a]: sequences bs
```

Can Increasing and Simple Lists Coexist?

We used **Abstract Refinements** in Types to Decouple code from Specifications

We will use **Abstract Refinements** in functions to Abstract over Properties

Example: Encode Natural Induction

Encoding Natural Induction

```
loop :: Int -> a -> (Int -> a -> a) -> a

loop hi base f = go 0 base
 where
  go i acc | i < hi     = go (i + 1) (f i acc)
           | otherwise = acc
```

What is a valid `loop` type that proves `add`?

```
add :: n:Int -> m:Int -> {v:Int | v = n + m }

add n m = loop n m (\_ z -> z + 1)
```

Encoding Natural Induction

Type is too specific

```
loop ::  n:{v | 0 <= v}
      -> m:Int
      ->(i:Int -> {v | v=m+i} -> {v | v=m+(i+1)})
      -> {v | v=m+n}
```

```
add :: n:Int -> m:Int -> {v:Int | v = n + m }
add n m = loop n m (\_ z -> z + 1)
```

Encoding Natural Induction

Type is too specific

```
loop :: n:{v | 0 <= v}
     -> m:Int
     ->(i:Int -> {v | v=m+i} -> {v | v=m+(i+1)})
     -> {v | v=m+n}
```

```
add :: n:Int -> m:Int -> {v:Int | v = n + m }
add n m = loop n m (\_ z -> z - 1)
```

Fail

Encoding Natural Induction

```
p |-> \i v-> v=m+i
```

```
loop ::  n:{v|0 <= v}
      -> m:{v|p 0 v}
      ->(i:Int->{v| p i v}->{v| p (i+1) n})
      -> {v| p i n}
```

```
add :: n:Int -> m:Int -> {v:Int | v = n + m }
add n m = loop n m (\_ z -> z + 1)
```

Encoding Natural Induction

Base Case

```
loop ::  n:{v | 0 <= v}
      -> m:{v | p 0 v}
      ->(i:Int->{v | p i v}->{v | p (i+1) n})
      -> {v | p i n}
```

By induction

Inductive Step

Abstract Refinements

Decouple code from Specifications in Types

Abstract over Properties in Functions

What are the limitations?

Can we express function composition?

1 Motivation

How to reason about semantics of values?

2 Liquid Typing 101

How to reason about data structures?

3 Measures

Can Increasing and Simple Lists Coexist?

4 Abstract Refinements

Can we specify function composition?

Can we specify function composition?

```
incr    :: x:Int -> {v| v = x+1}
incr x = x+1
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

Can we specify function composition?

Type is too specific

```
compose ::
          (y:b-> {z:c|z = y+1})
       -> (x:a-> {z:b|y = x+1})
       -> (x:a-> {z:c|z = x+2})
compose f g x = f (g x)
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

OK

Can we specify function composition?

Type is too specific

```
compose ::
          (y:b-> {z:c|z = y+1})
       -> (x:a-> {z:b|y = x+1})
       -> (x:a-> {z:c|z = x+2})
compose f g x = f (g x)
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose decr incr
```

Fail

Can we specify function composition?

Type is too specific

```
compose ::
          (y:b-> {z:c|z = y+1})
       -> (x:a-> {z:b|y = x+1})
       -> (x:a-> {z:c|z = x+2})
compose f g x = f (g x)
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

Can we specify function composition?

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = f (g x)
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

Can we specify function composition?

Type is wrong

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

Can we specify function composition?

Type is wrong

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

```
y:{y:b|q x y} ⊢ {z:c|p y z} <: {z:c|r x z}
```

Can we specify function composition?

Type is wrong

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

$$y{:}\{y{:}b \mid q\ x\ y\} \vdash \{z{:}c \mid p\ y\ z\} <: \{z{:}c \mid r\ x\ z\}$$

Can we specify function composition?

Type is wrong

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

$$y{:}\{y{:}b \mid q\ x\ y\} \vdash \{z{:}c \mid p\ y\ z\} <: \{z{:}c \mid r\ x\ z\}$$

Can we specify function composition?

