Refinement Types and Abstract Refinements

Niki Vazou

UC San Diego

Simple Types

LIBRARY

```
div :: Int -> Int -> Int

div x y = x / y
```

USER

```
div e1 e2
```

Simple Type Error

LIBRARY

```
div :: Int -> Int -> Int

div x y = x / y
```

USER

```
div 4 "cat"
```

Simple Type Error

LIBRARY

```
div :: Int -> Int -> Int

div x y = x / y
```

USER

```
div 4 "cat"
```

Type error:

"Couldn't match expected type `Int` with actual type `String`"

Run Time Error

LIBRARY

```
div :: Int -> Int -> Int
div x y = x / y
```

USER

```
div 4 0
```

Run time error:

"Exception: divide by zero"

Refinement Types

an **`Int`** value, different than `0`

```
div :: Int
    -> {v:Int|v!=0}
    -> Int
```

Refinement Types

LIBRARY

```
div :: Int -> {v:Int | v!=0} -> Int
div x y = x / y
```

USER

```
div 4 e        -- e::Int
```

Refinement Types

LIBRARY

```
div :: Int -> {v:Int|v!=0} -> Int

div x y = x / y
```

USER

```
div 4 e          -- e::Int
```

Type error:

"Couldn't match expected type `{v:Int|v!=0}` with actual type `Int`"

Contracts as Casts

$$\langle \boxed{\texttt{Int}} \Rightarrow \{\texttt{v:Int} \mid \texttt{v!=0}\} \rangle^{\texttt{l}}\ \mathbf{e}$$

Cast from source type $\boxed{\texttt{Int}}$
to target type $\{\texttt{v:Int} \mid \texttt{v!=0}\}$
with a label $\texttt{l}$

Contracts as Casts

$$\langle \texttt{Int} \Rightarrow \boxed{\{\texttt{v:Int} \mid \texttt{v!=0}\}} \rangle^{\texttt{l}}\ \texttt{e}$$

Cast from source type `Int`

to target type $\boxed{\{\texttt{v:Int} \mid \texttt{v!=0}\}}$

with a label `l`

Contracts as Casts

$\langle \mathtt{Int} \Rightarrow \{\mathtt{v:Int} \mid \mathtt{v!=0}\} \rangle^{\boxed{1}} \mathbf{e}$

Cast from source type `Int`
to target type `{v:Int | v!=0}`
with a label $\boxed{1}$

Contracts as Casts

$\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v!=0}\} \rangle^{\texttt{l}}\ \texttt{e}$

$\rightarrow$ **if** (e!=0) **then** e **else** $\Uparrow$ l

Cast from source type `Int`
to target type `{v:Int | v!=0}`
with a label `l`

Contracts

LIBRARY

```
div :: Int -> {v:Int | v!=0} -> Int
div x y = x / y
```

USER

```
div 4 (<Int ⇒ {v:Int | v!=0}>¹ e)
```

$e \rightarrow 0$

Run time error:

"Exception: ⇑1"

A predecessor example

```
pred :: Int -> Int
pred n = n - 1
```

“the result is less than the argument”

A predecessor example

```
pred :: n:Int -> {v:Int | v < n}

pred n = n - 1
```

“the result is less than the argument”

Using predecessor

LIBRARY

```
pred :: n:Int -> {v:Int | v<n}

pred = <Int -> Int ⇒ n:Int->{v:Int | v<n}>¹ f
  where f x = x - 1 :: Int -> Int
```

Using predecessor

LIBRARY

```
pred :: n:Int -> {v:Int | v<n}

pred = < Int -> Int ⇒ n:Int->{v:Int | v<n}>¹f
  where f x = x - 1 :: Int -> Int
```

Using predecessor

LIBRARY

```
pred :: n:Int -> {v:Int | v<n}
pred = < Int -> Int ⇒ n:Int->{v:Int| v<n}>¹ f
  where f x = x - 1 :: Int -> Int
```

USER

```
p = pred 4                          -- assert (p<4) ✓
```

```
pred 4 :: {v:Int | v<4}
```

Using predecessor

LIBRARY

```
pred :: n:Int -> {v:Int | v<n}

pred = < Int -> Int ⇒ n:Int->{v:Int | v<n}>¹ f
   where f x = x - 1 :: Int -> Int
```

