Coverage Types: Type-Based Verification of Test Input Generators

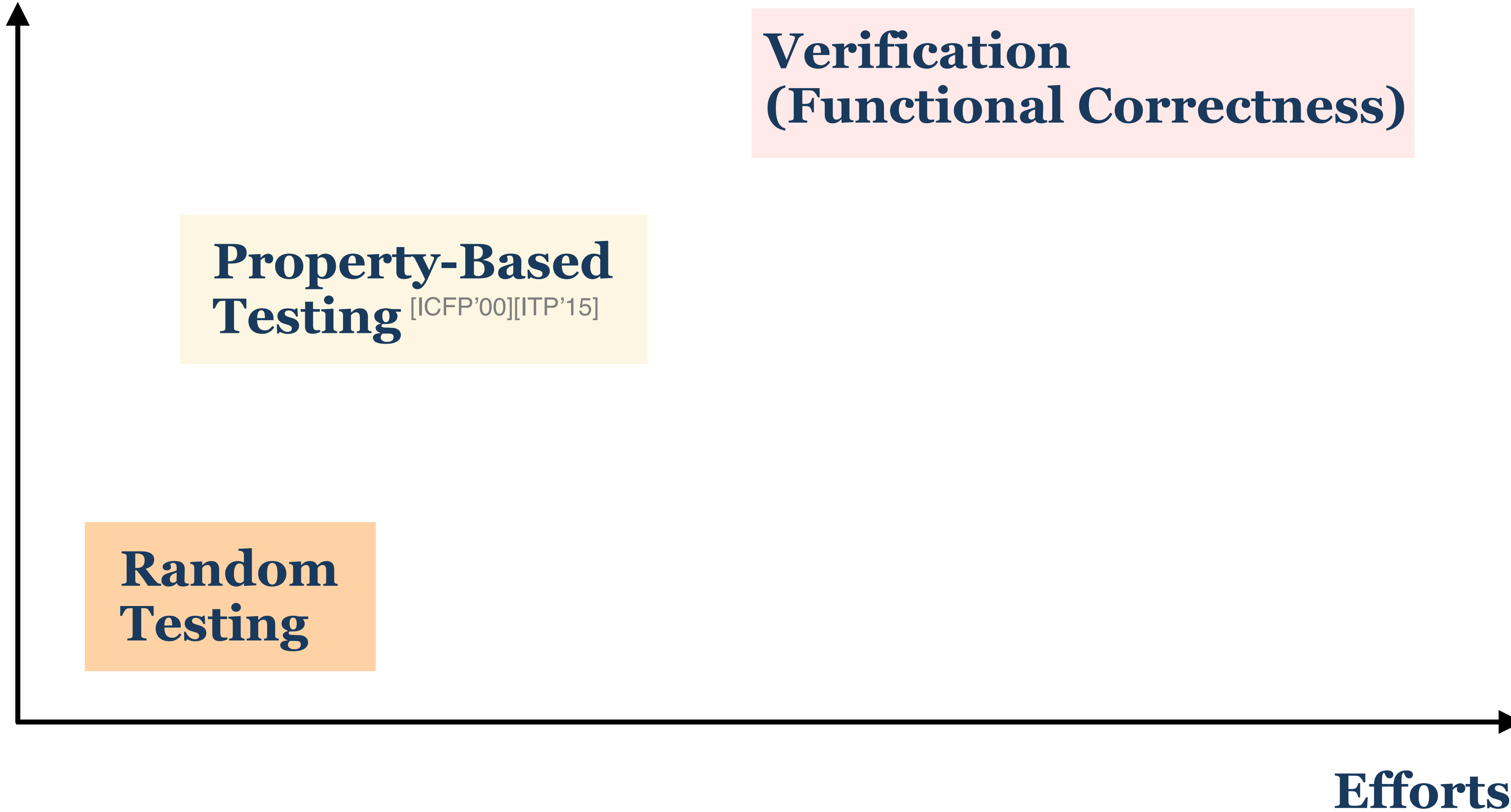
Zhe Zhou, **Ashish Mishra**, Benjamin Delaware, and Suresh Jagannathan

PURDUE UNIVERSITY
Department of Computer Science
Department of Computer Science and Engineering, Indian Institute of Technology Hyderabad