Type is wrong

```
compose ::
          => (y:b-> {z:c|p y z})
          -> (x:a-> {y:b|q x y})
          -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

```
bound Chain p q r = \x y z ->
   q x y => p y z => r x z
```

Can we specify function composition?

```
compose :: (Chain p q r)
        => (y:b-> {z:c|p y z})
        -> (x:a-> {y:b|q x y})
        -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

OK

```
bound Chain p q r = \x y z ->
   q x y => p y z => r x z
```

Can we specify function composition?

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

```
bound Chain p q r = \x y z ->
   q x y => p y z => r x z
```

Can we specify function composition?

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

OK

```
bound Chain = \x y z ->
  y=x+1 => z=y+1 => z=x+2
```

Valid

Can we specify function composition?

Bounds let us specify function composition

Do bounds add complexity?

No. Bounds are desugared to unbounded types

Bounds are desugared to unbounded types

```
compose :: (Chain p q r)
        => (y:b-> {z:c|p y z})
        -> (x:a-> {y:b|q x y})
        -> (x:a-> {z:c|r x z})
compose f g x =
  let y = g x in
  let z = f y in z
```

```
bound Chain p q r = \x y z ->
   q x y => p y z => r x z
```

Bounds are desugared to unbounded types

```
compose :: $chain:(tchain p q r)
        -> (y:b-> {z:c|p y z})
        -> (x:a-> {y:b|q x y})
        -> (x:a-> {z:c|r x z})
compose f g x =
  let y = g x in
  let z = f y in z
```

```
bound Chain p q r = \x y z ->
   q x y => p y z => r x z
```

Bounds are desugared to unbounded types

```
compose :: $chain:(tchain p q r)
        -> (y:b-> {z:c|p y z})
        -> (x:a-> {y:b|q x y})
        -> (x:a-> {z:c|r x z})
compose $chain f g x =
  let y = g x in
  let z = f y in
  let _ = $chain x y z in z
```

OK

```
type tchain p q r = x:a -> y:b -> z:c ->
   {v|q x y => p y z => r x z}
```

Bounds are desugared to unbounded types

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose $chain incr incr
 where $chain :: tchain p q r)
       $chain =   ???
```

```
type tchain p q r = x:a -> y:b -> z:c ->
  {v|q x y => p y z => r x z}
```

Bounds are desugared to unbounded types

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose $chain incr incr
 where $chain :: tchain
       $chain =   ???
```

```
type tchain     = x:a -> y:b -> z:c ->
  {v|y=x+1 => z=y+1 => z=x+2}
```

Bounds are desugared to unbounded types

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose $chain incr incr
 where $chain :: tchain
       $chain =   ???
```

