Abstract Refinement Types

Niki Vazou[1], Patrick M. Rondon[2], and Ranjit Jhala[1]
[1]UC San Diego [2]Google

Vanilla Types

12 :: **Int**

Refinement Types

$$12 :: \{v{:}\ \mathbf{Int} \mid v > 10\}$$

Refinement Types

$$12 :: \{v : \mathbf{Int} \mid v = 12\}$$

Refinement Types

$$v = 12 \Rightarrow 0 < v < 20 \wedge \text{even } v$$

$$12 :: \{v\text{: } \mathbf{Int} \mid v = 12\}$$

Refinement Types

$$v = 12 \Rightarrow 0 < v < 20 \land \text{even } v$$

$$12 :: \{v\text{: }\mathbf{Int} \mid v = 12\}$$

$$<: \{v\text{: }\mathbf{Int} \mid 0 < v < 20 \land \text{even } v\}$$

Refinement Types

$12 :: \{v : \mathbf{Int} \mid v = 12\}$

$12 :: \{v : \mathbf{Int} \mid 0 < v < 20 \land \text{even } v\}$

A max function

```haskell
max :: Int -> Int -> Int

max x y = if x > y then x else y
```

A refined max function

```
max::x:Int-> y:Int -> {v:Int | v ≥ x ∧ v ≥ y}
max x y = if x > y then x else y
```

Using max

```
max::x:Int-> y:Int -> {v:Int | v ≥ x ∧ v ≥ y}
max x y = if x > y then x else y
```

```
b = max 8 12                  -- assert (b > 0)
```

max 8 12 :: { v : **Int** | v ≥ x ∧ v ≥ y }[8/x][12/y]

Using max

```
max::x:Int-> y:Int -> {v:Int | v ≥ x ∧ v ≥ y}
max x y = if x > y then x else y
```

```
b = max 8 12                    -- assert (b > 0)
```

max 8 12 :: { v : **Int** | v ≥ 8 ∧ v ≥ 12 }

Using max

```
max::x:Int-> y:Int -> {v:Int | v ≥ x ∧ v ≥ y}
max x y = if x > y then x else y
```

```
b = max 8 12                -- assert (b > 0)
```

max 8 12 :: { v : **Int** | v ≥ 12 }

Using max

```
max::x:Int-> y:Int -> {v:Int | v ≥ x ∧ v ≥ y}
max x y = if x > y then x else y
```

```
b = max 8 12              -- assert (b > 0)
```

$v \geq 12 \Rightarrow v > 0$

max 8 12 :: $\{ v : \mathbf{Int} \mid v \geq 12 \}$ <: $\{ v : \mathbf{Int} \mid v > 0 \}$

Using max

```
max::x:Int-> y:Int -> {v:Int | v ≥ x ∧ v ≥ y}
max x y = if x > y then x else y
```

```
b = max 8 12                  -- assert (b > 0)
```

max 8 12 :: { v : **Int** | v > 0 }

Using max

```
max::x:Int-> y:Int -> {v:Int | v ≥ x ∧ v ≥ y}
max x y = if x > y then x else y
```

```
b = max 8 12                  -- assert (b > 0)✓
```

max 8 12 :: { v : **Int** | v > 0 }

Using max

```
max::x:Int-> y:Int -> {v:Int | v ≥ x ∧ v ≥ y}
max x y = if x > y then x else y
```

```
b = max 8 12            -- assert (b > 0)✓
c = max 3  5            -- assert (odd c)
```

We **get**

max 3 5 :: { v : **Int** | v ≥ 5 }

We **want**

max 3 5 :: { v : **Int** | v ≥ 5 ∧ **odd** v }

Using max

Problem:

Information of Input Refinements is Lost

We **get**

max 3 5 :: { v : **Int** | v ≥ 5 }

We **want**

max 3 5 :: { v : **Int** | v ≥ 5 ∧ **odd** v }

Our Solution

Problem:

Information of Input Refinements is Lost

Solution:

Parameterize Type Over Input Refinement

Abstract Refinements

```
max::forall <p::Int -> Prop>.
          Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

Solution:

Parameterize Type Over Input Refinement

Abstract Refinements

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

Abstract refinement

“if **both** arguments satisfy **p**,
then the **result** satisfies **p**”

Abstract Refinements

Abstract refinement

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

"if **both** arguments satisfy **p**,
then the **result** satisfies **p**"

Abstract Refinements

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

Abstract refinement

“if **both** arguments satisfy **p**,
then the **result** satisfies **p**”

Abstract Refinements

```
max::forall <p::Int -> Prop>.
          Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

Abstract refinement

“if **both** arguments satisfy **p**,
then the **result** satisfies **p**”

Using max

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

```
b = max [(>0)] 8 12 -- assert (b > 0)✓
c = max [odd]  3  5 -- assert (odd c)
```

max :: forall <p :: **Int** -> **Prop**>.

Int<p> -> **Int**<p> -> **Int**<p>

Using max

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

```
b = max [(>0)] 8 12 -- assert (b > 0)✓
c = max [odd]  3  5 -- assert (odd c)
```

max :: forall <p :: **Int** -> **Prop**>.

Int<p> -> **Int**<p> -> **Int**<p>

Using max

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

```
b = max [(>0)] 8 12 -- assert (b > 0)✓
c = max [odd]  3  5 -- assert (odd c)
```

max [odd] ::

Int<p> -> **Int**<p> -> **Int**<p> [odd/p]

Using max

```
max::forall <p::Int -> Prop>.
              Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

```
b = max [(>0)] 8 12 -- assert (b > 0)✓
c = max [odd]  3  5 -- assert (odd c)
```

max [odd] ::

{v:Int | odd v} -> {v:Int | odd v } -> {v:Int | odd v}

Using max

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

```
b = max [(>0)] 8 12 -- assert (b > 0)✓
c = max [odd]  3  5 -- assert (odd c)
```

3 :: { v:Int | odd v }

max [odd] ::

{v:Int | odd v} -> {v:Int | odd v } -> {v:Int | odd v}

Using max

```
max::forall <p::Int -> Prop>.
          Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

```
b = max [(>0)] 8 12 -- assert (b > 0)✓
c = max [odd] 3 5 -- assert (odd c)
```

3 :: { v:**Int** | odd v }

max [odd] 3 ::

{v:**Int** | odd v } -> {v:**Int** | odd v}

Using max

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

```
b = max [(>0)] 8 12 -- assert (b > 0)✓
c = max [odd]  3  5 -- assert (odd c)
```

5 :: { v:Int | odd v }

max [odd] 3 ::

{v:Int | odd v } -> {v:Int | odd v}

Using max

```
max::forall <p::Int -> Prop>.
              Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

```
b = max [(>0)] 8 12 -- assert (b > 0)✓
c = max [odd]  3  5 -- assert (odd c)
```

5 :: { v:**Int** | odd v }

max [odd] 3 5 ::

{v:**Int** | odd v}

Using max

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

```
b = max [(>0)] 8 12 -- assert (b > 0)✓
c = max [odd]  3  5 -- assert (odd c)✓
```

max [odd] 3 5 ::

{v:**Int** | odd v}

Abstract Refinements

Abstract refinement

```
max::forall <p::Int -> Prop>.
            Int<p> -> Int<p> -> Int<p>

max x y = if x > y then x else y
```

“if **both** arguments satisfy **p**, then the **result** satisfies **p**”

Outline

Introduction

Monomorphic Refinements

Applications

Refinements and Type Classes

Inductive Refinements

Indexed Refinements

Recursive Refinements

Evaluation

Outline

Introduction

- Monomorphic Refinements

Applications

- Refinements and Type Classes
- Inductive Refinements
- Indexed Refinements
- Recursive Refinements

Evaluation

Outline

Introduction

Monomorphic Refinements

Applications

Refinements and Type Classes

Inductive Refinements

Indexed Refinements

Recursive Refinements

Evaluation

A polymorphic max function

