# Refinement Types
# and
# Abstract Refinements

**Niki Vazou**

UC San Diego

# Simple Types

```
    4 :: Int
"cat" :: String
  div :: Int -> Int -> Int
```

# Simple Type Errors

```
div :: Int -> Int -> Int
div 4 "cat"
```

# Simple Type Errors

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

```
div 4 "cat"
```

**Type error:**

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

# Run Time Errors

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

div 4 0
```

# Run Time Errors

an `Int` value, different than `0`

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

```
div 4 0
```

**Run time error:**

"Exception: divide by zero"

# Refinement Types

an `Int` value, different than `0`

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

# Refinement Types

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

# Refinement Function Types

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

# Refinement Function Types

```
pred :: n:{v:Int | v >0} -> {v:Int | v = n-1}

pred n = n - 1
```

# Refinement Function Types

```
pred :: n:{v:Int | v >0} -> {v:Int | v = n-1}

pred n = n - 1
```

```
pred 2 :: {v:Int | v = 1}
```

# Functional Specifications

Refinement Types as Functional Specifications:

```
pred :: n:{v:Int | v >0} -> {v:Int | v = n-1}
```

```
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:{v:Int | v >0} -> {v:Int | v = n-1}
```

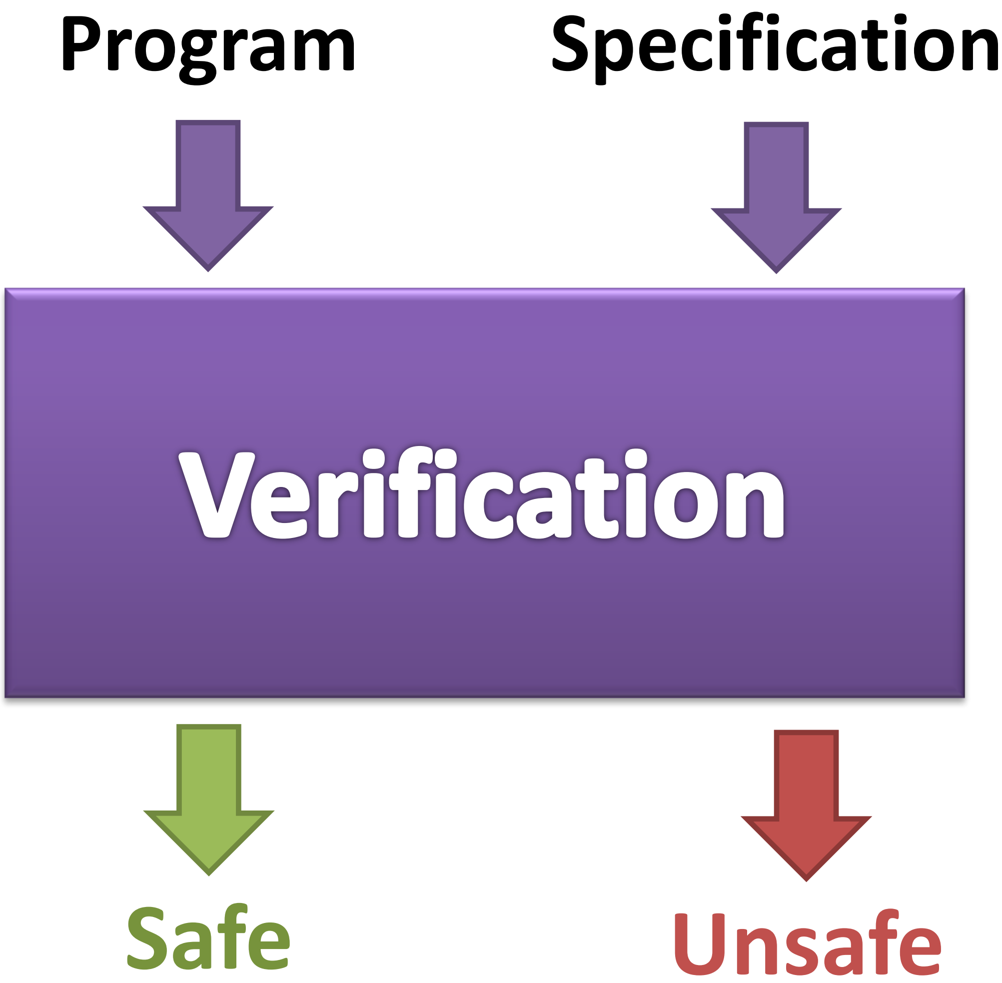
```
pred n = 0
```

✗