```
type tchain     = x:a -> y:b -> z:c ->
  {v|true}
```

Bounds enhance expressiveness

Do bounds add complexity?

No. Bounds are desugared to unbounded types

Are bounds useful?

Function Composition

List Filtering and List Folding

Floyd-Hoare Logic in the State monad

Relational DataBases

LiquidHaskell

1 Motivation
How to reason about semantics of values?

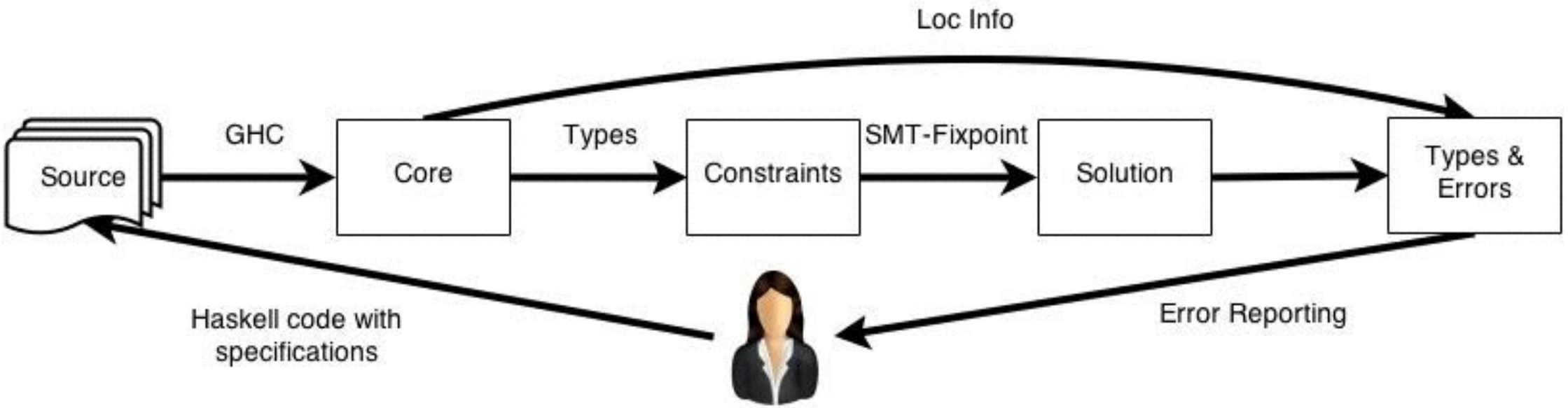
2 Liquid Typing 101
How to reason about data structures?

3 Measures
Can Increasing and Simple Lists Coexist?

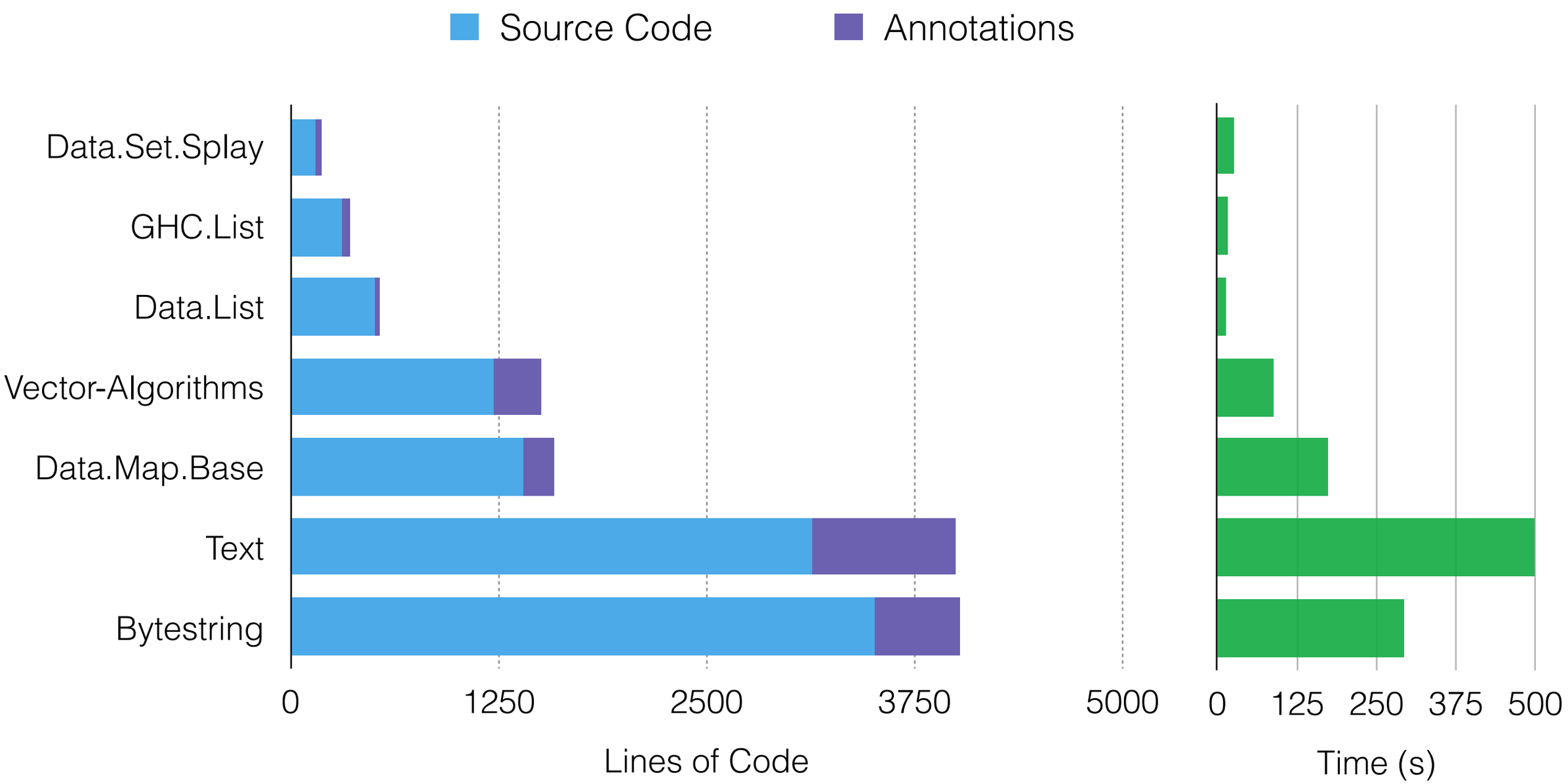
4 Abstract Refinements
Can we specify function composition?

5 Bounded Refinements

LiquidHaskell



LiquidHaskell



LiquidHaskell publications

Abstract Refinement Types (ESOP ‘13)
N. Vazou, P. Rondon, and R. Jhala

LiquidHaskell: Experience with Refinement Types in the Real World (Haskell ‘14)
N. Vazou, E. Seidel, and R. Jhala

Refinement Types for Haskell (ICFP ‘14)
N. Vazou, E. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones

Type Targeting Testing (ESOP ‘15)
E. Seidel, N. Vazou, and R. Jhala

Bounded Refinement Types (ICFP ‘15)
N. Vazou, A. Bakst, and R. Jhala

LiquidHaskell what is next?

1. Increase Expressiveness

Multi-parameter Measures

Single-parameter measures

