# Liquid Types

Pat Rondon
Ming Kawaguchi
Ranjit Jhala
UCSD

# Algorithmic Software Verification

# A *Really* Hard Problem

# Floyd-Hoare: Code & Property in Logic

**Invariant**

Set of Program Configurations

**Property**

Set of Acceptable Configurations

**Check: Invariant $\Rightarrow$ Property**

Automate via “SMT”

# So, what’s hard?

# Need Universally Quantified Invariants

```
else{p = table[p-1] + 1;}
```

## Prove Access Within Array Bounds

# Need Universally Quantified Invariants

More complex for lists, trees, etc.

$$\forall x\text{: } \mathtt{next}^*(\mathit{root}, x) \Rightarrow \mathit{-1} \leq x.\mathit{data}$$

# Quantifiers Kill SMT Solvers

How to Generalize and Instantiate?

# Key: Invariants Without Quantifiers

# Idea: Liquid Types

Factor Invariant to Logic x Type

**Logic**

Describes Individual Data

**Type**

Quantifies over Structure

$$\forall i: 0 \leq i < table.length \Rightarrow -1 \leq table[i]$$

Logic factored into Type

`table :: {v:int | -1 ≤ v} array`

$$\forall x\text{: next*}(root, x) \Rightarrow -1 \leq x.data$$

Logic factored into Type

$$\textbf{root ::} \{v{:}int \mid -1 \leq v\}\ list$$

**Logic**

**Theorem Prover**

Describes Individual Data

Reasoning about Individual Data

**Type System**

**Type**

Quantified Reasoning about Structure

Quantifies over Structure

# Demo

# Base Types

# Collections

# Closures

# Generics

```
let rec ffor l u f =
  if l < u then (
    f l;
    ffor (l+1) u f
  )
```

**Template of f**

**Type of f**

$\{v{:}\mathit{int} \mid X_1\} \rightarrow \mathit{unit}$

**Solution**

$X_1 = l \leq v \wedge v < u$

**l Flows Into Input of f**

**Liquid Type of f**

$l<u$ $\quad \{v{:}\mathit{int} \mid v=l\}$ $\quad \{v{:}\mathit{int} \mid X_1\}$

$\{v{:}\mathit{int} \mid l \leq v \wedge v<u\} \rightarrow \mathit{unit}$

**Reduces to**

$l<u \wedge v=l \Rightarrow X_1$

**Base Types**

**Collections**

(Structure)

**Closures**

**Generics**

```
let vs = H.mem t k ? H.find t k : [] in
H.add t k (v::vs)
```

```
let vs = H.mem t k ? H.find t k : [] in
H.add t k (v::vs)
```

# Solution

$X_1 = 0 < \text{len } v$

$X_2 = 0 \leq \text{len } v$

# Templates

# Types

t : ('a, {v:'b list | $X_1$}) H.t

vs : {v:'b list | $X_2$}

{v:'b list | $X_1$} <: {v:'b list | $X_2$}

$X_1 \Rightarrow X_2$

{v:'b list | len v = 0} <: {v:'b list | $X_2$}

len v = 0 $\Rightarrow X_2$

vs : {$X_2$} |- {len v = len vs + 1} <: {$X_1$}

$X_2$[vs/v] $\wedge$ len v = len vs + 1 $\Rightarrow X_1$

# Liquid Type of t

t : ('a, {v:'b list | 0 < len v}) H.t

# Collections

(Data)

```
let nearest dist ctra x =
  let da = Array.map (dist x) ctra in
  [min_index da, (x, 1)]
```

## Template of Output

## Type of Output

$\{v{:}int \mid X_1\} * {'b} * \{v{:}int \mid X_2\}\ list$

$int * {'b} * int\ list$

## Solution

## Liquid Type of

da : {len v = len ctra}

|- {0 ≤ v < len da}

da : {v : 'b array |

$X_1 = 0 \le v < len\ ctra$

$X_2 = 0 < v$

min_index da : $\{v : int \mid 0 \le v \wedge v < len\ da\}$

## Reduces To

## Liquid Type of Output

$len\ da = len\ ctra \wedge 0 \le v < len\ da \Rightarrow X_1$

$len\ da = len\ ctra \wedge v = 1 \Rightarrow X_2$

x : $\{v{:}int \mid 0 \le v < len\ ctra\} * {'b} * \{v{:}int \mid 0 < v\}\ list$

Base Types

Collections

Closures

Generics

```
let min_index a =
  let min = ref 0 in
  ffor 0 (Array.length a) (fun i ->
    if a.(i) < a.(!min) then min := i
  ); !min
```