USER

```
p = pred 4                          -- assert (p<4)
```

Using predecessor

LIBRARY

```
pred :: n:Int -> {v:Int | v<n}

pred = < Int -> Int ⇒ n:Int->{v:Int| v<n}>¹ f
  where f x = x + 1 :: Int -> Int
```

USER

```
p = pred 4                              -- assert (p<4)
```

Run time error:

"Exception: ⇑`Library.1`"

Functional Specifications

Refinement Types as Functional Specifications:

```
pred :: n:Int -> {v:Int | v < n}
```

```
div :: Int -> {v:Int | v =!0} -> Int
```

Specifications:

Properties that the program should satisfy

Functional Specifications:

Treat the program as collection of functions

Functional Specifications

Refinement Types as Functional Specifications:

```
pred :: n:Int -> {v:Int | v < n}
```

```
pred n = n + 1
```

✗

```
_ = div 4 0
```

✗

Check Specifications with Contracts

1970 Object Oriented Programming **Eiffel**

2002 Higher Order Programming (Findler, Felleisen)

✓ **Expressive** (express higher order predicates)
✓ **Blame assignment** (to the supplier of bad value)

✗ **Run time** checks (consume computation cycles)
✗ **Limited coverage** (one execution path is checked)

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Refinements and Type Classes

Inductive Refinements

Indexed Refinements

Recursive Refinements

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Refinement Types

LIBRARY

```
div :: Int -> {v:Int|v!=0} -> Int

div x y = x / y
```

USER

```
          div 4 e          -- e::Int
```

```
e :: {v:Int|v!=0} ?
```

Refinement Types

We **want**

```
e :: {v:Int | v!=0}
```

$$p \Rightarrow v!=0$$

We **have**

```
e :: {v:Int | p}
```

Basic Subtyping

$$\frac{p_s \Rightarrow p_t}{\{\, v : \mathbf{b} \mid p_s \} <: \{\, v : \mathbf{b} \mid p_t \}}$$

$$s <: t$$

If e::s **then** e::t

Decidable Subtyping

$$\frac{p_s \Rightarrow p_t}{\{\, v : b \mid \boxed{p_s} \,\} <: \{\, v : b \mid \boxed{p_t} \,\}}$$

refinement language in decidable theories

Propositional Logic +

Theories (equality, linear arithmetic, unint. functions)

Liquid Types

$\mathbb{Q}$: Logical qualifiers (predicates on v, $\star$)

e.g., $\mathbb{Q}$ = {`v>0`, `★>0`, `v<★`, `v=★-1`}

$\mathbb{Q}^\star$: instantiate $\star$ with program variables

e.g., $\mathbb{Q}^\star$ = {`v>0`, `y>0`, `v<n`, `v=n+1`}

Liquid Types: { v:b | p }

with p=$\bigwedge q$, $q \in \mathbb{Q}^\star$

refinement language in decidable theories

Propositional Logic +

Theories (equality, linear arithmetic, unint. functions)

Liquid Types

$\mathbb{Q}$: Logical qualifiers (predicates on v, $\star$)

e.g., $\mathbb{Q}$ = {v>0, $\star$>0, v<$\star$, v=$\star$-1}

$\mathbb{Q}^\star$: instantiate $\star$ with program variables

e.g., $\mathbb{Q}^\star$ = {v>0, y>0, v<n, v=n+1}

Liquid Types: { v:b | p **}**

with p=$\wedge\, q$, $q \in \mathbb{Q}^\star$

refinement language in decidable theories

Propositional Logic +

Theories (equality, linear arithmetic, unint. functions)

Basic Subtyping

$$\frac{p_s \Rightarrow p_t}{\{\, v : b \mid \boxed{p_s} \,\} <: \{\, v : b \mid \boxed{p_t} \,\}}$$

refinement language in decidable theories

Propositional Logic +

Theories (equality, linear arithmetic, unint. functions)

Basic Subtyping

SMT solver:
SAT + Theory Solvers

$$\frac{\boxed{p_s \Rightarrow p_t}}{\{ v : b \mid p_s \} <: \{ v : b \mid p_t \}}$$

refinement language in decidable theories
Propositional Logic +
Theories (equality, linear arithmetic, unint. functions)

