Language-Integrated Verification

Ranjit Jhala

UC San Diego
What is Programming Languages Research?
Programming Languages Research

George Orwell
“We shall make thoughtcrime literally impossible: there will be no words to express it.” (1984)

George Orwell
Programming Languages Research

Advances in Program Analysis & Verification

SMT  Abs-Int  Dataflow  Proof-Assist.

Sym-Ex  Model-Check  Synthesis  …

Why limited impact and adoption?

(cf. Profilers, Garbage Collection, Version Control, Debuggers…)
Program Analysis & Verification

Why limited broader impact and adoption?

Programmer

Verifier
Program Analysis & Verification

Why limited broader impact and adoption?

Programmer

Verifier
Program Analysis & Verification
Why limited broader impact and adoption?

Programmer

Verifier
Program Analysis & Verification
Why limited broader impact and adoption?

Programmer
Cannot influence verification

Verifier
Cannot influence program
Language Integrated Verification

Verify *during* development
vs. hunt for bugs *after*
Language Integrated Verification

Program influences Analysis’ Abilities
Language Integrated Verification

Program influences Analysis’ Abilities

Analysis influences Program’s Design
Language Integrated Verification

Program influences Analysis’ Abilities

Analysis influences Program’s Design
Analysis influences Program’s Design

“Precision”

real-bug!

false-alarm…
Analysis influences Program’s Design

```
“Utility”

real-bug!

false-alarm…
```

Time-to-report
Analysis influences Program’s Design

1. **Late feedback:** For diffs with NPE problems, developers only received warnings at the final stage of making a code change. *This late feedback could lead to a frustrating experience:* a developer’s change could pass code review and all other checks, only to be rejected on the Submit Queue due to an easy-to-fix issue. The developer may well have moved on to another task by this point, and would have to context-switch to return to the code and address the warning.

2. **Higher latency, decreased developer productivity:** The overall latency of the Submit Queue experience increased, thereby reducing developer productivity. Since many diffs failed on the Submit Queue with NPE warnings, they needed to be submitted multiple times, increasing overall queue lengths.

**Uber Null-Pointer Checker**

[Sridharan 2017] [Ernst et al. 2010]
**Analysis influences Program's Design**

*Reporting issues sooner is better.* Google's centralized build system logs all builds and build results, so we identified all users who had seen one of the error messages in a given time window. We sent a survey to developers who recently encountered a compiler error and developers who had received a patch with a fix for the same problem. *Google developers perceive that issues flagged at compile time (as opposed to patches for checked-in code) catch more important bugs;* for example, survey participants deemed 74% of the issues flagged at compile time as "real problems," compared to 21% of those found in checked-in code. In addition, survey participants deemed 6% of the issues found at compiletime (vs. 0% in checked-in code) "critical." This result is explained by the "survivor effect";³ that is, by the time code is submitted, the errors are likely to have been caught by more expensive means (such as testing and code review). Moving as many checks into the compiler as possible is one proven way to avoid those costs.

**Building Static Analyses at Google**

[Sadowski et al. 2018]
Analysis influences Program’s Design

Scale was one thing, but an even bigger surprise was how important incrementality — dealing quickly with code changes — was for addressing the human side of the engineering problem we faced. We first tried a “batch” deployment of Infer, where it was run overnight and produced bug lists that developers might act on. Issues were presented to developers outside their workflow and the fix rate — the rate at which they fixed the issues discovered — was near 0%. Then, when we switched on Infer in an incremental-online mode, as a bot during code review, the fix rate shot up to 70%. We were stunned. Although the POPL paper had provided the techniques to enable this form of analysis, we had no idea at the time how powerful the effect of its fast incremental analysis of code changes would be.

Infer Analysis at Facebook

[Calcagno et al. 2018]
Language Integrated Verification

Program influences Analysis’ Abilities

Analysis influences Program’s Design
Program influences Analysis’ Abilities

Specification
“What to analyze”

Verification
“How to analyze”
Program influences Analysis’ Abilities

Specification
“*What to analyze*”

Verification
“*How to analyze*”

- Null Refs
- Array Bounds
- Integer Overflows
- User-def. Invariants
- Functional Correctness
Language Integrated Verification

Program influences Analysis’ Abilities

Analysis influences Program’s Design
Why LIVE
Language Integrated Verification
Plan

Why LIVE

How to LIVE

LIVE and Learn
How to LIVE

I. Specification

II. Verification

III. Proofs
I. Specification

Refining Types with Logic

- NullRefs
- Array Bounds
- Integer Overflows
- User-def. Invariants
- Functional Correctness
I. Specification

- Null Refs
- Array Bounds
- Integer Overflows
- User-def. Invariants
- Functional Correctness