```
_ = div 4 0
```

✗

# Verification



# Decidability vs Expressiveness

$$\texttt{f::(a->b)->\{v:Bool|}\underbrace{\texttt{terminates f}}_{\text{refinement language}}\texttt{\}}$$

Arbitrary **refinement language**

expressive specifications

undecidable verification

Restrict **refinement language** (decidable logic)

less expressive specifications

decidable verification

# Outline

**Introduction**

Contracts

Liquid Types

Abstract Refinements

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

# Outline

Introduction

**Contracts**

Liquid Types

Abstract Refinements

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

# Basic Contracts

$$\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v>0}\} \rangle\ \texttt{2}$$

# Basic Contracts

$$\langle\, \mathtt{Int} \Rightarrow \{v{:}\mathtt{Int} \mid v>0\} \,\rangle\ 2$$

$$\rightarrow \textbf{if}\ (v>0)[2/v]\ \textbf{then}\ 2$$

$$\rightarrow \textbf{if}\ (2>0)\ \textbf{then}\ 2$$

$$\rightarrow 2$$

**Statically:**

assert $2 : \mathtt{Int}$

assume $2 : \{v{:}\mathtt{Int} \mid v>0\}$

**Dynamically:**

if **check** succeeds

then return 2

# Basic Contracts

$$\langle \texttt{Int} \Rightarrow \{v:\texttt{Int} \mid v>0\} \rangle\ 2$$

**Statically:**

assert $2 : \texttt{Int}$

assume $2 : \{v:\texttt{Int} \mid v>0\}$

**Dynamically:**

if **check** succeeds

then return 2

# Basic Contracts

$$\langle \mathtt{Int} \Rightarrow \{\mathtt{v}:\mathtt{Int} \mid \mathtt{v}>0\} \rangle\ 0$$

**Statically:**

assert $0 : \mathtt{Int}$

assume $0 : \{\mathtt{v}:\mathtt{Int} \mid \mathtt{v}>0\}$

**Dynamically:**

if **check** succeeds

then return 0

# Basic Contracts

$$\langle \mathtt{Int} \Rightarrow \{v{:}\mathtt{Int} \mid v{>}0\} \rangle\ 0$$

$$\rightarrow \mathbf{if}\ (v{>}0)[0/v]\ \mathbf{then}\ 0$$

$$\rightarrow \mathbf{if}\ (0{>}0)\ \mathbf{then}\ 0$$

**Statically:**

assert $0 : \mathtt{Int}$

assume $0 : \{v{:}\mathtt{Int} \mid v{>}0\}$

**Dynamically:**

if **check** succeeds

then return 0

# Basic Contracts

$$\langle \mathtt{Int} \Rightarrow \{v:\mathtt{Int} \mid v>0\} \rangle^{1}\ 0$$

$$\to \textbf{if } (v>0)[0/v] \textbf{ then } 0$$

$$\to \textbf{if } (0>0) \textbf{ then } 0$$

**Statically:**

assert $0 : \mathtt{Int}$

assume $0 : \{v:\mathtt{Int} \mid v>0\}$

**Dynamically:**

if **check** succeeds

then return 0

# Basic Contracts

$$\langle \mathtt{Int} \Rightarrow \{\mathtt{v}:\mathtt{Int} \mid \mathtt{v>0}\} \rangle^{\mathtt{l}}\ \mathtt{0}$$

$$\rightarrow \textbf{if}\ (\mathtt{v>0})[\mathtt{0/v}]\ \textbf{then}\ \mathtt{0}\ \textbf{else}\ \Uparrow \mathtt{l}$$

$$\rightarrow \textbf{if}\ (\mathtt{0>0})\ \textbf{then}\ \mathtt{0}\ \textbf{else}\ \Uparrow \mathtt{l}$$

**Statically:**

assert $\mathtt{0} : \mathtt{Int}$

assume $\mathtt{0} : \{\mathtt{v}:\mathtt{Int} \mid \mathtt{v>0}\}$

**Dynamically:**

if **check** succeeds

then return $\mathtt{0}$

else blame $\mathtt{l}$