Introduction

Zhe Zhou | Ashish Misra | Benjamin Delaware | Suresh Jagannathan

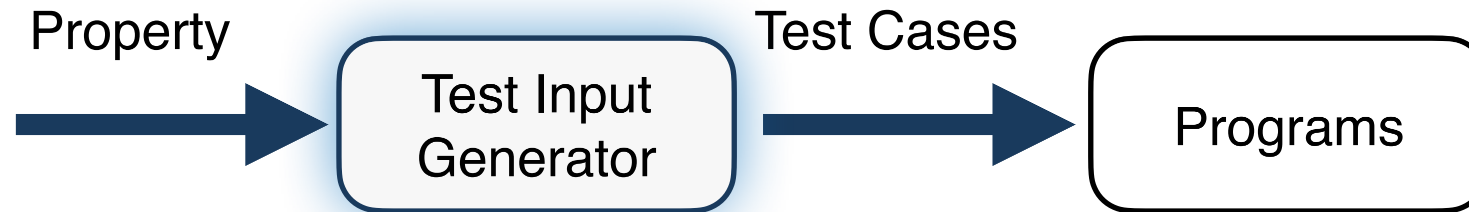
Context: Property-Based Testing



[ICFP'00] QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs
[ITP'15] Foundational Property-Based Testing

Context: Test Input Generator

Property-Based Testing



Context: Test Input Generator

Safety — Generated test cases satisfy given property.

Efficiency — Can efficiently detect bugs. [OOPSLA'19][ECOP'21][ICFP'23]

Coverage Completeness — Must generate all test cases satisfying given property. [FLOPS'14][POPL'17][POPL'18]

[OOPSLA'19] Coverage Guided, Property Based Testing
[ESOP'21] Do Judge a Test by its Cover: Combining Combinatorial and Property-Based Testing
[ICFP'23] Reflecting on Random Generation
[FLOPS'14] Generating Constrained Random Data with Uniform Distribution
[POPL'17] Beginner's Luck: A Language for Property-Based Generator
[POPL'18] Generating Good Generators for Inductive Relations

Why Coverage Is Important for Test Input Generators

Program

```
let prog (x : int * int) = match x with a, b -> a / (a - b + 1)
```

Property

```
(* precondition : Integers in the input pair >= 0 postcondition : no-division-by-zero*)
```

Test Input Generators

```
let gen1 () = (nat_gen (), nat_gen ())
```

```
let gen2 () = let x = nat_gen () in (x, x)
```

Property-Based Testing

```
prog (gen1 ())
↪* prog (2, 3)
↪* 2 / (2 - 3 + 1)
↪* 2 / 0
```

```
prog (gen2 ())
```

cannot cover (n, n + 1)

Success in finding devision-by-zero exception

Fail to find the bug **regardless how may time** we run the test

Safe test input generator may still lose **coverage**

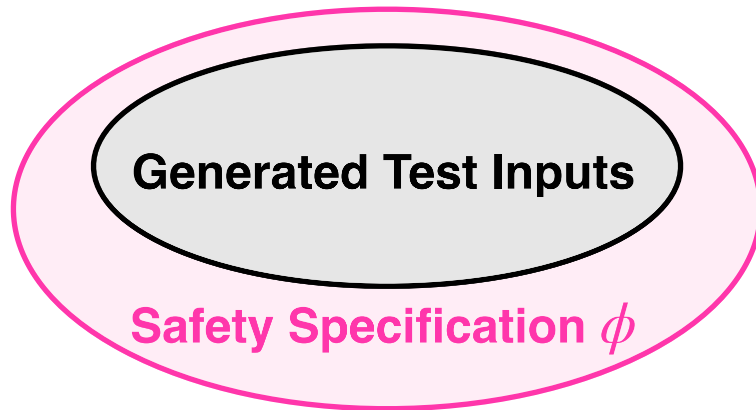
Safety v.s. Coverage Verification

ϕ : (* Integers in the input pair >= 0. *)

```
let gen2 () = let x = nat_gen () in (x, x)
```