```
max :: a -> a -> a
max x y = if x > y then x else y
```

```
c = max 3 5          -- assert (odd c) ✓
```

We **instantiate**

```
a := { v:Int | odd v }
```

We **get**

```
max [{v:Int | odd v}] 3 5 ::{v:Int | odd v}
```

Type Class Constraints

```
max :: Ord a => a -> a -> a
max x y = if x > y then x else y
```

```
c = max 3 5            -- assert (odd c) ✓
```

We **instantiate**

```
a := { v:Int | odd v }
```

We **get**

```
max [{v:Int | odd v}] 3 5 ::{v:Int | odd v}
```

Unsound Reasoning

```
minus :: Num a => a -> a -> a
minus x y = x - y
```

```
b = max   3 5        -- assert (odd b)✓
c = minus 3 5        -- assert (odd c)
```

We **instantiate**

```
a := { v:Int | odd v }
```

We **get**

```
minus [{v:Int|odd v}] 3 5 :: {v:Int | odd v }
```

Abstract Refinements and Type Classes

```
max ::forall <p::a -> Prop>.
          Ord a => a<p> -> a<p> -> a<p>
max x y = if x > y then x else y
```

```
b = max [Int] [odd] 3 5-- assert (odd b)✓
```

We **get**

```
max [Int] [odd] 3 5 :: {v:Int | odd v}
```

Abstract Refinements and Type Classes

```
minus :: Num a => a -> a -> a
minus x y = x - y
```

```
b = max [Int] [odd] 3 5  -- assert (odd b)✓
c = minus [Int] 3 5      -- assert (odd b)✗
```

We **get**

```
minus [Int] 3 5 :: Int
```

Outline

Introduction

- Monomorphic Refinements

Applications

- **Refinements and Type Classes**
- Inductive Refinements
- Indexed Refinements
- Recursive Refinements

Evaluation

Outline

A loop function

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
 where go i acc | i < n = go (i+1) (f i acc)
                | otherwise = acc
```

A loop function

```
loop :: (Int -> a -> a) -> I          a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

next acc

loop iteration

final result

A loop function

initial index

initial acc

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

``loop f n z = $f^n(z)$''

A loop function

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

``loop f n z = $f^n(z)$''

A loop function

```haskell
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
 where go i acc | i < n = go (i+1) (f i acc)
                | otherwise = acc
```

```haskell
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

A loop function

```haskell
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

```haskell
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

incr acc by 1

A loop function

```haskell
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

```haskell
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

incr acc by 1

Question: Does ``**incr** n z = n+z`` hold?

Answer: Proof by Induction

Inductive Proof

```haskell
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

Loop Invariant: R :: (**Int**, a)

loop iteration

accumulator

Inductive Proof

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

Loop Invariant: R :: (**Int**, a)

Base: R(0, z)

Inductive Step: R(i, acc) ⇒ R(i+1, f i acc)

Conclusion: R(n, loop f n z)

Induction via Abstract Refinements

```haskell
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

$$\frac{\begin{array}{l} R :: (\mathbf{Int}, a) \\ R(0, z) \\ R(i, acc) \Rightarrow \\ \quad R(i+1, f\ i\ acc) \end{array}}{R(n, \text{loop } f\ n\ z)}$$

Induction via Abstract Refinements

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

R :: (**Int**, a)

`r :: Int -> a -> Prop`

R(0, z)

`z :: a<r 0>`

R(i, acc) ⇒
    R(i+1, f i acc)

R(n, loop f n z)

Induction via Abstract Refinements

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n     = go (i+1) (f i acc)
                 | otherwise = acc
```

R :: (**Int**, a)