Refinement Types
Refinement Types

\{ x : b \mid p \}

Value-Name Base-Type Refinement

“Set of values $x$ of type $b$ such that $p$ is true”

[Constable-Smith 1987, Rushby et al. 1998]
Refinement Types

\{ x : b \mid p \}\}

Refinements

Formulas from SMT-decidable logic
type Nat = \{ i : \text{Int} \mid 0 \leq i \} 

Integers $i$ greater than $0$
Refinement Types

Null Refs
Array Bounds
Integer Overflows
User-def. Invariants
Functional Correctness

Ex: Binary Search
Ex: Binary Search

```haskell
binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
    loop v vec 0 (length vec)

loop :: Ord a => a -> a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
    let mid = (lo + hi) `div` 2
    if v < vec!mid
        then do
            let hi' = mid - 1
            if lo <= hi'
                then loop v vec lo hi'
                else Nothing
        else if vec!mid < v
            then do
                let lo' = mid + 1
                if lo' <= hi
                    then loop v vec lo' hi
                    else Nothing
            else Just mid
```
Ex: Binary Search

```haskell
binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
  loop v vec 0 (length vec)

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
  let mid = (lo + hi) `div` 2
  if v < vec!mid
    then do
      let hi' = mid - 1
      if lo <= hi'
        then loop v vec lo hi'
        else Nothing
    else if vec!mid < v
      then do
        let lo' = mid + 1
        if lo' <= hi
          then loop v vec lo' hi
          else Nothing
    else Just mid
```

Midpoint
Ex: Binary Search

```haskell
binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
  loop v vec 0 (length vec)

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
  let mid = (lo + hi) `div` 2
  if v < vec!mid
    then do
      let hi' = mid - 1
      if lo <= hi'
        then loop v vec lo hi'
      else Nothing
    else if vec!mid < v
    then do
      let lo' = mid + 1
      if lo' <= hi
        then loop v vec lo' hi
      else Nothing
    else Just mid
```

Recurse on left
Ex: Binary Search

```haskell
binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
  loop v vec 0 (length vec)

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
  let mid = (lo + hi) `div` 2
  if v < vec!mid
    then do
      let hi' = mid - 1
          if lo <= hi'
            then loop v vec lo hi'
            else Nothing
      else Nothing
  else if vec!mid < v
    then do
      let lo' = mid + 1
          if lo' <= hi
            then loop v vec lo' hi
            else Nothing
      else Just mid
```

Recurse on right
Ex: Binary Search

```haskell
binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
    loop v vec 0 (length vec)

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
    let mid = (lo + hi) `div` 2
    if v < vec!mid
      then do
          let hi' = mid - 1
          if lo <= hi'
            then loop v vec lo hi'
            else Nothing
      else if vec!mid < v
        then do
            let lo' = mid + 1
            if lo' <= hi
              then loop v vec lo' hi
              else Nothing
        else Just mid  
```

Return midpoint
Array Bounds Specification

```haskell
binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
  loop v vec 0 (length vec)

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
  let mid = (lo + hi) `div` 2
  if v < vec!mid
    then do
      let hi' = mid - 1
      if lo <= hi'
        then loop v vec lo hi'
      else Nothing
    else if vec!mid < v
    then do
      let lo' = mid + 1
      if lo' <= hi
        then loop v vec lo' hi
      else Nothing
    else Just mid
```
Array Bounds Specification

Length

\[
\text{length} :: \forall v : \text{Vector} \ a \rightarrow \{n : \text{Nat} \mid n = \text{vlen} \ v\}
\]

Refined Output (Postcondition)

Output value equals input vector’s length
Array Bounds Specification

Index

(!) :: v:Vector a -> {i:Nat | i<vlen v} -> a

Refined Input (Precondition)

Input index between 0 and vector’s length
Array Bounds Specification

Length

\[\text{length} :: \text{v}: \text{Vector } a \rightarrow \{n:\text{Nat} | n = \text{vlen } v\}\]

Index

\[(!) :: \text{v}: \text{Vector } a \rightarrow \{i:\text{Nat} | i < \text{vlen } v\} \rightarrow a\]
Ex: Binary Search

```
binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
  loop v vec 0 (length vec)

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
  let mid = (lo + hi) `div` 2
  if v < vec!mid
    then do
      let hi' = mid - 1
      if lo <= hi'
        then loop v vec lo hi'
      else Nothing
    else if vec!mid < v
      then do
        let lo' = mid + 1
        if lo' <= hi
          then loop v vec lo' hi
        else Nothing
    else Just mid
```
Ex: Binary Search

```haskell
{-@ assume (!) :: v:Vector a -> {i:Nat | i < vlen v} -> a @-}
{-@ assume length :: v:Vector a -> {n:Nat | n = vlen v} @-}

binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
  loop v vec 0 (length vec) Empty & off-by-one

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
  let mid = (lo + hi) `div` 2
  • if v < vec!mid
    then do
      let hi' = mid - 1
      if lo <= hi'
        then loop v vec lo hi'
        else Nothing
      else Nothing
  • else if vec!mid < v
    then do
      let lo' = mid + 1
      if lo' <= hi
        then loop v vec lo' hi
        else Nothing
      else Just mid
```
Ex: Binary Search

```haskell
binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
  loop v vec 0 (length vec)

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
  let mid = (lo + hi) `div` 2
  if v < vec!mid
    then do
      let hi' = mid - 1
      if lo <= hi'
        then loop v vec lo hi'
        else Nothing
    else if vec!mid < v
      then do
        let lo' = mid + 1
        if lo' <= hi
          then loop v vec lo' hi
          else Nothing
      else Just mid
```
Refinement Types