Safety verification: test inputs **may** only be in ϕ

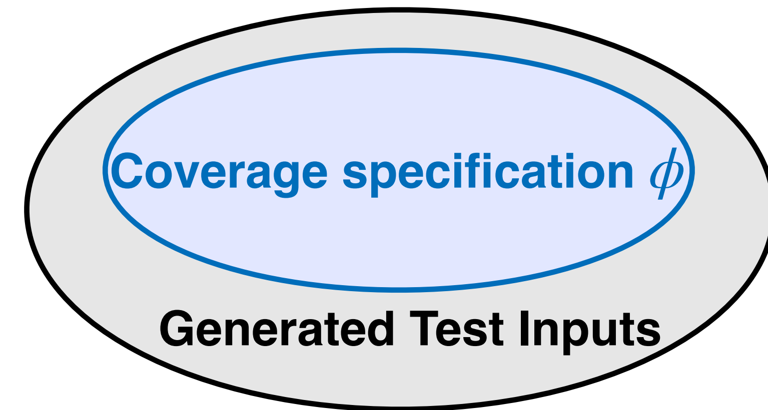
gen2 is safe



ϕ is interpreted as an **overapproximation**

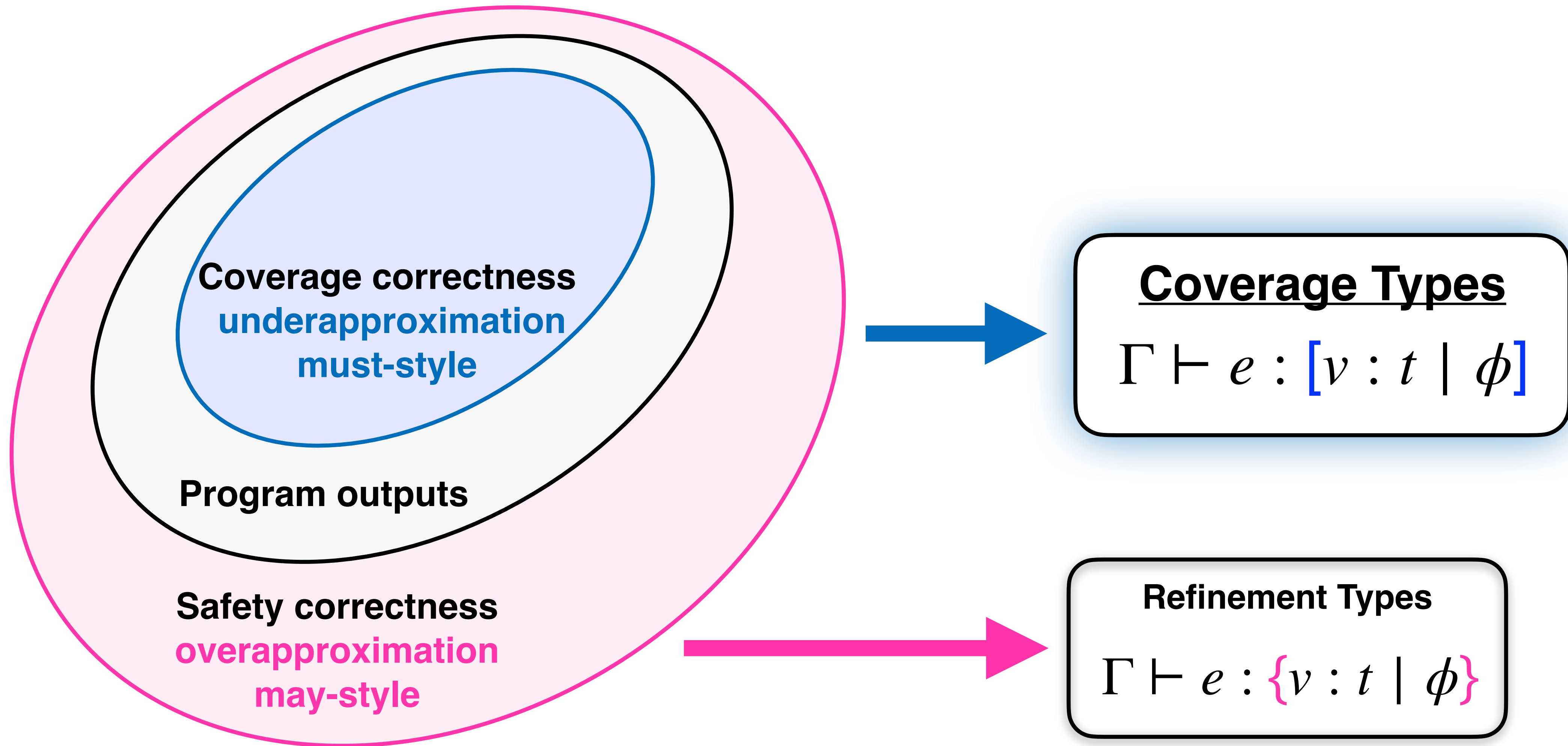
Coverage verification: test inputs **must cover** ϕ

gen2 is not coverage complete



ϕ is interpreted as an **underapproximation**

Automated Verification



Outlines

- Coverage v.s. Safety Refinement Types
- Coverage Typing
- Soundness & Algorithm & Implementation
- Evaluation