```
r :: Int -> a -> Prop
```

$R(0, z)$

```
z :: a<r 0>
```

$R(i, acc) \Rightarrow$
$R(i+1, f\ i\ acc)$

```
f :: i:Int -> a<r i>
       -> a<r (i+1)>
```

$$\frac{}{R(n, \text{loop}\ f\ n\ z)}$$

Induction via Abstract Refinements

```haskell
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

R :: (**Int**, a)	`r :: Int -> a -> Prop`
R(0, z)	`z :: a<r 0>`
R(i, acc) ⇒ R(i+1, f i acc)	`f :: i:Int -> a<r i> -> a<r (i+1)>`
R(n, loop f n z)	`loop f n z :: a<r n>`

Induction via Abstract Refinements

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

```
r :: Int -> a -> Prop

z :: a<r 0>

f :: i:Int -> a<r i>
   -> a<r (i+1)>
--------------------------------
loop f n z :: a<r n>
```

Induction via Abstract Refinements

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n = go (i+1) (f i acc)
                 | otherwise = acc
```

```
loop
  :: forall <r :: Int -> a -> Prop>.
     f:(i:Int -> a<r i> -> a<r (i+1)>)
  -> n:{ v:Int | v>=0 }
  -> z:a<r 0>
  -> a<r n>
```

Induction via Abstract Refinements

```
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

incr acc by 1

$$R(i, acc) \Leftrightarrow \boxed{acc = i + z}$$

```
loop
  :: forall <r :: Int -> a -> Prop>.
     f:(i:Int -> a<r i> -> a<r (i+1)>)
  -> n:{ v:Int | v>=0 }
  -> z:a<r 0>
  -> a<r n>
```

Induction via Abstract Refinements

```
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

$$R(i, acc) \Leftrightarrow acc = i + z$$

```
loop [{\i acc -> acc = i + z}]
  :: f:(i:Int -> {v:a | v=i+z}
              -> {v:a | v=(i+1)+z})
  -> n:{v:Int | v>=0}
  -> z:Int
  -> {v:Int | v=n+z}
```

Induction via Abstract Refinements

```
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

$$\mathcal{R}(i, acc) \Leftrightarrow acc = i + z$$

```
loop [{\i acc -> acc = i + z}]
  :: f:(i:Int -> {v:a | v=i+z}
              -> {v:a | v=(i+1)+z})
  -> n:{v:Int | v>=0}
  -> z:Int
  -> {v:Int | v=n+z}
```

Induction via Abstract Refinements

```
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

$$\mathcal{R}(i, acc) \Leftrightarrow acc = i + z$$

```
loop [{\i acc -> acc = i + z}] f
  :: n:{v:Int | v>=0}
  -> z:Int
  -> {v:Int | v=n+z}
```

Induction via Abstract Refinements

```
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

$$\mathcal{R}(i, acc) \Leftrightarrow acc = i + z$$

```
loop [{\i acc -> acc = i + z}] f
   :: n:{v:Int | v>=0}
   -> z:Int
   -> {v:Int | v=n+z}
```

Induction via Abstract Refinements

```haskell
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

$$R(i, acc) \Leftrightarrow acc = i + z$$

```haskell
incr
  :: n:{v:Int | v>=0}
  -> z:Int
  -> {v:Int | v=n+z}
```

Induction via Abstract Refinements

```
incr :: n:{v:Int | v>=0}
     -> z:Int
     -> {v:Int | v=n+z}
incr n z = loop f n z
  where f i acc = acc + 1
```

Question: Does ``**incr** n z = n+z`` hold?

Answer: Yes

Outline

Introduction

Monomorphic Refinements

Applications

Refinements and Type Classes

Inductive Refinements

Indexed Refinements

Recursive Refinements

Evaluation

Outline

Introduction

Monomorphic Refinements

Applications

Refinements and Type Classes

Inductive Refinements

Indexed Refinements

Recursive Refinements

Evaluation

A Vector Data Type

```
data Vec a
  = V {f :: i:Int -> a}
```

Goal: Encode the domain of Vector

Encoding the Domain of a Vector

Abstract refinement