Null Refs
Array Bounds
Integer Overflows
User-def. Invariants
Functional Correctness

Ex: Binary Search
Refinement Types

Null Refs
Array Bounds
Integer Overflows
User-def. Invariants
Functional Correctness

Ex: Binary Search
Integer Overflows

Specify Bounded Arithmetic
Specify Bounded Arithmetic

{─@ measure maxInt :: Int
  @─}

{─@ predicate Bounded N = 0 < N + maxInt && N < maxInt @─}

Define Bounded Integers

Between -maxInt and +maxInt
Specify Bounded Arithmetic

```haskell
{-@ measure maxInt :: Int @-}

{-@ predicate Bounded N = 0 < N + maxInt && N < maxInt @-}

{-@ instance BoundedNum Int where
    + :: x:Int -> y:{Int | Bounded (x + y)} -> {v:Int | v == x + y };
    - :: x:Int -> y:{Int | Bounded (x - y)} -> {v:Int | v == x - y }
@-}
```

Require Bounded Integers
For arithmetic operations
Verify with Bounded Arithmetic

binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
    if 0 < length vec then loop v vec 0 (length vec - 1)
    else Nothing

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
    let mid = (lo + hi) `div` 2
    if v < vec!mid
        then do
            let hi' = mid - 1
            if lo <= hi'
                then loop v vec lo hi
            else Nothing
        else if vec!mid < v
            then do
                let lo' = mid + 1
                if lo' <= hi
                    then loop v vec lo' hi
                else Nothing
        else Just mid

Vector Size “Fences-in” Indices
Vector Size is a Bounded Integer

import BoundedNum

binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
  if 0 < length vec then loop v vec 0 (length vec - 1)
  else Nothing

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
  let mid = (lo + hi) `div` 2
  if v < vec!mid
    then do
      let hi' = mid - 1
        if lo <= hi'
          then loop v vec lo hi'
          else Nothing
        else if vec!mid < v
          then do
            let lo' = mid + 1
              if lo' <= hi
                then loop v vec lo' hi
                else Nothing
          else Just mid
    else do
      let hi' = mid - 1
        if lo <= hi'
          then loop v vec lo hi'
          else Nothing
        else if vec!mid > v
          then do
            let lo' = mid + 1
              if lo' <= hi
                then loop v vec lo' hi
                else Nothing
          else Just mid
    else Just mid

Midpoint Can Overflow!
Midpoint Can Overflow!

“Nearly All Binary Searches and Mergesorts are Broken”
— Joshua Bloch (java.util.Arrays)
Midpoint Can Overflow!

```
import BoundedNum
{-@ using (Vector a) as {v:Vector a |BoundInt (vlen v)} @-}

binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
    if 0 < length vec then loop v vec 0 (length vec - 1)
    else Nothing

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
    let mid = (lo + hi) `div` 2
    if v < vec!mid
        then do
            let hi' = mid - 1
            if lo <= hi'
                then loop v vec lo hi'
                else Nothing
            else if vec!mid < v
                then do
                    let lo' = mid + 1
                    if lo' <= hi
                        then loop v vec lo' hi
                        else Nothing
                    else Just mid
```
I. Specification

Null Refs
Array Bounds
Integer Overflows
User-def. Invariants
Functional Correctness

Refining Types with Logic
How to LIVE

I. Specification
   Refining Types with Logic

II. Verification

III. Proofs
II. Verification

Classical Floyd-Hoare style
II. Verification
Classical Floyd-Hoare style

Techniques for Program Verification

by Greg Nelson

CSL-81-10 JUNE 1981

© 1980 by Charles Gregory Nelson

II. Verification

Classical Floyd-Hoare style

+ *Type-directed* Abstract Interpretation
Type-directed Abstract Interpretation

Code

```haskell
-- Using library function
-- sum :: [Double] -> Double

-- Compute
sumOfSquares :: Vector Double -> Double
```
Type-directed Abstract Interpretation