Outlines

- Coverage v.s. Safety Refinement Types
- Coverage Typing
- Soundness & Algorithm & Implementation
- Evaluation

Overview: Syntax and Semantics

Terms	`1`	$1 \oplus 2$	`nat_gen ()`	`int_gen ()`
Reduced Values $\{v_e \mid e \hookrightarrow^* v_e\}$	$\{1\}$	$\{1, 2\}$	$\{v \in \mathbb{Z} \mid v \geq 0\}$	$\mathbb{Z}$

Overview: Base Refinement Type

Refinement Type

$\{v : \mathtt{int} \mid v = 1 \lor v = 2\}$

Base types, e.g., bool, int list, …

$\{v : b \mid \phi\}$

Self reference

Qualifier

Coverage Type

$[v : \mathtt{int} \mid v = 1 \lor v = 2]$

Base types, e.g., bool, int list, …

$[v : b \mid \phi]$

Self reference

Qualifier

Overview: Base Refinement Type

Terms	1	$1 \oplus 2$	$\texttt{nat_gen}\ ()$	$\texttt{int_gen}\ ()$
Reduced Values	$\{1\}$	$\{1, 2\}$	$\{v \in \mathbb{Z} \mid v \geq 0\}$	$\mathbb{Z}$
Refinement Type	$\{v : \texttt{int} \mid v = 1\}$	$\{v : \texttt{int} \mid v = 1 \lor v = 2\}$	$\{v : \texttt{int} \mid v > 0\}$	$\{v : \texttt{int} \mid \top\}$
Coverage Type	$[v : \texttt{int} \mid v = 1]$	$[v : \texttt{int} \mid v = 1 \lor v = 2]$	$[v : \texttt{int} \mid v > 0]$	$[v : \texttt{int} \mid \top]$

Overview: Base Type

Terms	1	$1 \oplus 2$	nat_gen ()	int_gen ()
Reduced Values	$\{1\}$	$\{1, 2\}$	$\{v \in \mathbb{Z} \mid v \geq 0\}$	$\mathbb{Z}$
Refinement Type	$\{v : \texttt{int} \mid v = 1\}$	$\{v : \texttt{int} \mid v = 1 \lor v = 2\}$	$\{v : \texttt{int} \mid v > 0\}$	$\{v : \texttt{int} \mid \top\}$
Coverage Type	$[v : \texttt{int} \mid v = 1]$	$[v : \texttt{int} \mid v = 1 \lor v = 2]$	$[v : \texttt{int} \mid v > 0]$	$[v : \texttt{int} \mid \top]$

Overview: Base Type

Terms	1	$1 \oplus 2$	nat_gen ()	int_gen ()
Reduced Values	$\{1\}$	$\{1, 2\}$	$\{v \in \mathbb{Z} \mid v \geq 0\}$	$\mathbb{Z}$
Refinement Type	$\{v : \texttt{int} \mid v = 1\}$	$\{v : \texttt{int} \mid v = 1 \lor v = 2\}$	$\{v : \texttt{int} \mid v > 0\}$	$\{v : \texttt{int} \mid \top\}$
Coverage Type	$[v : \texttt{int} \mid v = 1]$	$[v : \texttt{int} \mid v = 1 \lor v = 2]$	$[v : \texttt{int} \mid v > 0]$	$[v : \texttt{int} \mid \top]$

Overview: Base Type

Terms	1	$1 \oplus 2$	`nat_gen ()`	`int_gen ()`
Reduced Values	$\{1\}$	$\{1, 2\}$	$\{v \in \mathbb{Z} \mid v \geq 0\}$	$\mathbb{Z}$
Refinement Type	$\{v : \texttt{int} \mid v = 1\}$	$\{v : \texttt{int} \mid v = 1 \lor v = 2\}$	$\{v : \texttt{int} \mid v > 0\}$	$\{v : \texttt{int} \mid \top\}$
Coverage Type	$[v : \texttt{int} \mid v = 1]$	$[v : \texttt{int} \mid v = 1 \lor v = 2]$	$[v : \texttt{int} \mid v > 0]$	$[v : \texttt{int} \mid \top]$