```
data Vec <d::Int -> Prop> a
  = V {f :: i:Int<d> -> a}
```

index satisfies d

Encoding the Domain of a Vector

```
data Vec <d::Int -> Prop> a
  = V {f :: i:Int<d> -> a}
```

Encoding the Domain of a Vector

```
data Vec <d::Int -> Prop> a
  = V {f :: i:Int<d> -> a}
```

"vector defined on **positive integers**"

Vec <{\v -> v > 0}> a

Encoding the Domain of a Vector

```
data Vec <d::Int -> Prop> a
  = V {f :: i:Int<d> -> a}
```

"vector defined **only on 1**"

Vec <{\v -> v = 1}> a

Encoding the Domain of a Vector

```
data Vec <d::Int -> Prop> a
  = V {f :: i:Int<d> -> a}
```

“vector defined on **the range 0 .. n**”

Vec <{\v -> 0 ≤ v < n}> a

Encoding Domain and Range of a Vector

Abstract refinement

```
data Vec <d::Int -> Prop, r::Int -> a -> Prop> a
   = V {f :: i:Int<d> -> a<r i>}
```

value satisfies r at i

Encoding Domain and Range of a Vector

```
data Vec <d::Int -> Prop, r::Int -> a -> Prop> a
  = V {f :: i:Int<d> -> a<r i>}
```

Encoding Domain and Range of a Vector

```
data Vec <d::Int -> Prop, r::Int -> a -> Prop> a
   = V {f :: i:Int<d> -> a<r i>}
```

"vector defined on **positive integers**, with **values equal** to their **index**"

Vec <{\v -> v > 0}, {\i v -> i = v}> **Int**

Encoding Domain and Range of a Vector

```
data Vec <d::Int -> Prop, r::Int -> a -> Prop> a
   = V {f :: i:Int<d> -> a<r i>}
```

"vector defined **only on 1**,
with **values equal** to **12**"

Vec <{\v -> v = 1}, {\i v -> v = 12}> **Int**

Null Terminating Strings

```
data Vec <d::Int -> Prop, r::Int -> a -> Prop> a
   = V {f :: i:Int<d> -> a<r i>}
```

"vector defined on **the range 0 .. n,**
with its **last value equal to `\0`**"

Vec <{\v -> 0 ≤ v < n},
{\i v -> i = n-1 => v = `\0`}> **Char**

Fibonacci Memoization

```
data Vec <d::Int -> Prop, r::Int -> a -> Prop> a
   = V {f :: i:Int<d> -> a<r i>}
```

“vector defined on **positives**,
with i-th value equal to **zero or i-th fibonacci**”

Vec <{\v -> 0 ≤ v},
{\i v -> v != 0 => v = fib(i)}> **Int**

Using Vectors

- **Abstract** over d and r in vector op (get, set, …)
- **Specify** vector properties (NullTerm, FibV, …)
- **Verify** that user functions preserve properties

Using Vectors

```
type NullTerm n =
   Vec <{\v -> 0<=v<n},
         {\i v -> i=n-1 => v='\0'}> Char

upperCase
   :: n:{v: Int| v>0}
   -> NullTerm n
   -> NullTerm n
upperCase n s = ucs 0 s where

ucs i s =
   case get i s of
   '\0' -> s
   c    -> ucs (i + 1) (set i (toUpper c) s)
```

Outline

Introduction

- Monomorphic Refinements

Applications

- Refinements and Type Classes
- Inductive Refinements
- **Indexed Refinements**
- Recursive Refinements

Evaluation

Outline

Introduction

Monomorphic Refinements

Applications

Refinements and Type Classes

Inductive Refinements

Indexed Refinements

Recursive Refinements

Evaluation

List Data Type

```
data List a
   = N
   | C (h :: a) (tl :: List a)
```

Goal: Relate tail elements with the head

Recursive Refinements

Abstract refinement