# Basic Contracts

$$\langle \mathtt{Int} \Rightarrow \{\mathtt{v}:\mathtt{Int} \mid \mathtt{v>0}\}\rangle^{l}\ \mathtt{0}$$

$$\rightarrow \textbf{if}\ \mathtt{(v>0)[0/v]}\ \textbf{then}\ \mathtt{0}\ \textbf{else}\ \Uparrow l$$

$$\rightarrow \textbf{if}\ \mathtt{(0>0)}\ \textbf{then}\ \mathtt{0}\ \textbf{else}\ \Uparrow l$$

$$\rightarrow \Uparrow l$$

**Statically:**

assert $\mathtt{0} : \mathtt{Int}$

assume $\mathtt{0} : \{\mathtt{v}:\mathtt{Int} \mid \mathtt{v>0}\}$

**Dynamically:**

if **check** succeeds

then return $\mathtt{0}$

else blame $l$

# Basic Contracts

$$\langle \mathtt{Int} \Rightarrow \{\mathtt{v}:\mathtt{Int} \mid \mathtt{v>0}\} \rangle^{l}\ \mathtt{0}$$

**Statically:**

assert $\mathtt{0} : \mathtt{Int}$

assume $\mathtt{0} : \{\mathtt{v}:\mathtt{Int} \mid \mathtt{v>0}\}$

**Dynamically:**

if **check** succeeds

then return $\mathtt{0}$

else blame $l$

# Basic Contracts

$$\langle \{v:\mathbf{b} \mid p_s\} \Rightarrow \{v:\mathbf{b} \mid p_t\} \rangle^{l}\, e$$

$$\rightarrow \mathbf{if}\ p_t[e/v]\ \mathbf{then}\ e\ \mathbf{else}\ \Uparrow l$$

**Statically:**

assert $e : \{v:\mathbf{b} \mid p_s\}$

assume $e : \{v:\mathbf{b} \mid p_t\}$

**Dynamically:**

if **check** succeeds

then return e

else blame $l$

# Functional Contracts

$$\langle s_x \texttt{->} s \Rightarrow t_x \texttt{->} t \rangle^{l} f$$

**Statically:**

assert $f : s_x \texttt{->} s$

assume $f : t_x \texttt{->} t$

# Functional Contracts

$$(\langle s_x \text{->} s \Rightarrow t_x \text{->} t \rangle^l f)\ v$$

$$\rightarrow \langle s \Rightarrow t \rangle^l (f\,(\langle t_x \Rightarrow s_x \rangle^l v))$$

**Statically:**

assert $f : s_x \text{->} s$

assume $f : t_x \text{->} t$

# Functional Contracts

$$(\langle s_x \texttt{->} s \Rightarrow t_x \texttt{->} t \rangle^{l} f)\ v$$

$$\rightarrow \langle s \Rightarrow t \rangle^{l} (f\ \boxed{(\langle t_x \Rightarrow s_x \rangle^{l} v)})$$

**Statically:**

assert $f : s_x \texttt{->} s$

assume $f : t_x \texttt{->} t$

# Functional Contracts

$$(\langle s_x \to s \Rightarrow t_x \to t \rangle^{l} f)\ v$$

$$\to \langle s \Rightarrow t \rangle^{l} \boxed{(f\,(\langle t_x \Rightarrow s_x \rangle^{l} v))}$$

**Statically:**

assert $f : s_x \to s$

assume $f : t_x \to t$

# Functional Contracts

$$(\langle s_x \to s \Rightarrow t_x \to t \rangle^{l} f)\ v$$

$$\to \boxed{\langle s \Rightarrow t \rangle^{l} (f\,(\langle t_x \Rightarrow s_x \rangle^{l}\, v))}$$

**Statically:**

assert $f : s_x \to s$

assume $f : t_x \to t$

# Functional Contracts

$$(\langle s_x \text{ -> } s \Rightarrow t_x \text{ -> } t \rangle^{l} f)\ \ v$$

$$\rightarrow \langle s \Rightarrow t \rangle^{l} (f\,(\langle t_x \Rightarrow s_x \rangle^{l} v))$$

**Statically:**

assert $f : s_x \text{ -> } s$

assume $f : t_x \text{ -> } t$

# Predecessor example

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

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred=< Int -> Int ⇒ n:{v:Int|v>0}->{v:Int| v=n-1}>^l f
```