Overview: Base Type

Terms	1	$1 \oplus 2$	`nat_gen ()`	`int_gen ()`
Reduced Values	$\{1\}$	$\{1, 2\}$	$\{v \in \mathbb{Z} \mid v \geq 0\}$	$\mathbb{Z}$
Refinement Type	$\{v : \texttt{int} \mid v = 1\}$	$\{v : \texttt{int} \mid v = 1 \lor v = 2\}$	$\{v : \texttt{int} \mid v > 0\}$	$\{v : \texttt{int} \mid \top\}$
Coverage Type	$[v : \texttt{int} \mid v = 1]$	$[v : \texttt{int} \mid v = 1 \lor v = 2]$	$[v : \texttt{int} \mid v > 0]$	$[v : \texttt{int} \mid \top]$

Overview: Base Refinement Type

Refinement Type

Base types, e.g., bool, int list, …

$$e : \{v : b \mid \phi\}$$

Self reference

Qualifier

$$\forall v_e : b.\, e \hookrightarrow^* v_e \implies \phi[v \mapsto v_e]$$

Coverage Type

Base types, e.g., bool, int list, …

$$e : [v : b \mid \phi]$$

Self reference

Qualifier

$$\forall v_e : b.\, \phi[v \mapsto v_e] \implies e \hookrightarrow^* v_e$$

Overview: Subtyping

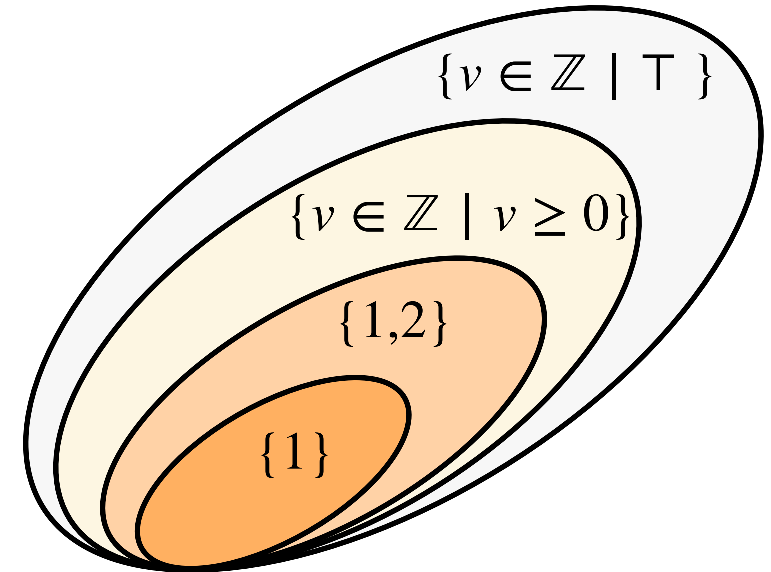
Top type is the subtype of others

$$\vdash int_gen\ () \quad : [v : int \mid \top]$$

$$<: [v : int \mid v \geq 0]$$

$$<: [v : int \mid v = 1 \lor v = 2]$$

$$<: [v : int \mid v = 1]$$



Overview: Subtyping

Coverage types **invert** the standard subtyping relationship.

$$\{1\} \subseteq \{1,2\} \subseteq \{v \in \mathbb{Z} \mid v \geq 0\} \subseteq \{v \in \mathbb{Z} \mid \top\}$$

$$[v : int \mid \top] <: [v : int \mid v \geq 0] <: [v : int \mid v = 1 \lor v = 2] <: [v : int \mid v = 1]$$ Contravariant

$$\{v : int \mid v = 1\} <: \{v : int \mid v = 1 \lor v = 2\} <: \{v : int \mid v \geq 0\} <: \{v : int \mid \top\}$$ Covariant

Intuition : Subtyping

*"We are always allowed to **weaken** an **overapproximation** (i.e., grow the set of values an expression may evaluate to), and **strengthen** an **underapproximation** (i.e., shrink the set of values an expression must evaluate to)."*