```
data List a <p :: a -> a -> Prop>
    = N
    | C (h :: a) (tl :: List <p> (a<p h>))
```

tail elements satisfy p at h

Unfolding Recursive Refinements

```
data List a <p :: a -> a -> Prop>
  = N
  | C (h :: a) (tl :: List <p> (a<p h>))
```

Unfolding Recursive Refinements

```
data List a <p :: a -> a -> Prop>
    = N
    | C (h :: a) (tl :: List <p> (a<p h>))
```

h_1 \`C\` h_2 \`C\` h_3 \`C\` N :: List <p> a

Unfolding Recursive Refinements (1/3)

```
data List a <p :: a -> a -> Prop>
    = N
    | C (h :: a) (tl :: List <p> (a<p h>))
```

h_1 \`C\` h_2 \`C\` h_3 \`C\` N :: List <p> a

h_1 :: a

tl_1 :: List <p> (a<p h_1>)

Unfolding Recursive Refinements (2/3)

```
data List a <p :: a -> a -> Prop>
   = N
   | C (h :: a) (tl :: List <p> (a<p h>))
```

h_1 \`C\` h_2 \`C\` h_3 \`C\` N :: List <p> a

h_1 :: a

h_2 :: a<p h_1>

tl_2 :: List <p> (a<p $h_1 \wedge$ p h_2>)

Unfolding Recursive Refinements (3/3)

```
data List a <p :: a -> a -> Prop>
   = N
   | C (h :: a) (tl :: List <p> (a<p h>))
```

h_1 \`C\` h_2 \`C\` h_3 \`C\` N :: List <p> a

h_1 :: a

h_2 :: a<p h_1>

h_3 :: a<p h_1 ∧ p h_2>

N :: List <p> (a<p h_1 ∧ p h_2 ∧ p h_3>)

Increasing Lists

```
data List a <p :: a -> a -> Prop>
   = N
   | C (h :: a) (tl :: List <p> (a<p h>))
```

```
type IncrL a = List <{\hd v -> hd ≤ v}> a
```

h_1 \`C\` h_2 \`C\` h_3 \`C\` N :: IncrL a

Increasing Lists

```
data List a <p :: a -> a -> Prop>
  = N
  | C (h :: a) (tl :: List <p> (a<p h>))
```

```
type IncrL a = List <{\hd v -> hd ≤ v}> a
```

h_1 \`C\` h_2 \`C\` h_3 \`C\` N :: IncrL a

h_1 :: a

h_2 :: { v:a | $h_1 \leq v$ }

h_3 :: { v:a | $h_1 \leq v \wedge h_2 \leq v$ }

N :: IncrL { v:a | $h_1 \leq v \wedge h_2 \leq v \wedge h_3 \leq v$ }

Increasing Lists

```
data List a <p :: a -> a -> Prop>
   = N
   | C (h :: a) (tl :: List <p> (a<p h>))
```