$\{X_i\} \rightarrow unit$ <: $\{0 \leq v < len\ a\} \rightarrow unit$

**Solution**

$X_i = 0 \leq v < len\ a$

**Template of f**

**Reduces To** (fun i -> ...)

$\{v{:}int \mid X_i\} \rightarrow unit$

**Liquid Type of** (fun i -> ...)

$\{0 \leq v < len\ a\}$ <: $\{X_i\}$ $unit$ <: $unit$

**Liquid Type of** ffor

**Liquid Type of** ffor 0 (len a)

$\{v{:}int \mid 0 \leq v < len\ a\} \rightarrow unit) \rightarrow unit$

$l{:}int \rightarrow u{:}int \rightarrow (\{v{:}int \mid l \leq v < u\} \rightarrow unit) \rightarrow unit$

Base Types

Collections

Closures

Generics

```
mapreduce (nearest dist ctra) (centroid plus) xs
|> List.iter (fun (i,(x,sz)) -> ctra.(i)<- div x sz)
```

## Template of mapreduce

## Type of mapreduce

$('a \to \{X_1\} * 'b * \{X_2\}\ list) \to \ldots \to \{X_1\}\ 'b * 'c\ list$

$('a \to 'b * 'c\ list) \to \ldots \to 'b * 'c\ list$

## Liquid Type of (nearest dist ya)

$'a \to \{0 \leq v < len\ ctra\} * 'a * \{0 < v\}\ list$

## Template Instantiation

## Type Instantiation

$'a$ with $'a$

$'b$ with $\{v:int \mid X_1\}$

$'c$ with $'a * \{v:int \mid X_2\}$

$'a \to \{X_1\} * 'a * \{X_2\}\ list$

$X_1 = 0 \leq v < len\ ctra$

$X_2 = 0 < v$

## Liquid Type of mapreduce Output

$\{0 \leq v < len\ ctra\} * 'a * \{0 < v\}\ list$

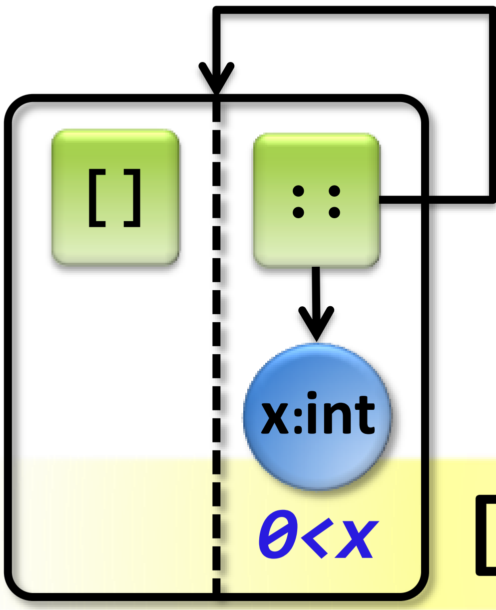
# Liquid Types

Expressive

# Piggyback Predicates on Type

1. Representation
2. Instantiation
3. Generalization