Overview: Function Type

Return type specifies the **coverage guarantee** over given parameters.

```
type tree = Leaf | Node of int * tree * tree

(* return Binary Search Trees (BSTs) *)
(* whose elements are in the exclusive range (lo, hi). *)
let rec bst_gen (lo : int) (hi : int) : int tree =…
```

Method Predicates
Uninterpreted Functions)

$bst(tree)$ $mem(tree, elem)$

Specification

$$\phi(v, \texttt{lo}, \texttt{hi}) \doteq \underbrace{bst(v)}_{\text{binary search tree}} \land \underbrace{\forall n{:}int\,.\, mem(v, n) \implies \texttt{lo} < n < \texttt{hi}}_{\text{all elements are bound by the range (lo, hi)}}$$

Coverage Return Type

$[v{:}int\ tree \mid \phi(v, \texttt{lo}, \texttt{hi})]$

Overview: Function Type

Parameter types require **safety** correctness

```
type tree = Leaf | Node of int * tree * tree

(* return Binary Search Trees (BSTs) *)
(* whose elements are in the exclusive range (lo, hi). *)
let rec bst_gen (lo : int) (hi : int) : int tree =…
```

Coverage Parameter Type $\texttt{lo}{:}[v{:}int \mid \top] \rightarrow \texttt{hi}{:}[v{:}int \mid \texttt{lo} < \texttt{hi}] \rightarrow$

Specification $\texttt{lo} < \texttt{hi}$

Need to allow any application with valid arguments

Coverage Parameter Type $\texttt{lo}{:}\{v{:}int \mid \top\} \rightarrow \texttt{hi}{:}\{v{:}int \mid \texttt{lo} < v\} \rightarrow [v{:}int\ tree \mid \phi(v, \texttt{lo}, \texttt{hi})]$

"if the inputs lo and hi are any number such that lo ≤ hi,
then the output must cover all possible BSTs whose elements are between lo and hi"

Overview: Function Type

Function types mix both **overapproximation** and **underapproximation**.

$$\texttt{lo}{:}\{v{:}int \mid \top\} \rightarrow \texttt{hi}{:}\{v{:}int \mid \texttt{lo} < v\} \rightarrow [v{:}int\ tree \mid \phi(v, \texttt{lo}, \texttt{hi})]$$

```
(* return Binary Search Trees (BSTs) *)
(* whose elements are in the exclusive range (lo, hi). *)
let rec bst_gen (lo : int) (hi : int) : int tree =…
```

Outlines

- Coverage v.s. Safety Refinement Types
- Coverage Type Context & Typing
- Soundness & Algorithm & Implementation
- Evaluation