# Predecessor example

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

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred=<Int -> Int => n:{v:Int|v>0}->{v:Int| v=n-1}>1 f
```

# Predecessor example

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

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred=< Int -> Int ⇒ n:{v:Int|v>0}->{v:Int| v=n-1}>1 f
```

# Predecessor example

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

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred=< Int -> Int ⇒ n:{v:Int|v>0}->{v:Int| v=n-1}>^l f
```

# Predecessor example

```
pred 2
 = (<Int -> Int ⇒ n:{v:Int|v>0}->{v:Int| v=n-1}>¹f) 2
```

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred =< Int -> Int ⇒ n:{v:Int|v>0}->{v:Int| v=n-1}>¹f
```

# Predecessor example

pred 2

$$= (\langle \mathtt{Int} \to \mathtt{Int} \Rightarrow n{:}\{v{:}\mathtt{Int} \mid v>0\} \to \{v{:}\mathtt{Int} \mid v=n-1\}\rangle^{l}\ f)\ 2$$

$$(\langle s_x \to s \Rightarrow t_x \to t \rangle^{l}\ f)\ v$$

$$\longrightarrow \langle s \Rightarrow t \rangle^{l} (f\,(\langle t_x \Rightarrow s_x \rangle^{l}\, v))$$

# Predecessor example

```
pred 2
 = (<Int -> Int ⇒ n:{v:Int | v>0}->{v:Int | v=n-1}>¹ f) 2
 → <Int ⇒ {v:Int | v=2-1}>¹ (f (<{v:Int | v>0} ⇒ Int >¹ 2))
```

$$(\langle s_x \text{->} s \Rightarrow t_x \text{->} t \rangle^{l} f)\ v$$

$$\rightarrow \langle s \Rightarrow t \rangle^{l} (f\,(\langle t_x \Rightarrow s_x \rangle^{l} v))$$

# Predecessor example

```
pred 2
 = (<Int -> Int ⇒ n:{v:Int|v>0}->{v:Int| v=n-1}>^l f) 2
 → <Int ⇒ {v:Int| v=2-1}>^l (f (<{v:Int|v>0} ⇒ Int >^l 2))
 → <Int ⇒ {v:Int| v=2-1}>^l (f 2)
```

# Predecessor example

```
pred 2
 = (<Int -> Int ⇒ n:{v:Int|v>0}->{v:Int| v=n-1}>^l f) 2
 → <Int ⇒ {v:Int| v=2-1}>^l (f (<{v:Int|v>0} ⇒ Int >^l 2))
 → <Int ⇒ {v:Int| v=2-1}>^l (f 2)
 → <Int ⇒ {v:Int| v=2-1}>^l 1
```

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

# Predecessor example

```
pred 2
 = (<Int -> Int ⇒ n:{v:Int | v>0}->{v:Int | v=n-1}>^l f) 2
 → <Int ⇒ {v:Int | v=2-1}>^l (f (<{v:Int | v>0} ⇒ Int >^l 2))
 → <Int ⇒ {v:Int | v=2-1}>^l (f 2)
 → <Int ⇒ {v:Int | v=2-1}>^l 1
 → 1
```

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

# Predecessor example

```
pred 2
 = (<Int -> Int ⇒ n:{v:Int | v>0} -> {v:Int | v=n-1}>^l f) 2
 → <Int ⇒ {v:Int | v=2-1}>^l (f (<{v:Int | v>0} ⇒ Int>^l 2))
 → <Int ⇒ {v:Int | v=2-1}>^l (f 2)
 → <Int ⇒ {v:Int | v=2-1}>^l 0
 → ⇑l
```

```
f :: Int -> Int
f n = 0
```