Code

```haskell
-- Using library function
sum :: [Double] -> Double

sumOfSquares x =
  sum [ x!i ** 2 | i <- [0 .. length x - 1] ]
```
Type-directed Abstract Interpretation

-- Using library function

```haskell
sum :: [Double] -> Double

sumOfSquares x =
    sum [ x!i ** 2 | i <- [0 .. length x - 1]]
```

Intermediate Representation

-- Using library functions

```haskell
sum :: [Double] -> Double
range :: n:Int -> m:Int -> [v:Int | n<=v && v<=m]
map :: (a -> b) -> [a] -> [b]

sumOfSquares x =
    let is = range 0 (length x - 1)
        body i = x!i ** 2
        vs = map body is
    in sum vs
```
Type-directed Abstract Interpretation

--- Using library function
sum :: [Double] -> Double

sumOfSquares x =
    sum [ x!i ** 2 | i <- [0 .. length x - 1] ]

--- Using library functions
sum :: [Double] -> Double
range :: n:Int -> m:Int -> [{v:Int | n<=v && v<=m}]
map :: (a -> b) -> [a] -> [b]

sumOfSquares x =
    let is = range 0 (length x - 1)
    body i = x!i ** 2
    vs = map body is
    in sum vs
Type-directed Abstract Interpretation

$$\text{sum} :: [\text{Double}] \rightarrow \text{Double}$$

$$\text{sumOfSquares} \; x =$$

$$\text{let is } = \text{range} \; 0 \; (\text{length} \; x - 1)$$

$$\text{body} \; i = x!i \; \ast\ast \; 2$$

$$\text{vs } = \text{map} \; \text{body} \; \text{is}$$

$$\text{in} \; \text{sum} \; \text{vs}$$
Intermediate Representation
Type-directed Abstract Interpretation

Code

Intermediate Representation

```
sumOfSquares x =
  let is = range 0 (length x - 1)
  body i = x!i ** 2
  vs = map body is
  in sum vs
```
Type-directed Abstract Interpretation

Code

Type Checking

Horn Constraints

SMT+Abs-Int

✓ ❌
Type Checking

Substitute *actuals* for *formals*

```
sumOfSquares x =
  let is = range 0 (length x - 1)
  body i = x!i ** 2
  vs = map body is
  in sum vs
```

```
range :: n:Int -> m:Int -> [{v:Int | n<=v && v<=m}]
```
Type Checking

Types

Substitute actuals for formals
Type Checking

Represent unknown refinements with $K$ variables
Type Checking

Types

```haskell
sumOfSquares x =
  let is = range 0 (length x - 1)
      body i = x!i ** 2
    vs = map body is
  in sum vs
```

`is :: [{v: Int | 0 <= v && v <= vlen x - 1}]`

`body :: {i: Int | K(i)} -> Double`
Type Checking

```
sumOfSquares x = 
  let is = range 0 (length x - 1) 
  body i = x!i ** 2
  vs = map body is 
  in sum vs
```

Types

```
is :: [{v:Int | 0 <= v && v <= vlen x - 1}]

body :: {i:Int|K(i)} -> Double

i :: {i:Int|K(i)}
```
Type Checking

\[ P \xrightarrow{\text{Flows}} Q \]
Type Checking

\[ P \xrightarrow{\text{Flows}} Q \]

Induces a Set Constraint

\[ P \subseteq Q \]

Type Checking

\[ P \xrightarrow{\text{Flows}} Q \]

Induces a *Horn* Constraint

\[ P \Rightarrow Q \]

[Flanagan-Knowles 2007, Rondon et al. 2008]
Type Checking

```
sumOfSquares x =
  let is = range 0 (length x - 1)
  body i = x!i
  vs = map body is
  in sum vs
```

Flows

- $i \xrightsquigarrow flowsto index x$ (1)
- $is \xrightsquigarrow flowsto body$ (2)
Type Checking

sumOfSquares x =

let is = range 0 (length x - 1)
body i = x!i
vs = map body is
in sum vs

is :: [{v: Int | 0 <= v && v <= vlen x - 1}]
body :: {i: Int | K(i)} -> Double
i :: {i: Int | K(i)}

Horn Constraints

K(i) \Rightarrow 0 \leq i \land i < vlen x \quad (1)

0 \leq i \land i \leq vlen x - 1 \Rightarrow K(i) \quad (2)
Type Checking

\[
\text{sumOfSquares } x = \\
\begin{cases}
\text{let is } &= \text{range } 0 \text{ (length } x - 1) \\
\text{body } i &= x!i \times 2 \\
vs &= \text{map body is} \\
\text{in sum vs}
\end{cases}
\]

\[
\text{is } : [\{v: \text{Int} \mid 0 \leq v \land v \leq \text{vlen } x - 1\}] \\
\text{body } : \{i: \text{Int} \mid K(i)\} \rightarrow \text{Double} \\
i : \{i: \text{Int} \mid K(i)\}
\]