Predecessor Example

```
pred :: n:Int->{v:Int| v<n}
pred n = n - 1
```

Predecessor Example

```
pred :: n:Int->{v:Int| v<n}
pred n = n - 1
```

We **want**

```
n :: Int
```

```
n-1 :: {v:Int|v<n}
```

Predecessor Example

```
pred :: n:Int->{v:Int| v<n}
pred n = n - 1
```

```
n   :: Int
(-) :: x:Int -> y:Int -> {v:Int|v=x-y}
```

Predecessor Example

```
pred :: n:Int->{v:Int| v<n}
pred n = n - 1
```

```
v=1 ⇒ true
```

```
1   :: {v:Int|v=1} <: Int
n   :: Int
(-) :: x:Int -> y:Int -> {v:Int|v=x-y}
```

```
(n-1) :: {v:Int|v=n-1}
```

Predecessor Example

```
pred :: n:Int->{v:Int| v<n}
pred n = n - 1
```

```
1   :: {v:Int|v=1} <: Int
n   :: Int
(-) :: x:Int -> y:Int -> {v:Int|v=x-y}
```

$$\frac{\begin{array}{l}\texttt{1 :: \{v:Int|v=1\} <: Int} \\ \texttt{n :: Int} \\ \texttt{(-) :: x:Int -> y:Int -> \{v:Int|v=x-y\}}\end{array}}{\texttt{(n-1) :: \{v:Int|v=n-1\}<:\{v:Int|v<n\}}}\ \boxed{\texttt{v=n-1} \Rightarrow \texttt{v<n}}$$

```
(n-1) :: {v:Int|v=n-1}<:{v:Int|v<n}
```

Predecessor Example

```
pred :: n:Int->{v:Int| v<n}
pred n = n - 1
```

```
1   :: {v:Int|v=1} <: Int
n   :: Int
(-) :: x:Int -> y:Int -> {v:Int|v=x-y}
```

```
(n-1) :: {v:Int|v<n}
```

✓

Division Example

```
div :: Int -> {v:Int | v!=0} -> Int
```

$$\dfrac{4::\{v:Int \mid v=4\} \qquad \dfrac{v=4 \Rightarrow true}{\{v:Int \mid v=4\} <: Int}}{\dfrac{4 :: Int}{div\ 4 :: ???}}$$

Division Example

```
div :: Int -> {v:Int|v!=0} -> Int
```

```
div 4 :: {v:Int|v!=0} -> Int
```

$$\dfrac{\dfrac{4::\{v:Int\,|\,v=4\} \qquad \dfrac{v=4 \Rightarrow true}{\{v:Int\,|\,v=4\} <: Int}}{4 :: Int}}{div\ 4 :: \{v:Int\,|\,v!=0\} \rightarrow Int}$$

Division Example

```
div :: Int -> {v:Int|v!=0} -> Int
```

```
div 4 :: {v:Int|v!=0} -> Int
```

$$\frac{\dfrac{\dfrac{v=2 \Rightarrow v!=0}{2::\{v:Int|v=2\} \quad \{v:Int|v=2\}<:\{v:Int|v!=0\}}}{2 :: \{v:Int|v!=0\}}}{div\ 4\ 2 :: Int}$$

Division Example

```
div :: Int -> {v:Int|v!=0} -> Int
```

```
div 4 :: {v:Int|v!=0} -> Int
```

$$\dfrac{\dfrac{\dfrac{v=2 \Rightarrow v!=0}{\texttt{2::\{v:Int|v=2\}} \quad \texttt{\{v:Int|v=2\}<:\{v:Int|v!=0\}}}}{\texttt{2 :: \{v:Int|v!=0\}}}}{\texttt{div 4 0 :: ???}}$$

Division Example

```
div :: Int -> {v:Int | v!=0} -> Int
```

```
div 4 :: {v:Int | v!=0} -> Int
```

```
                    v=0 ⇒ v!=0        ✗
              ──────────────────────────────
0::{v:Int|v=0}  {v:Int|v=0} <: {v:Int|v!=0}
─────────────────────────────────────────────
           0 :: {v:Int|v!=0}              ✗
─────────────────────────────────────────────
           div 4 0 :: ???                 ✗
```