# Predecessor example

```
pred 0        ✗
```

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred=< Int -> Int ⇒ n:{v:Int|v>0}->{v:Int| v=n-1}>¹ f
```

# Predecessor example

$$\texttt{pred}\ (\langle\, \texttt{Int} \Rightarrow \boxed{\{v{:}\texttt{Int} \mid v>0\}} \,\rangle^{\texttt{zero}}\ 0)$$

$$\texttt{pred} :: \boxed{n{:}\{v{:}\texttt{Int} \mid v>0\}} \texttt{->} \{v{:}\texttt{Int} \mid v=n-1\}$$

$$\texttt{pred} = \langle\, \texttt{Int} \texttt{->} \texttt{Int} \Rightarrow n{:}\{v{:}\texttt{Int} \mid v>0\} \texttt{->} \{v{:}\texttt{Int} \mid v=n-1\} \,\rangle^{1}\ f$$

# Predecessor example

$$\texttt{pred}\ (\langle\, \texttt{Int} \Rightarrow \{\texttt{v}:\texttt{Int} \mid \texttt{v>0}\} \rangle^{\texttt{zero}}\ \texttt{0})$$

$$\rightarrow\ \Uparrow \texttt{zero}$$

$$\texttt{pred} :: \texttt{n}:\{\texttt{v}:\texttt{Int} \mid \texttt{v>0}\} \texttt{->} \{\texttt{v}:\texttt{Int} \mid \texttt{v=n-1}\}$$

$$\texttt{pred} = \langle\ \texttt{Int -> Int} \Rightarrow \texttt{n}:\{\texttt{v}:\texttt{Int} \mid \texttt{v>0}\} \texttt{->} \{\texttt{v}:\texttt{Int} \mid \texttt{v=n-1}\} \rangle^{1}\ \texttt{f}$$

# Contracts

1970 Object Oriented Programming **Eiffel**

2002 Higher Order Programming (Findler, Felleisen)

**Racket**

✓ **Expressive** (express higher order predicates)

# Contracts

```
;; contract for the derivative function
;; for some natural number n and reals δ, ε:
(->d ([f (0<real<1? . -> . 0<real<1?)])
     (fp (0<real<1? . -> . real?))
     #:post-cond
     (for/and ([i (in-range 0 n)])
       (define x (random-number))
       (define slope (/ (- (f (- x ε))
                           (f (+ x ε)))
                        (* 2 ε)))
       (<= (abs (- slope (fp x))) δ)))
```

# Contracts

1970 Object Oriented Programming **Eiffel**

2002 Higher Order Programming (Findler, Felleisen)

**Racket**

✓ **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**

Liquid Types

Abstract Refinements

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

# Outline

Introduction

Contracts

**Liquid Types**

Abstract Refinements

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

# Refinement Types

```
2 :: {v:Int | v>0}
```

$v=2 \Rightarrow v>0$

```
2 :: {v:Int | v=2}
```

# Basic Subtyping

$$\frac{v=2 \Rightarrow v>0}{\{v:\mathtt{Int} \mid v=2\} <: \{v:\mathtt{Int} \mid v>0\}}$$

**If** `e::s` **and** `s<:t`

**then** `e::t`

# Basic Subtyping

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

**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=`$\wedge\, q$, $q \in \mathbb{Q}^\star$

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

**refinement language** in decidable theories

Propositional Logic +

Theories (equality, linear arithmetic, unint. functions)

# Basic Subtyping

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

**refinement language** in decidable theories

Propositional Logic +

Theories (equality, linear arithmetic, unint. functions)

# Basic Subtyping

**SMT solver:**
SAT + Theory Solvers

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

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

# Functional Subtyping

$$\frac{\Gamma \vdash t_x <: s_x \qquad \Gamma \vdash s <: t}{\Gamma \vdash s_x \texttt{->} s <: t_x \texttt{->} t}$$

# Predecessor Example

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred n = n - 1
```

# Function Typing

$$\frac{\Gamma, x{:}t_x \vdash e :: t}{\Gamma \vdash \backslash x{:}t_x\text{->}\ e :: x{:}t_x\text{->}t}$$

# Predecessor Example

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred n = n - 1
```

We **want**

```
n :: {v:Int|v>0}
```

---

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

# Predecessor Example

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred n = n - 1
```

```
n   :: {v:Int|v>0}
(-) :: x:Int -> y:Int -> {v:Int|v=x-y}
```

# Application Typing

$$\frac{\Gamma \vdash e_f :: (x{:}t_x \texttt{->} t) \quad \Gamma \vdash e :: t_x}{\Gamma \vdash e_f\ e :: t\ [e/x]}$$