Horn Constraints

\[
K(i) \Rightarrow 0 \leq i \land i < \text{vlen } x \tag{1}
\]

\[
0 \leq i \land i \leq \text{vlen } x - 1 \Rightarrow K(i) \tag{2}
\]
Type Checking

Horn Constraints

\[ K(i) \Rightarrow 0 \leq i \land i < vlen \ x \quad (1) \]

\[ 0 \leq i \land i \leq vlen \ x - 1 \Rightarrow K(i) \quad (2) \]
SMT + Abs-Int*

Horn Constraints

\[ K(i) \Rightarrow 0 \leq i \land i < vlen \ x \]
\[ 0 \leq i \land i \leq vlen \ x - 1 \Rightarrow K(i) \]

Solution

\[ K(v) \triangleq 0 \leq v \land v \leq vlen \ x - 1 \]

*Fixpoint computed using SMT [Flanagan-Leino 2000]
Type-directed Abstract Interpretation

Code

Type Checking

Horn Constraints

SMT+Abs-Int

✓  X
Type-directed Abstract Interpretation

- Code & Spec
  - Type Checking
    - Horn Constraints
      - SMT+Abs-Int
        - ✔
        - ✗
- Null Refs
- Array Bounds
- Integer Overflows
- User-defined Invariants
- Functional Correctness
User-defined Invariants

Interval Sets

Data Structure to Represent Sets

[Breitner 2017]
Data Structure to Represent Sets

\{ 7, 1, 10, 3, 11, 2, 9, 12, 4 \}
Data Structure to Represent Sets

\[
\{ 7 \ 1 \ 10 \ 3 \ 11 \ 2 \ 9 \ 12 \ 4 \}
\]

By Ordering into a sequence…

\[
[1 \ 2 \ 3 \ 4 \ 7 \ 9 \ 10 \ 11 \ 12]
\]
Data Structure to Represent Sets

\{ 7 \ 1 \ 10 \ 3 \ 11 \ 2 \ 9 \ 12 \ 4 \ \}

By \textbf{Ordering} into a sequence…

\[
[ 1 \ 2 \ 3 \ 4 \ 7 \ 9 \ 10 \ 11 \ 12 ]
\]

and \textbf{Partitioning} into Intervals

\[
[ 1-5 \ 7-8 \ 9-13 ]
\]
Interval Sets

\[ [ \text{1-5} \quad \text{7-8} \quad \text{9-13} ] \]

Represented as a Data Type

data ItvSet a
  = Empty
  \ |
  , I \{ from :: a
  , to :: a
  , rest :: ItvSet a
  \}
Interval Sets

\[ [1-5 \ 7-8 \ 9-13] \]

Represented as a Data Type

data ItvSet a
  = Empty
  | I { from :: a
       , to :: a
       , rest :: ItvSet a
     }

“Head”
Interval
Interval Sets

\[ [1-5 \ 7-8 \ 9-13] \]

Represented as a Data Type

```haskell
data ItvSet a
    = Empty
    | I { from :: a
        , to :: a
        , rest :: ItvSet a
    }
```

“Tail”
Interval Set
Interval Sets

Invariants: Refined Data Type

data ItvSet a
    = Empty
    | I { from :: a
        , to :: {v:a | from < v}
        , rest :: ItvSet {v:a | to <= v}
    }
Interval Sets

Invariants: Refined Data Type

```
data ItvSet a
  = Empty
  | I { from :: a, to :: {v:a | from < v}, rest :: ItvSet {v:a | to <= v} }
```

“Head”
Non-Empty
Interval Sets

[1-5 7-8 9-13]

Invariants: Refined Data Type

data ItvSet a
  = Empty
  | I { from :: a
       , to :: {v:a | from < v}
       , rest :: ItvSet {v:a | to <= v} }
Invariants: Refined Data Type

\[
\text{test1} = \text{I 8 5 Empty}
\]

Rejected

Not an Interval!
Invariants: Refined Data Type

\[ \text{test2} = I\ 1\ 7\ (I\ 5\ 8\ \text{Empty}) \]

Rejected
Not Disjoint!
Invariants: Refined Data Type

\textbf{test3} = \textbf{I 1 5 (I 9 13 (I 7 8 Empty))}

Rejected
Not Ordered!
Invariants: Refined Data Type

Inference makes programming pleasant…
Inference makes programming *pleasant*…

Tricky recursive Interval manipulation
Invariants: Refined Data Type

Inference makes programming pleasant …

e.g. compared to interactive proof
Invariants: Refined Data Type

Inference makes programming **pleasant**…

e.g. compared to **interactive proof**

[Breitner et al. 2017]
Type-directed Abstract Interpretation

Code & Spec

Type Checking

Horn Constraints

SMT+Abs-Int

Null Refs
Array Bounds
Integer Overflows
User-defined Invariants
Functional Correctness
Type-directed Abstract Interpretation Enables *Highly* Automated Verification