```
type IncrL a = List <{\hd v -> hd ≤ v}> a
```

Demo from
http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/

Outline

Introduction

- Monomorphic Refinements

Applications

- Refinements and Type Classes
- Inductive Refinements
- Indexed Refinements
- **Recursive Refinements**

Evaluation

Outline

Introduction

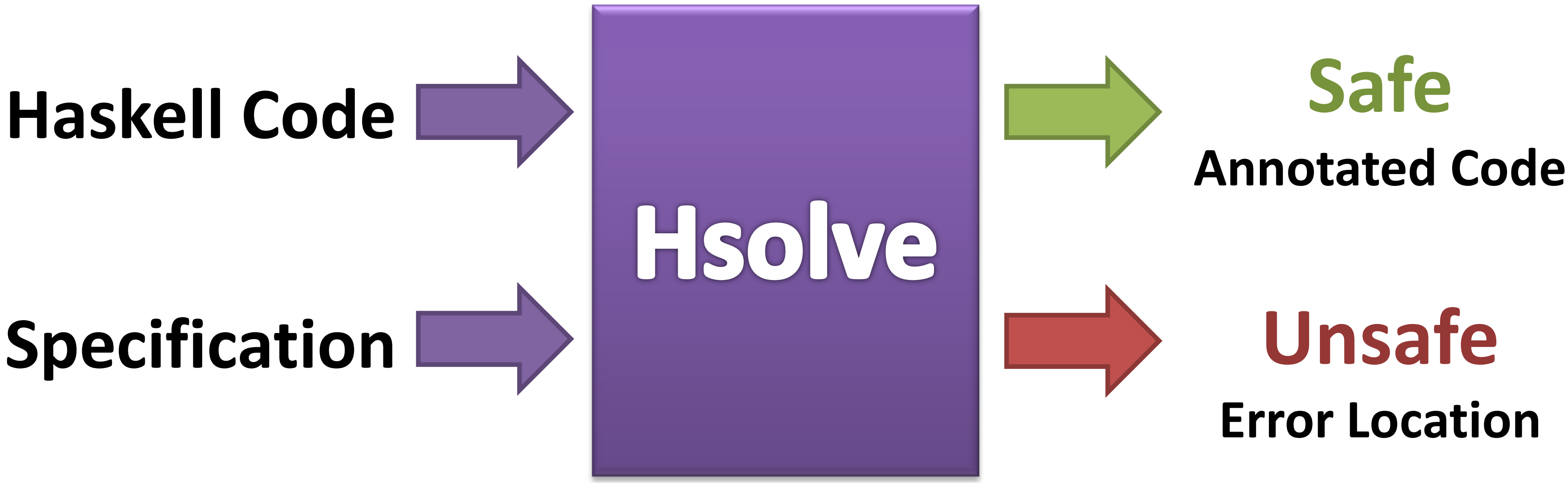
- Monomorphic Refinements

Applications

- Refinements and Type Classes
- Inductive Refinements
- Indexed Refinements
- Recursive Refinements

Evaluation

Our Tool


Haskell Code
Specification
Hsolve
Safe
Annotated Code
Unsafe
Error Location

Implementing HSolve

Hsolve = Liquid Types [PLDI 2008]
+ Abstract Refinements

Refinement Abstraction

Does not increase complexity

Refinement Application

Refinement application is **inferred**

Benchmarks

Program	LOC	Annotations	Time (s)
Micro	32	23	2
Vector	33	53	5
ListSort	29	5	3
Data.List.sort	71	4	8
Data.Set.Splay	136	24	13
Data.Map.Base	1395	152	136
Total	1696	261	167

Benchmarks

Program	LOC	Annotations	Time (s)
Micro	32	23	2
Vector	33	53	5
ListSort	29	5	3
Data.List.sort	71	4	8
Data.Set.Splay	136	24	13
Data.Map.Base	**1395**	**152**	**136**
Total	1696	261	167

Data.Map.Base

```
data Map k a
  = Bin Size k a (Map k a) (Map k a)
  | Tip
```

``Relate keys of **left** and **right** subtrees with the **root** key``

Data.Map.Base

```
data Map k a < l :: root:k -> k -> Prop
             , r :: root:k -> k -> Prop>
  = Bin (sz :: Size) (key :: k) (value :: a)
        (left  :: Map <l, r> (k <l key>) a)
        (right :: Map <l, r> (k <r key>) a)
  | Tip
```

``Relate keys of **left** and **right** subtrees with the **root** key``

Data.Map.Base

```
type OMap k a =
  Map <{\root v -> v<root}
      ,{\root v -> v>root}> k a
```

OMap is a BST

``left : keys less than the root

right: keys greater than the root``

Data.Map.Base