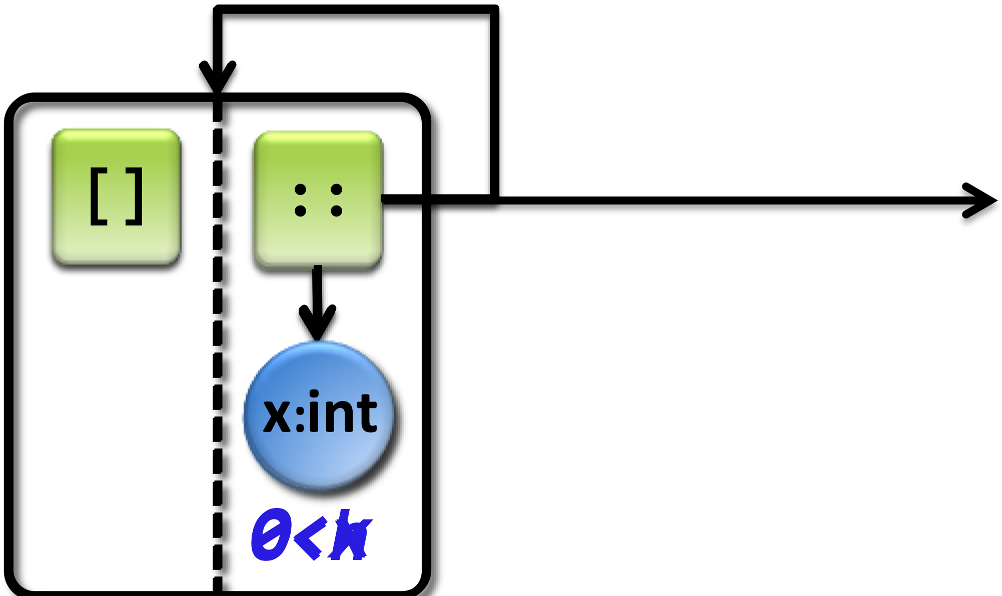
# Representation




{x:int|0<x} list

Describes all elements

# Type Unfolding




Empty

Head

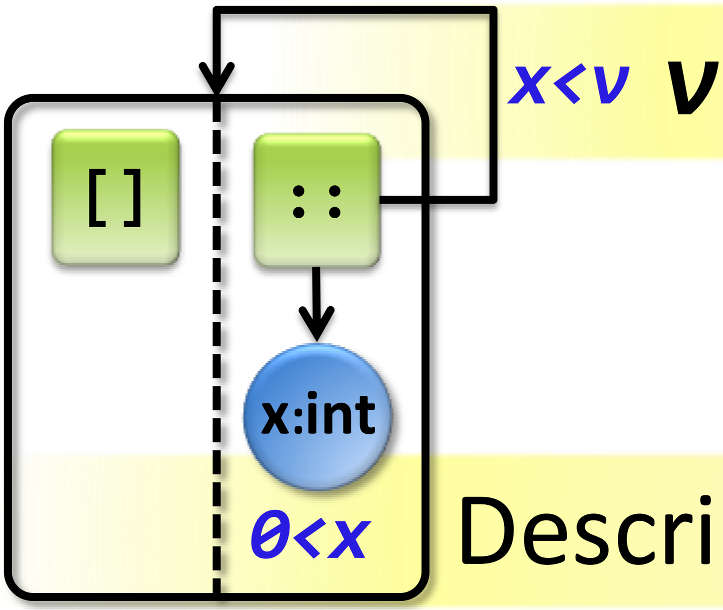
Tail

**List of positive integers**

Positive

Property holds recursively

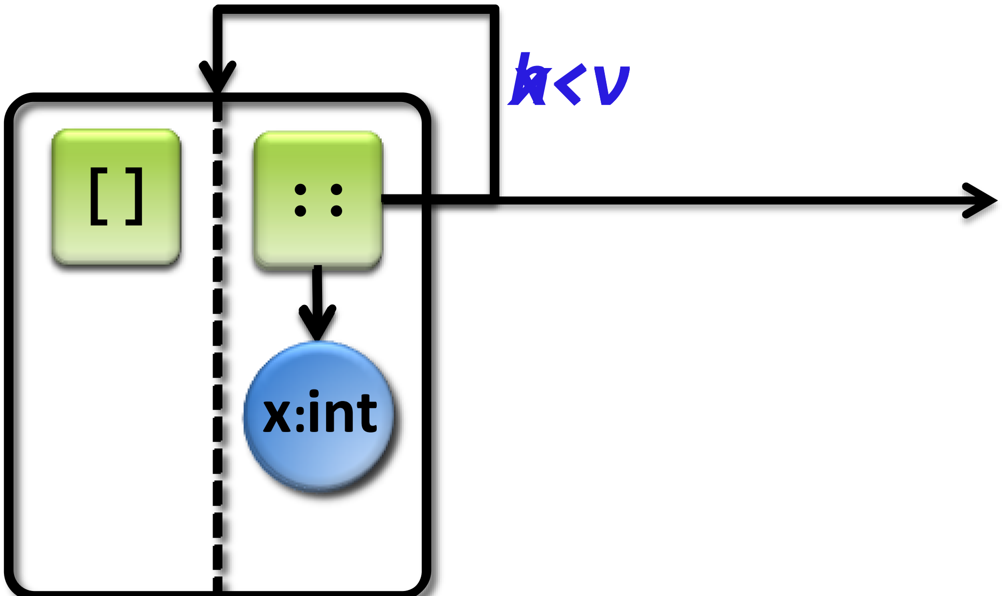
# Representation




Describes tail elements

Describes all elements

# Type Unfolding




*h<x*

**Empty**

**Head**

**Tail**

**Push Edge Property to Node**

**List of sorted integers**

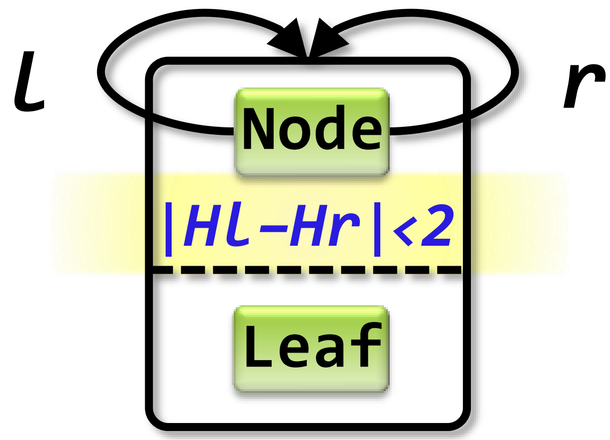
**Refinement Variable**

Elements larger than head

Property holds recursively

# Height Balanced Tree

# Height




measure H =
| Leaf = 0
| Node x l r = 1 + max (H l) (H r)