# Predecessor Example

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred n = n - 1
```

```
v>0 ⇒ true
```

```
n   :: {v:Int|v>0} <: Int
(-) :: x:Int -> y:Int -> {v:Int|v=x-y}
```

# Predecessor Example

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred n = n - 1
```

```
1   :: {v:Int|v=1} <: Int
n   :: {v:Int|v>0} <: Int
(-) :: x:Int -> y:Int -> {v:Int|v=x-y}
```

---

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

✓

# Predecessor Example

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred n = n - 1
```

```
                    v=2 ⇒ v>0
              ─────────────────────
2::{v:Int|v=2}  {v:Int|v=2}<:{v:Int|v>0}
─────────────────────────────────────────
           2 :: {v:Int|v>0}
─────────────────────────────────────────
           pred 2 :: ???
```

# Predecessor Example

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred n = n - 1
```

$$\frac{\dfrac{2::\{v:Int \mid v=2\} \qquad \dfrac{v=2 \Rightarrow v>0}{\{v:Int \mid v=2\} <: \{v:Int \mid v>0\}}}{2 :: \{v:Int \mid v>0\}}}{\text{pred } 2 :: \{v:Int \mid v=1\}}$$

# Predecessor Example

```
pred :: n:{v:Int|v>0}->{v:Int| v=n-1}
pred n = n - 1
```

```
                  v=0 ⇒ v>0    ✗
---------------------------------------------
0::{v:Int|v=0}  {v:Int|v=0}<:{v:Int|v>0}  ✗
---------------------------------------------
          0 :: {v:Int|v>0}           ✗
---------------------------------------------
      pred 0 :: ??                   ✗
```

# Refinement Types

1991 Freeman and Pfennning

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

**Liquid Types**

Abstract Refinements

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

# Outline

Introduction

Contracts

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
```

# Max example

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

```
x:Int
y:Int
```

**x>y**

$$\frac{\begin{array}{c} x::\{v:Int \mid v=x\} \\ v=x \Rightarrow v\geq x \land v\geq y \end{array}}{x::\{v:Int \mid v\geq x \land v\geq y\}}$$

**not(x>y)**

$$\frac{\begin{array}{c} y::\{v:Int \mid v=y\} \\ v=y \Rightarrow v\geq x \land v\geq y \end{array}}{x::\{v:Int \mid v\geq x \land v\geq y\}}$$

# 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)
```

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

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

# 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**

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**

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

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

```
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**”

# 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**”

# 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 $\boxed{\text{<p :: Int -> Prop>.}}$

**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

Contracts

Liquid Types

**Abstract Refinements**

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

# Outline

Introduction

Contracts

Liquid Types

**Abstract Refinements**

- Inductive Refinements
- Indexed Refinements
- Recursive Refinements

# Outline

Introduction

Contracts

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

next acc

```haskell
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
```

loop iteration

final result

# A loop function

initial index

initial acc

```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 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:** $R(i, acc) \Rightarrow$

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

---

**Conclusion:** $R(n, \text{loop}\ f\ n\ z)$

# 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
```

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

$$R(0, z)$$

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

$$\overline{\qquad\qquad\qquad\qquad}$$

$$R(n, loop\ f\ n\ z)$$

# 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)

---

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>`

$R(i, acc) \Rightarrow$

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

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

$R(n, loop\ f\ n\ z)$

# 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
```

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

---

$R(n, loop\ f\ n\ z)$ `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

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

```
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
```

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

```
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

```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
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

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

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

```
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

Liquid Types

**Abstract Refinements**

- **Inductive Refinements**
- Indexed Refinements
- Recursive Refinements

# Outline

Introduction

Contracts

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

Liquid Types

**Abstract Refinements**

- Inductive Refinements
- **Indexed Refinements**
- Recursive Refinements

# Outline

Introduction

Contracts

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 $h_1 \wedge$ 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

$h_1 :: a$

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

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

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

# 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: List 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

Liquid Types

**Abstract Refinements**

- Inductive Refinements
- Indexed Refinements
- **Recursive Refinements**

# Outline

Introduction

Contracts

Liquid Types

**Abstract Refinements**

- Inductive Refinements
- Indexed Refinements
- Recursive Refinements

# Outline

Introduction

Contracts

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

## Verification

At run Time (Contracts)

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

Statically (Liquid Types)

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

## Abstract Refinements

Increase expressiveness without complexity

***Thank you!***