```
{-@ balanceL :: kcut:k -> a -> OMap {v:k | v < kcut} a -> OMap {v:k| v > kcut} a -> OMap k a @-}
balanceL :: k -> a -> Map k a -> Map k a -> Map k a

balanceL k x l r = case r of
  Tip -> case l of
        Tip -> Bin 1 k x Tip Tip
        (Bin _ _ _ Tip Tip) -> Bin 2 k x l Tip
        (Bin _ lk lx Tip (Bin _ lrk lrx _ _)) -> Bin 3 lrk lrx (Bin 1 lk lx Tip Tip) (Bin 1 k x Tip Tip)
        (Bin _ lk lx ll@(Bin _ _ _ _ _) Tip) -> Bin 3 lk lx ll (Bin 1 k x Tip Tip)
        (Bin ls lk lx ll@(Bin lls _ _ _ _) lr@(Bin lrs lrk lrx lrl lrr))
          | lrs < ratio*lls -> Bin (1+ls) lk lx ll (Bin (1+lrs) k x lr Tip)
          | otherwise -> Bin (1+ls) lrk lrx (Bin (1+lls+size lrl) lk lx ll lrl) (Bin (1+size lrr) k x lrr Tip)
  (Bin rs _ _ _ _) -> case l of
        Tip -> Bin (1+rs) k x Tip r
        (Bin ls lk lx ll lr)
          | ls > delta*rs -> case (ll, lr) of
              (Bin lls _ _ _ _, Bin lrs lrk lrx lrl lrr)
                | lrs < ratio*lls -> Bin (1+ls+rs) lk lx ll (Bin (1+rs+lrs) k x lr r)
                | otherwise -> Bin (1+ls+rs) lrk lrx (Bin (1+lls+size lrl) lk lx ll lrl) (Bin (1+rs+size lrr) k x lrr r)
              (_, _) -> error "Failure in Data.Map.balanceL"
          | otherwise -> Bin (1+ls+rs) k x l r
```

Data.Map.Base

```
{-@ balanceL :: kcut:k -> a -> OMap {v:k | v < kcut} a -> OMap {v:k| v > kcut} a -> OMap k a @-}
balanceL :: k -> a -> Map k a -> Map k a -> Map k a

balanceL k x l r = case r of
  Tip -> case l of
         Tip -> Bin 1 k x Tip Tip
         (Bin _ _ _ Tip Tip) -> Bin 2 k x l Tip
         (Bin _ lk lx Tip (Bin _ lrk lrx _ _)) -> Bin 3 lrk lrx (Bin 1 lk lx Tip Tip) (Bin 1 k x Tip Tip)
         (Bin _ lk lx ll@(Bin _ _ _ _ _) Tip) -> Bin 3 lk lx ll (Bin 1 k x Tip Tip)
         (Bin ls lk lx ll@(Bin lls _ _ _ _) lr@(Bin lrs lrk lrx lrl lrr))
           | lrs < ratio*lls -> Bin (1+ls) lk lx ll (Bin (1+lrs) k x lr Tip)
           | otherwise -> Bin (1+ls) lrk lrx (Bin (1+lls+size lrl) lk lx ll lrl) (Bin (1+size lrr) k x lrr Tip)
  (Bin rs _ _ _ _) -> case l of
         Tip -> Bin (1+rs) k x Tip r
         (Bin ls lk lx ll lr)
            | ls > delta*rs -> case (ll, lr) of
                (Bin lls _ _ _ _, Bin lrs lrk lrx lrl lrr)
                  | lrs < ratio*lls -> Bin (1+ls+rs) lk lx ll (Bin (1+rs+lrs) k x lr r)
                  | otherwise -> Bin (1+ls+rs) lrk lrx (Bin (1+lls+size lrl) lk lx ll lrl) (Bin (1+rs+size lrr) k x lrr r)
                (_, _) -> error "Failure in Data.Map.balanceL"
            | otherwise -> Bin (1+ls+rs) k x l r
```

Conclusion

Abstract Refinements

- Increase expressiveness without complexity
- Behave as uninterpreted functions in logic
- Relate arguments with result, i.e., max
- Relate expressions inside a structure, i.e., Vec, List
- Express recursive properties, i.e., List
- Express inductive properties, i.e., loop

Thank you!