H l = Left subtree height

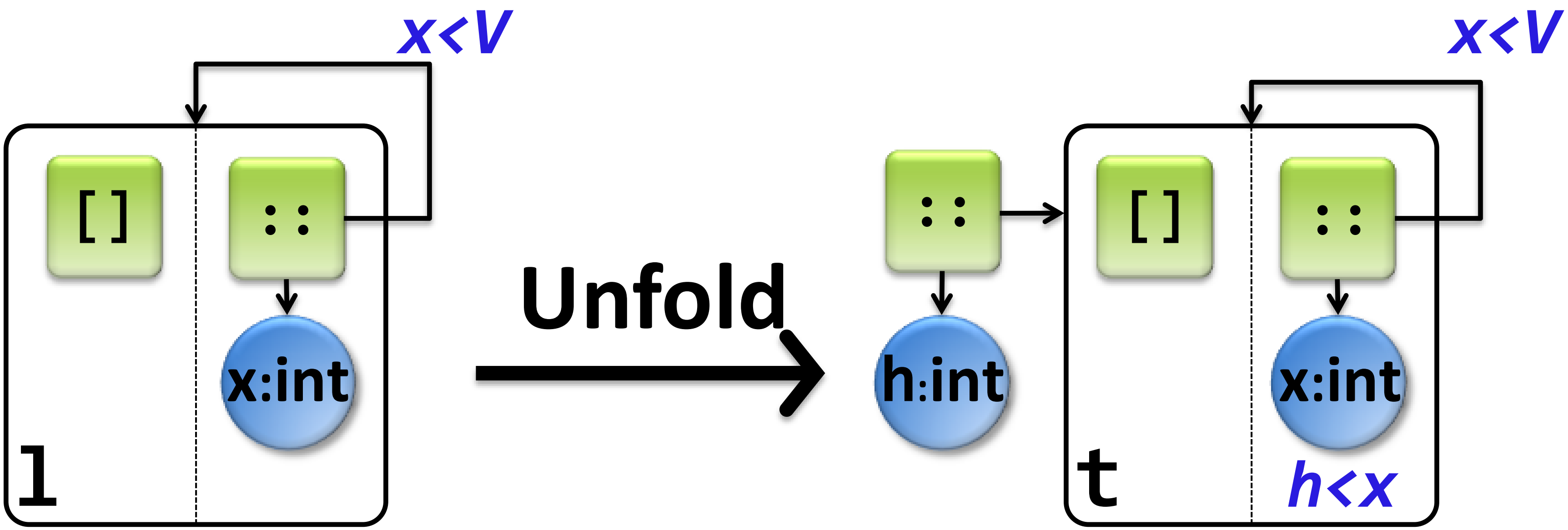
H r = Right subtree height

**Height difference bounded at each node**

# Piggyback Predicates on Type

1. Representation
2. Instantiation
3. Generalization

# Quantifier Instantiation




match **l** with **h::t**

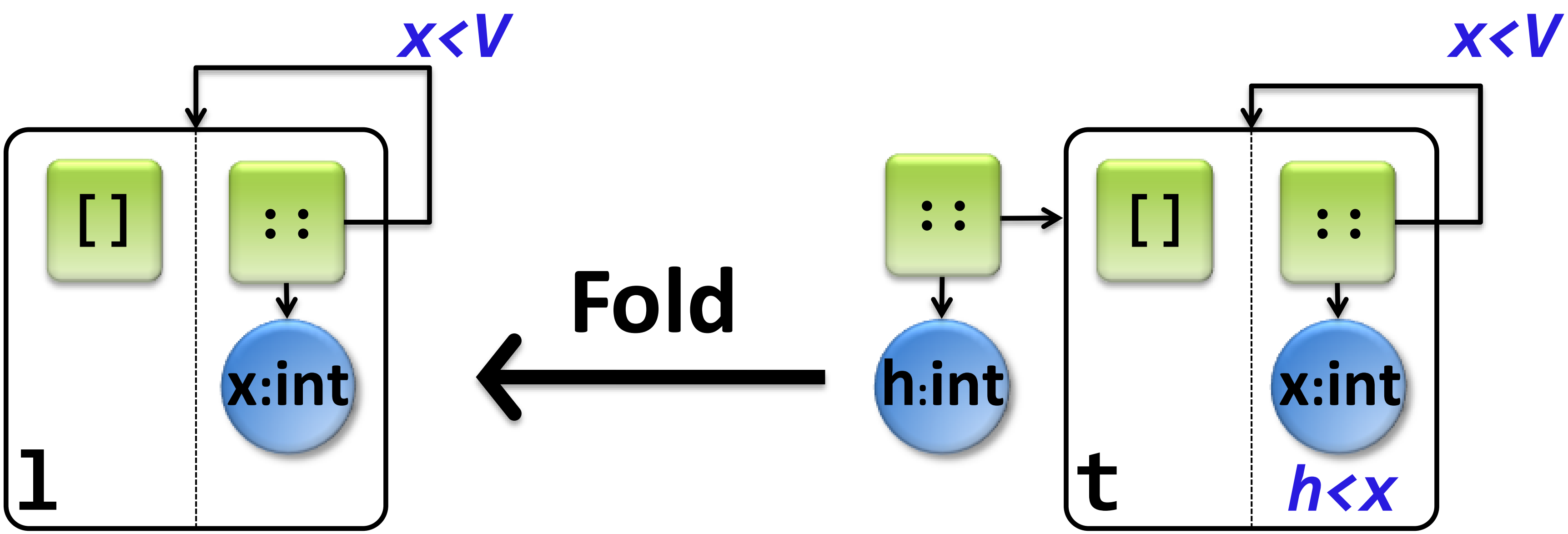
l:sorted list ⟶ h:int t:sorted list

**Instantiate** & {h<x} list

# Piggyback Predicates on Type

1. Representation
2. Instantiation
3. Generalization