```
measure hasZero

hasZero          :: [Int] -> Int
hasZero (x:xs) = x == 0 || hasZero xs
hasZero []     = False
```

LiquidHaskell **automatically strengthens** the types of data constructors

```
data [a] where
  []  :: {v:[a]|not (hasZero v)}
  (:) :: x:a -> xs:[a]
      -> {v:[a]|hasZero v <=> x == 0 || hasZero xs}
```

Mutli-parameter measures

```
measure hasElem

hasElem             :: a -> [a] -> a
hasElem y (x:xs) = x == y || hasZero xs
hasElem y []     = False
```

LiquidHaskell should **strengthen** the types of data constructors

```
data [a] where
  []  :: forall (y :: a). {v:[a]|not (hasElem y v)}
  (:) :: forall (y :: a). x:a -> xs:[a]
      -> {v:[a]|hasZero v <=> x == y || hasZero xs}
```

Mutli-parameter measures

Open Questions

- What is the semantics of forall (y :: a). t?
  - What should y be instantiated with?
  - Inference at type level or at value level

- Can we eliminate multi-parameter measures before logic,
  and thus stay decidable?

- Which argument should we refine?
  reverse :: [a] -> [a] -> [a]

LiquidHaskell what is next?

1. Increase Expressiveness

Multi-parameter Measures

2. Into the real world

Haxl*

Haxl is a Haskell library that simplifies *access to remote data*, such as databases or web-based services.

Haxl can *automatically*

- batch multiple requests to the same data source,

- request data from multiple data sources concurrently,

- cache previous requests.

*Simon Marlow et al. There is no Fork: an Abstraction for Efficient,Concurrent, and Concise Data Access (ICFP ’14)

The Haxl Monad

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

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

Concurrency comes for free by the (library) instance of Applicative

```
instance Applicative Haxl where
 - library implementation that makes
 - Haxl concurrent
```

Creating Haxl Requests

Haxl

```
dataFetch :: (DataSource r) => r a -> Haxl a
```

User

```
data UserReq a where
 GetFriendIds :: Id -> UserReq [Id]
 GetLocation  :: Id -> UserReq Location
```

```
instance DataSource UserReq where
 – how to actually fetch data from the data base
```

Creating Haxl Requests

Haxl

```
dataFetch :: (DataSource r) => r a -> Haxl a
```

User

```
data UserReq a where
 GetFriendIds :: Id -> UserReq [Id]
 GetLocation  :: Id -> UserReq Location
```

```
getUserLocation  :: Id -> Haxl Location
getUserLocation  = dataFetch . GetLocation

getFriendIdsById :: Id -> Haxl [Id]
getFriendIdsById = dataFetch . GetFriendIds
```

Combining Haxl Requests

```
getFriendsLocations :: Id -> Haxl [(Id, Location)]
getFriendsLocations x = do
 userIds <- getFriendIdsById x
 locs    <- for userIds getUserLocation
 return   $ zip userIds locs
```

Concurrent Execution!