Approx. 1 line spec per 10 lines of code
Similar to standard type annotation
II. Verification

Type-directed Abstract Interpretation
How to LIVE

I. Specification
   Refining Types with Logic

II. Verification
   Type-directed Abstract Interpretation

III. Proofs
III. Proofs

NullRefs
Array Bounds
Integer Overflows
User-defined Invariants
Functional Correctness
III. Proofs

- Null Refs
- Array Bounds
- Integer Overflows
- User-defined Invariants
- Functional Correctness
Functional Correctness

A *Language-Integrated* Approach

Step 1
Define properties as *functions*

Step 2
Write proofs as *code*
Functional Correctness

\[
\text{union} :: \text{is1:}_\text{__} \rightarrow \text{is2:}_\text{__} \rightarrow \{v | \text{elts } v = (\text{elts is1}) \cup (\text{elts is2})\}
\]

\text{union} returns “correct” Interval-Set
Define properties as *functions*

**Interval Sets**

\[
[1-5 \quad 7-8 \quad 9-13]
\]
Define properties as functions

Single Interval

1-5
Define properties as functions

Single Interval has Elements

1-5 \rightarrow \{1, 2, 3, 4\}
Define properties as **functions**

Single Interval has Elements

\[ 1-5 \mapsto \{1,2,3,4\} \]

elts :: Int -> Int -> S.Set Int

```haskell
elts i j = 
    if i < j then S.add i (elts (i+1) j) 
    else S.empty
```

Define properties as functions

Reflect
Implementation as the Specification

```
elts :: i:Int -> j:Int ->
    \{ v | v = if i < j then S.add i (elts (i+1) j)
        else S.empty \}
```
Functional Correctness

A Language-Integrated Approach

Step 1
Define properties as functions

Step 2
Write proofs as code
Write proofs as *code*

*Equational* proof that \( i \leq x < j \) implies \( x \) is in \( \text{elts} \ i \ j \)
Write proofs as code

Equational proof that $i \leq x < j$ implies $x$ is in $(\text{elts } i \ j)$

```hs
{-# lemem :: i:Int -> j:Int -> x:{i <= x && x < j}
  -> {S.member x (elts i j)}

@-}
```

Specify Proposition as Type

[Howard 1980, Wadler 2015]
Write proofs as code

Equational proof that \(i \leq x < j\) implies \(x\) is in \((\text{elts } i \ j)\)

```haskell
{-@ lem_mem :: i:Int -> j:Int -> x:{i <= x && x < j}
      -> {S.member x (elts i j)}
  @-}
```
Write proofs as code

Equational proof that $i \leq x < j$ implies $x$ is in $(\text{elts } i \ j)$

```ocaml
{-@ lem_mem :: i:Int -> j:Int -> x:{i <= x && x < j}
   -> {S.member x (elts i j)}
@-}
lem_mem i j x =
  if i == x then     -- Base case: $i = x$
    S.member x (elts i j)
    === True
  else             -- Inductive case: $i < x$
    S.member x (elts i j)
    === S.member x (add i (elts (i+1) j))
    === S.member x (elts (i+1) j)
    && lem_mem (i+1) j x
    === True

Verified by SMT!
```
Write **proofs** as *code*

- Theorem
- Function
- ... application
- ... call
- Case-splits
- If-then-else
- Induction
- Recursion*

* Requires verifying termination; good (& easy!) to do anyway [Vazou et al. 2014]
Functional Correctness
A Language-Integrated Approach

Step 1
Define properties as functions

Step 2
Write proofs as code
Functional Correctness

A *Language-Integrated* Approach

Data-Serialization  Associativity

Regular-Expressions  Pointer-Arith  Distributed-Data

Splay-Trees  QuickSort  MergeSort  Information-Flow

Matrix  Heaps  Queues  Interpreter

Tensors  Red-Black  Typechecker
III. Proofs

- Null Refs
- Array Bounds
- Integer Overflows
- User-defined Invariants
- Functional Correctness
How to LIVE

I. Specification
Refine Types with Logic

II. Verification
Type-directed Abstract Interpretation

III. Proofs
Write and Synthesize Functions
Plan

Why LIVE

How to LIVE

LIVE and Learn
LIVE and Learn

Language is *Irrelevant*  
Verification is *Easy*  
Explanation is *Hard*

* ish
Language is *Orthogonal*

LIVE is Language-Independent
LIVE is Language-Independent

Functions, Data-Types, Polymorphism

Ocaml

Liquid Types *

Patrick M. Rondon  Ming Kawaguchi  Ranjit Jhala
University of California, San Diego
{prondon,mwookawa,jhala}@cs.ucsd.edu
LIVE is Language-Independent