# Quantifier Generalization




let l = **h::t** in

l:sorted list ⟵ h:int t:sorted list

& {h<x} list

**Generalize**

# Liquid Types

## Automatic Liquid Type Inference

By Predicate Abstraction

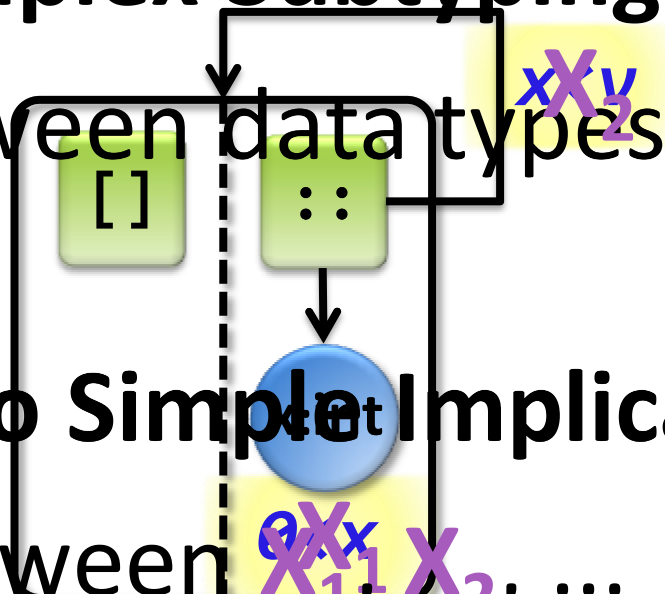
Expressive

Automatic

# Automatic Liquid Type Inference

## Complex Subtyping

Between data types




## Reduces To Simple Implications

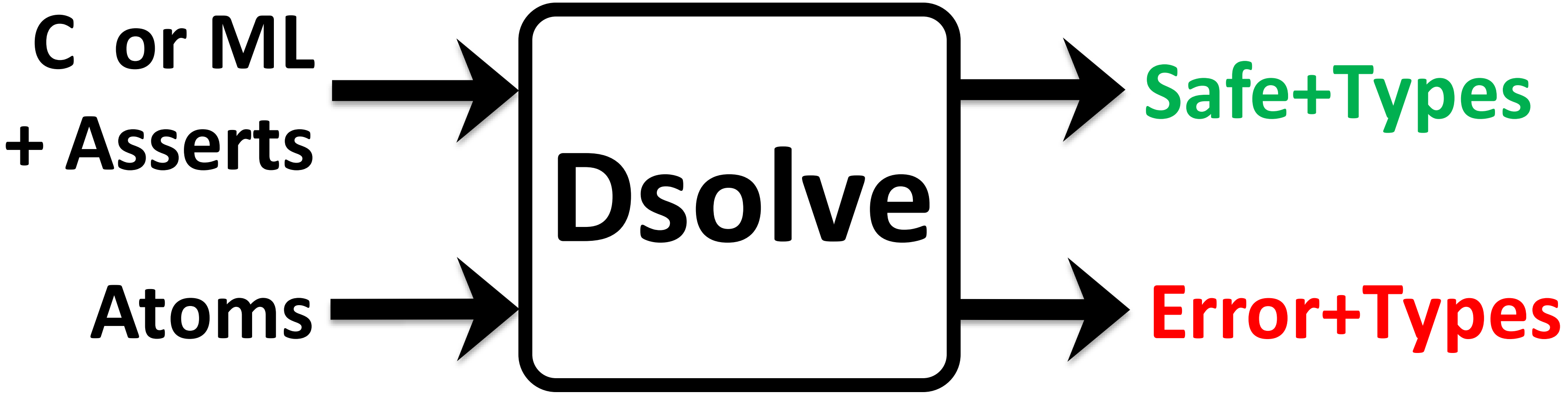
Between $X_1$, $X_2$, ...

## Let $X_1, X_2, \ldots$ = Unknown Predicates

Over atoms $0<x$, $x<v$, ...