Refinement Types

1991 Freeman and Pfenning

Refine specific data types (nil, singleton list)

1999 DML(C)

Refinements from a decidable domain C

2008 **Liquid Types** (Rondon *et. al.*)

Algorithmic Type Inference

✓ **Static Verification**

✓ **Limited annotations**

✗ **Limited expressiveness**

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Refinements and Type Classes

Inductive Refinements

Indexed Refinements

Recursive Refinements

Outline

Introduction

- Contracts

Refinement Types

- Liquid Types

Abstract Refinements

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

Max example

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

Max example

```
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)
```

$$\boxed{v \geq 12 \Rightarrow v > 0}$$

$$\text{max } 8\ 12 :: \{\, v : \textbf{Int} \mid \boxed{v \geq 12}\,\} \mathrel{\textbf{<:}} \{\, v : \textbf{Int} \mid \boxed{v > 0}\,\}$$

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**

$$\text{max } 3\ 5 :: \{\, v : \mathbf{Int} \mid v \geq 5 \,\}$$

We **want**

$$\text{max } 3\ 5 :: \{\, v : \mathbf{Int} \mid v \geq 5 \land \mathbf{odd}\ v \,\}$$

Using max

Problem:

Information of Input Refinements is Lost

We **get**

$$\text{max } 3\ 5 :: \{\ v : \textbf{Int} \mid v \geq 5\ \}$$

We **want**

$$\text{max } 3\ 5 :: \{\ v : \textbf{Int} \mid v \geq 5 \land \textbf{odd } v\ \}$$

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

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**"

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

```
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**”

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Refinements and Type Classes

Inductive Refinements

Indexed Refinements

Recursive Refinements

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Inductive Refinements

Indexed Refinements

Recursive Refinements

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Inductive Refinements

Indexed Refinements

Recursive Refinements

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

```
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

```
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
```

$$R :: (\mathbf{Int}, a)$$

$$\frac{R(0, z) \qquad R(i, acc) \Rightarrow R(i+1, f\ i\ acc)}{R(n, loop\ f\ n\ z)}$$

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>
```

$$\frac{R(i, acc) \Rightarrow R(i+1, f\ i\ acc)}{R(n, loop\ f\ n\ z)}$$

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
```

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)

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)	`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

```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 -> 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

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

```
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
```

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

```
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
```

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

```
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
```

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

```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

  Contracts

Refinement Types

  Liquid Types

Abstract Refinements

  Inductive Refinements

  Indexed Refinements

  Recursive Refinements

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Inductive Refinements

Indexed Refinements

Recursive Refinements

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

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Inductive Refinements

Indexed Refinements

Recursive Refinements

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Inductive Refinements

Indexed Refinements

Recursive Refinements

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 ∧ 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 $h_1 \wedge$ p h_2>

N :: List <p> (a<p $h_1 \wedge$ p $h_2 \wedge$ 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
```

$$h_1 \text{ `C` } h_2 \text{ `C` } h_3 \text{ `C` } N :: \text{IncrL } a$$

$h_1 :: a$

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

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

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

Sorting 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

insert :: y:a -> IncrL a -> IncrL a
insert y N = N
insert y (x`C`xs) | y < x      = y `C` x `C` xs
                  | otherwise = y `C` insert y xs

insertSort :: xs:[a] -> IncrL a
insertSort = foldr insert N
```

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Inductive Refinements

Indexed Refinements

Recursive Refinements

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Inductive Refinements

Indexed Refinements

Recursive Refinements

Outline

Introduction

Contracts

Refinement Types

Liquid Types

Abstract Refinements

Abstract Refinements

LiquidHaskell = Liquid Types
+ Abstract Refinements

Increase expressiveness without complexity

- Relate arguments with result, i.e., max
- Relate expressions inside a structure, i.e., Vec, List
- Express inductive properties, i.e., loop

Conclusion

Refinement Types for Functional Specifications

Verify Specifications

At run Time (Contracts)

✓ **Expressive** ✗ **Run time** checks

Statically (Liquid Types)

✗ **Less Expressive** ✓ **Static** verification

Abstract Refinements

Increase expressiveness without complexity

Thank you!