Functions, Data-Types, Polymorphism

<table>
<thead>
<tr>
<th>Ocaml</th>
<th>Liquid Types *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell</td>
<td>Refinement Types For Haskell *</td>
</tr>
</tbody>
</table>

Niki Vazou  
Eric L. Seidel  
Ranjit Jhala  
Dimitrios Vytiniotis  
Simon Peyton-Jones  
UC San Diego  
Microsoft Research
LIVE is Language-Independent

Functions, Data-Types, Polymorphism

<table>
<thead>
<tr>
<th>Ocaml</th>
<th>Liquid Types *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell</td>
<td>Refinement Types For Haskell *</td>
</tr>
<tr>
<td>C</td>
<td>Low-Level Liquid Types *</td>
</tr>
</tbody>
</table>

Patrick Rondon  
UCSD  
prndon@cs.ucsd.edu

Ming Kawaguchi  
UCSD  
mwookawa@cs.ucsd.edu

Ranjit Jhala  
UCSD  
jhala@cs.ucsd.edu

Refinements + “Physical” Subtyping
LIVE is Language-Independent
Functions, Data-Types, Polymorphism

<table>
<thead>
<tr>
<th>Language</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ocaml</strong></td>
<td>Liquid Types *</td>
</tr>
<tr>
<td><strong>Haskell</strong></td>
<td>Refinement Types For Haskell *</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>Low-Level Liquid Types *</td>
</tr>
<tr>
<td><strong>JavaScript</strong></td>
<td>Dependent Types for JavaScript</td>
</tr>
</tbody>
</table>


Refinements + *Alias Types & Ownership*

LIVE is Language-Independent

Functions, Data-Types, Polymorphism

<table>
<thead>
<tr>
<th>Language</th>
<th>Refinements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ocaml</td>
<td>Liquid Types*</td>
</tr>
<tr>
<td>Haskell</td>
<td>Refinement Types For Haskell *</td>
</tr>
<tr>
<td>C</td>
<td>Low-Level Liquid Types *</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Dependent Types for JavaScript</td>
</tr>
<tr>
<td>TypeScript</td>
<td>Refinement Types for TypeScript</td>
</tr>
</tbody>
</table>

Panagiotis Vekris  Benjamin Cosman  Ranjit Jhala
University of California, San Diego, USA
{pvekris, blcosman, jhala}@cs.ucsd.edu

Refinements + SSA & Reference Immutability

[Cytron et al. 1991, Zibin et al. 2010]
LIVE is Language-Independent
Functions, Data-Types, Polymorphism

**Racket**

Occurrence Typing Modulo Theories

Andrew M. Kent  David Kempe  Sam Tobin-Hochstadt
Indiana University  Bloomington, IN, USA
{andmkent,dkempe,samth}@indiana.edu

**Scala**

SMT-Based Checking of Predicate-Qualified Types for Scala

Georg Stefan Schmid  Viktor Kuncak
EPFL, Switzerland
{firstname.lastname}@epfl.ch
LIVE is Language-Independent

Essential to make verification *ubiquitous*

(cf. *Profilers, Garbage Collection, Version Control, Debuggers…*)
LIVE and Learn

Language is *Irrelevant* *

Verification is *Easy* *

Explanation is *Hard*
Verification is Easy

If you align abstractions of Analysis and Program
Verification is Easy

If you align abstractions of *Analysis* and *Program*

Hard

- Invariants
- Shape Analysis
- Context Sensitivity

Easy

- Refinements on *Components*
- Refinements on *Datatypes*
- Refinements on *Polymorphism*
Verification is Easy

If you align abstractions of Analysis and Program

Hard

Invariants

Easy

Refinements on Components

∀x ∈ next*(1) : 0 ≤ x.data

l :: List {v:Int | 0 ≤ v}

Types decompose SMT verification to simple logics
Verification is Easy

If you align abstractions of *Analysis* and *Program*

**Hard**

Shape Analysis

“Intervals are ordered”

**Easy**

Refinements on *Datatypes*

Heap relationships *encapsulated within recursive datatype*
Verification is Easy

If you align abstractions of Analysis and Program

Hard

Context Sensitivity

Easy

Refinements on Polymorphism

\[
\begin{align*}
    x2s &= \text{map} (\lambda i \to x!i \times 2) [0..\text{length } x - 1] \\
    y2s &= \text{map} (\lambda i \to y!i \times 2) [0..\text{length } x - 1]
\end{align*}
\]

map requires different inputs at each context
Verification is *Easy*

If you *align abstractions of Analysis and Program*

**Hard**  
Context Sensitivity  

**Easy**  
Refinements on *Polymorphism*

Polymorphism yields *specialized refinements at each instance* of `map`

```latex
\begin{align*}
  x2s &= \textbf{map} (\lambda i \to x!i \times 2) [0\ldots\text{length } x - 1] \\
  y2s &= \textbf{map} (\lambda i \to y!i \times 2) [0\ldots\text{length } x - 1]
\end{align*}
```

[Henle 1993, Foster et al. 1999, Fahndrich-Rehof 2001...]