```
getUserLocation :: Id -> Haxl Location
getUserLocation = dataFetch . GetLocation

getFriendIdsById :: Id -> Haxl [Id]
getFriendIdsById = dataFetch . GetFriendIds
```

Running Haxl Requests

Haxl

```
runHaxl :: Env       -- the reader environment
        -> Haxl a    -- the Haxl computation
        -> IO a      -- the fetch data
```

Haxl: Putting it all together

User

```
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
  locs1 <- runHaxl env (getHerFiendsLocations myid )
  putStrLn $ "My friend's Locs"  ++ show locs1

  -- fetch the locations of HER friends
  locs2 <- runHaxl env (getHerFiendsLocations herid)
  putStrLn $ "Her friend's Locs" ++ show locs2
```

Proposal: Use liquidHaskell to check Haxl policies

P1: “Anyone can look at my friend list”

forall x y. hasCredentials y (cfriends x)

P2: “Only my friends can look at my location”

forall x y. isFriend x y => hasCredentials x (cloc y)

Vocabulary

```
isFriend :: Id -> Id -> Bool
cfriends :: Id -> Credential
cloc     :: Id -> Credential
hasCredentials :: Id -> Credential -> Bool
```

Getting locations of her friends fails

```
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                    OK
 locs1 <- runHaxl env (getHerFiendsLocations myid )
 putStrLn $ "My friend's Locs"  ++ show locs1

 – fetch the locations of HER friends                   Error
 locs2 <- runHaxl env (getHerFiendsLocations herid)
 putStrLn $ "Her friend's Locs" ++ show locs2
```

Refine Constructors with credential info

```
data UserReq a where
 GetFriendIds :: x:Id
              -> {req:UserReq [{v:Id | isFriend x v}]
                  | creq req = cfriends x}

 GetLocation  :: x:Id
              -> {req:UserReq Location
                  | creq req = cloc x}
```

Vocabulary

```
isFriend :: Id -> Id -> Bool
creq     :: UserReq a -> Credential
cfriends :: Id -> Credential
cloc     :: Id -> Credential
```

Refine Constructors with credential info

```
dataFetch :: (DataSource r)
          => req:r a
          -> {haxl:Haxl a | creq req = chaxl haxl }
```

Vocabulary

```
isFriend :: Id -> Id -> Bool
creq     :: UserReq a -> Credential
cfriends :: Id -> Credential
cloc     :: Id -> Credential
chaxl    :: Haxl a -> Credential
```

Creating Haxl Requests

```
getUserLocation
 :: x:Id -> { haxl: Haxl a [{v:Id | isFriend x v}]
            | chaxl haxl = cfriends x}
getUserLocation  = dataFetch . GetLocation

getFriendIdsById
 :: x:Id -> {haxl: Haxl a
            | chaxl haxl = cloc x}
getFriendIdsById = dataFetch . GetFriendIds
```

Vocabulary

```
isFriend :: Id -> Id -> Bool
creq     :: UserReq a -> Credential
cfriends :: Id -> Credential
cloc     :: Id -> Credential
chaxl    :: Haxl a -> Credential
```

Combining Haxl Credentials

```
getFriendsLocations
 :: x:Id -> {haxl:Haxl [(Id, Location)]| ???}

getFriendsLocations x = do
 userIds <- getFriendIdsById x
 locs    <- for userIds getUserLocation
 return  $ zip userIds locs
```

Combining Haxl Credentials

```
getFriendsLocations
 :: x:Id -> {haxl:Haxl [(Id, Location)]| ???}

getFriendsLocations x = do
 userIds <- getFriendIdsById x
 locs    <- for userIds getUserLocation
 return  $ zip userIds locs
```

chaxl haxl = cfriends x

userIds :: [{v:Id | isFriend x v}]

forall x y. hasCredentials y (cfriends x)

forall x y. isFriend x y => hasCredentials x (cloc

hasCredentials x (chaxl haxl)

Combining Haxl Credentials