# Demo

# Results



| Program (ML) | Verified Invariants |
|---|---|
| **List-based Sorting** | *Sorted, Outputs Permutation of Input* |
| **Finite Map** | *Balance, BST, Implements a Set* |
| **Red-Black Trees** | *Balance, BST, Color* |
| **Stablesort** | *Sorted* |
| **Extensible Vectors** | *Balance, Bounds Checking, …* |
| **Binary Heaps** | *Heap, Returns Min, Implements Set* |
| **Splay Heaps** | *BST, Returns Min, Implements Set* |
| **Malloc** | *Used and Free Lists Are Accurate* |
| **BDDs** | *Variable Order* |
| **Union Find** | *Acyclicity* |
| **Bitvector Unification** | *Acyclicity* |

# Memory Safety of C Programs

| Program (C) | Lines | Data Structures Used |
|---|---|---|
| **stringlists** | 72 | Arrays, Linked Lists |
| **strcpy** | 77 | Arrays |
| **adpcm** | 198 | Arrays |
| **pagemap** | 250 | Arrays, Linked Lists |
| **mst** | 309 | Arrays, Linked Lists, Graphs |
| **power** | 620 | Arrays, Linked Lists, Graphs |
| **ks** | 650 | Arrays, Linked Lists |
| **ft** | 742 | Arrays, Graphs |

# Take Home Lessons

**Why is checking code hard?**

Universally quantified invariants

**How to avoid quantifiers?**

Factor invariant into liquid type

**How to compute liquid type?**

SMT + Abstraction over atoms

# Much Work Remains…

**"Back-End" Logic**

Constraint Solving

Rich Decidable Logics

Qualifier Discovery…

# Much Work Remains…

**"Front-End" Types**

Destructive Update

Concurrency

Objects & Classes

Dynamic Languages…

# Much Work Remains…

**User Interface**

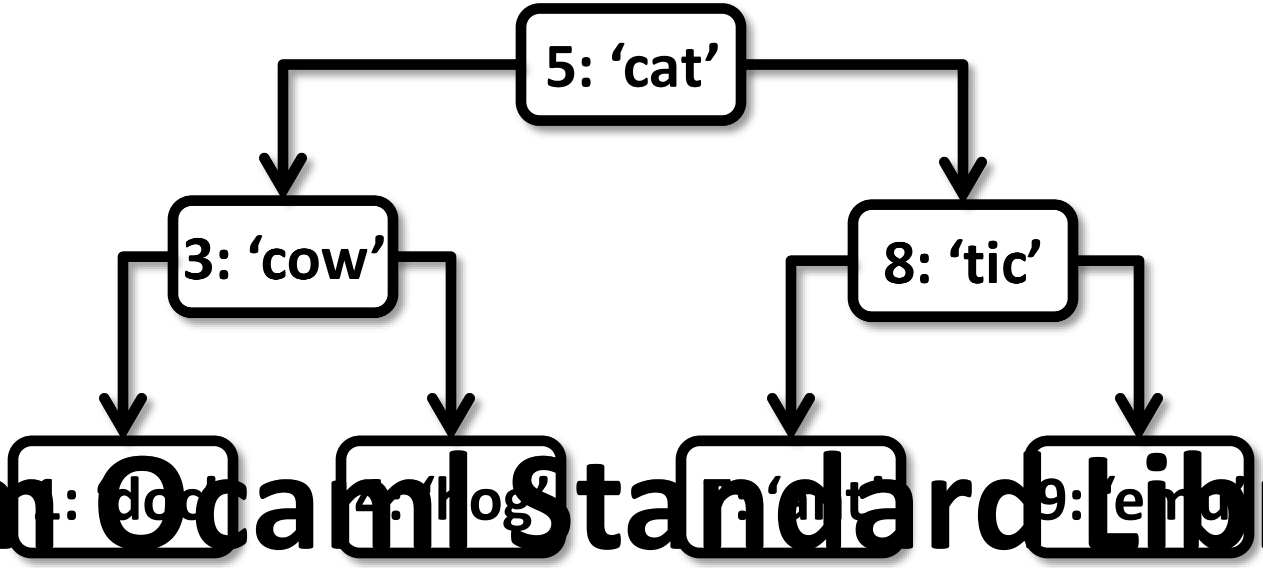
The smarter your analysis,

the harder to tell *why* it fails!