LIVE and Learn

Language is *Irrelevant* *

Verification is *Easy* *

Explanation is *Hard*

* ish
Explaination is Hard

Automation is a Frenemy
Automation is a *Frenemy*

Automation makes it hard to *explain* errors
Automation makes it hard to explain errors

Code is wrong
\textbf{Code is wrong: Binary Search}

\begin{verbatim}
{-@ assume (!) :: v:Vector a -> \{i:Nat \mid i < vlen v\} -> a @-}
{-@ assume length :: v:Vector a -> \{n:Nat \mid n = vlen v\} @-}

binarySearch :: Ord a => a -> Vector a -> Maybe Int
binarySearch v vec =
  loop v vec 0 (length vec)

loop :: Ord a => a -> Vector a -> Int -> Int -> Maybe Int
loop v vec lo hi = do
  let mid = (lo + hi) `div` 2
  \begin{itemize}
  \item if \textcolor{red}{v < vec!mid} then do
    let hi' = mid - 1
    if lo \leq hi' then loop v vec lo hi'
    else Nothing
  \item else if \textcolor{red}{vec!mid < v} then do
    let lo' = mid + 1
    if lo' \leq hi
    then loop v vec lo' hi
    else Nothing
  \end{itemize}
else Just mid
\end{verbatim}

\textbf{Off-by-one?}

Automation makes it hard to *explain errors*

*Code is wrong*

**Solution?**

Generate *counterexamples*
Automation makes it hard to *explain errors*

*Code is wrong*

*Spec is weak*

**How to *pinpoint* weak specification?**

*Which library function is to *blame?***

Automation makes it hard to explain errors

- Code is wrong
- Spec is weak
- Analysis is weird

[Erwig-Chen 2013, Zhang-Myers 2014, Naik et al 2017]
**Analysis is weird**

```haskell
sortAndCheck :: [Int] -> Bool
sortAndCheck xs = check (sort xs)

check (x1:x2:xs) = assert (x1 <= x2) && check (x2:xs)
check _        = True

sort []      = []
sort (x:xs) = insert x (sort xs)

insert :: Int -> [Int] -> [Int]
insert x []  = [x]
insert x (y:ys) = if x <= y
                    then x : y : ys
                    else y : insert x ys
```
Analysis is weird

```
sortAndCheck :: [Int] → Bool
sortAndCheck xs = check (sort xs)

check (x1:x2:xs) = assert (x1 <= x2) && check (x2:xs)
check _       = True

sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Int → [Int] → [Int]
insert x [] = [x]
insert x (y:ys) = if x <= y
    then x : y : ys
    else y : insert x ys
```

Verify output is indeed sorted
Analysis is weird

```
sortAndCheck :: [Int] -> Bool
sortAndCheck xs = check (sort xs)

check (x1:x2:xs) = assert (x1 <= x2) && check (x2:xs)
check _ = True

sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys) = if x <= y
    then x : y : ys
    else y : insert x ys
```

Verification Fails

User’s Monomorphic type blocks context-sensitivity!
Analysis is weird

sortAndCheck :: [Int] -> Bool
sortAndCheck xs = check (sort xs)
check (x1:x2:xs) = assert (x1 <= x2) && check (x2:xs)
check _ = True

sort [] = []
sort (x:xs) = insert x (sort xs)

inferred Polymorphic type enables context-sensitivity!
Analysis is weird

Programmer requires a mental model
Programmer requires a *mental model*

How to *smooth* the learning *curve*?

Utility (Expertise)

Time
Programmer requires a *mental model*

How to *smooth* the learning *curve*?

I. Data
On which errors arise *in practice*

II. Tools
To make analyses *transparent*

III. Teaching
To *integrate* verification & programming
Plan

Why LIVE

How to LIVE

LIVE and Learn
Thanks!

Dafny
Amazon

F*
MSR & INRIA

Leon
EPFL

Agda
Chalmers

Coq
INRIA

Isabelle
TU Munich

Idris
St. Andrews

...
Thanks!

P. Vekris  P. Rondon  M. Kawaguchi  A. Bakst

R. Chugh  N. Vazou  E. Seidel
refinement-types.org

Language Integrated Verification

Program influences Analysis’ Abilities

Analysis influences Program’s Design