```
getFriendsLocations
 :: x:Id -> {haxl:Haxl [(Id, Location)]| ???}


getFriendsLocations x = do
 userIds <- getFriendIdsById x
 locs    <- for userIds getUserLocation
 return  $ zip userIds locs
```

chaxl haxl = cfriends x

userIds :: [{v:Id | isFriend x v}]

forall x y

forall x y. isFriend x y => hasCredentials x (cloc y)

hasCredentials x (chaxl haxl)

Combining Haxl Credentials

```
getFriendsLocations
 :: x:Id -> {haxl:Haxl [(Id, Location)]| ???}


getFriendsLocations x = do
 userIds <- getFriendIdsById x
 locs    <- for userIds getUserLocation
 return  $ zip userIds locs
```

chaxl haxl = cfriends x

userIds :: [{v:Id | isFriend x v}]

forall x y

forall x y. isFriend x y => hasCredentials x (cloc

hasCredentials x (chaxl haxl)

Combining Haxl Credentials

```
getFriendsLocations
 :: x:Id -> {haxl:Haxl [(Id, Location)]|
             hasCredentials x (chaxl haxl) }
getFriendsLocations x = do
 userIds <- getFriendIdsById x
 locs    <- for userIds getUserLocation
 return  $ zip userIds locs
```

```
chaxl haxl = cfriends x

userIds :: [{v:Id | isFriend x v}]

forall x y

forall x y. isFriend x y => hasCredentials x (cloc
```

hasCredentials x (chaxl haxl)

Environment Upper Bounds Credentials

```
runHaxl :: env:Env
        -> {haxl:Haxl a
           | hasCredentials (envId env) haxl}
        -> IO a
```

Haxl: Putting it all together

```
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
  locs1 <- runHaxl env (getHerFiendsLocations myid )
  putStrLn $ "My friend's Locs"  ++ show locs1

  -- fetch the locations of HER friends
  locs2 <- runHaxl env (getHerFiendsLocations herid)
  putStrLn $ "Her friend's Locs" ++ show locs2
```

Haxl: Putting it all together

```
env :: {v:Env | envId env = myid }
```

```
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
  locs1 <- runHaxl env (getHerFiendsLocations myid )
  putStrLn $ "My friend's Locs"  ++ show locs1

  -- fetch the locations of HER friends
  locs2 <- runHaxl env (getHerFiendsLocations herid)
  putStrLn $ "Her friend's Locs" ++ show locs2
```

Haxl: Putting it all together

```
env :: {v:Env | envId env = myid }
```

```
{haxl:Haxl a | hasCredentials myid haxl} -> IO a
```

```
{haxl:Haxl a | hasCredentials myid  haxl}
```

```
locs1 <- runHaxl env (getHerFiendsLocations myid )
```

OK

Haxl: Putting it all together

```
main :: IO ()
main = do
  -- me and her are two random users
  let (myid, herid) = (42, 24)

  -- initialize the environment with my credentials
  env <- initEnv (stateStore :> Nil) ()

  -- fetch the locations of MY friends
  locs1 <- runHaxl env (getMyFriendsLocations myid)
  putStrLn $ "My friend's Locs" ++ show locs1

  -- fetch the locations of HER friends
  locs2 <- runHaxl env (getHerFiendsLocations herid)
  putStrLn $ "Her friend's Locs" ++ show locs2
```

```
env :: {v:Env | envId env = myid }
```

```
{haxl:Haxl a | hasCredentials myid haxl} -> IO a
```

```
{haxl:Haxl a | hasCredentials herid haxl}
```

Error

LiquidHaskell what is next?

1. Increase Expressiveness

Multi-parameter Measures

2. Into the real world

Security Verification of Data Fetching via Haxl

LiquidHaskell

Liquid Type Checker for Haskell Code

Specifications language is decidable
but it is expressive due to

Abstract and **Bounded Refinement Types**

Future Work:

More Expressive (Mutli-parameter Measures)

Verify Security Properties (Haxl)

Thanks!