# http://goto.ucsd.edu/liquid

**source, papers, demo, etc.**

# Finite Maps (ML)




## From Ocaml Standard Library

## Verified Invariants

## Implemented as AVL Trees

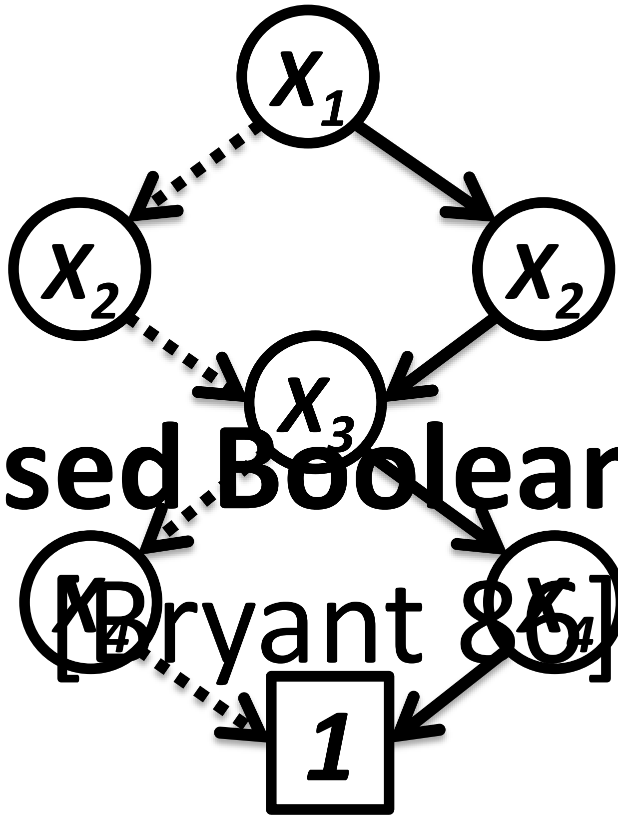
*Binary Search Ordered*

*Height Balanced*

Rotate/Rebalance on Insert/Delete

*Keys Implement Set*

# Binary Decision Diagrams (ML)




**Graph-Based Boolean Formulas**

[Bryant 86]

**Efficient Formula Manipulation**

**Verified Variant**

$$X_1 \Leftrightarrow X_2 \wedge X_3 \Leftrightarrow X_4$$

Memoizing Results on Subformulas

*Variables Ordered Along Each Path*

# Vec: Extensible Arrays (317 LOC)

"Python-style" extensible arrays for Ocaml

`find`, `insert`, `delete`, `join` etc.

Efficiency via **balanced** trees

**Balanced**

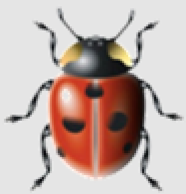
Height difference between siblings ≤ 2

**Dsolve** found balance violation

# Recursive Rebalance

vec.ml (~\Desktop) - GVIM

```
(* This is a recursive version of balance, which balances a tree all
 * the way down. The trees l and r can be of any height, but they
 * need to be internally balanced. Useful to implement concat. *)
let rec recbal l d r =
  let hl = match l with Empty -> 0 | Node(_,_,_,_,_,h) -> h in
  let hr = match r with Empty -> 0 | Node(_,_,_,_,_,h) -> h in
  if hl > hr + 2 then begin
    match l with
      Empty -> invalid_arg "Vec.bal"
    | Node(ll, _, ld, lr, _, h) ->
        if height ll >= height lr
        then bal ll ld (recbal lr d r)
        else begin
          match lr with
            Empty -> invalid_arg "Vec.bal"
          | Node(lrl, _, lrd, lrr, _, h) ->
              let nr = recbal lrr d r in
                if height nr <= height lr - 3
                then makenode ll ld (bal lrl lrd nr)
                else makenode (makenode ll ld lrl) lrd nr
        end
  end else if hr > hl + 2 then begin
"vec.ml" [unix] 383L, 11087C written                    83,60          22%
```



# Debugging via Inference

Using **Dsolve** we found

- **Where** imbalance occurred (specific path conditions)
- **How** imbalance occurred (left tree off by up to 4)

Leading to **test** and **fix**