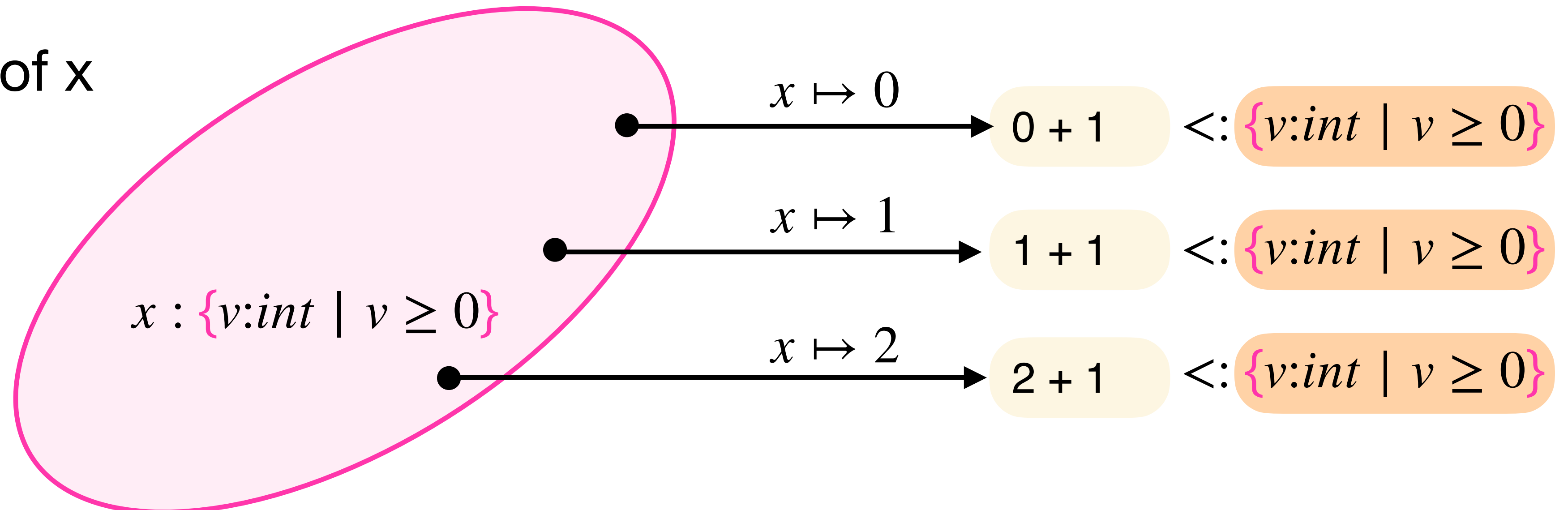
Standard Subtyping: Type Context

Bindings with **safety** refinement types are interpreted as **universal** quantifiers

(nat_gen ()) + 1 is not negative.

$x : \{v{:}int \mid v \geq 0\} \vdash \{v{:}int \mid v = x + 1\} <: \{v{:}int \mid v \geq 0\}$

For all assignments of x



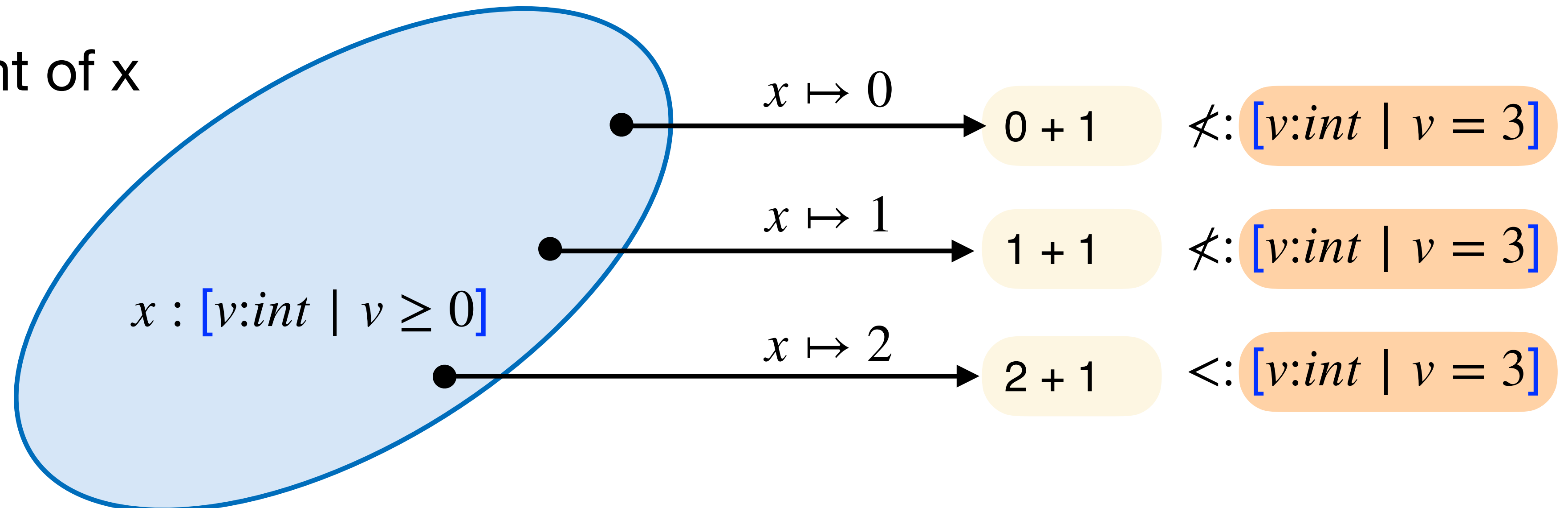
Coverage Subtyping: Type Context

Bindings with **coverage** types are interpreted as **existential** quantifiers

(nat_gen ()) + 1 covers 3.

$$x : [v{:}int \mid v \geq 0] \vdash \quad [v{:}int \mid v = x + 1] \;<:\; [v{:}int \mid v = 3]$$

Exists an assignment of x



Coverage Subtyping: Type Context

Bindings with **coverage** types are interpreted as **existential** quantifiers

(nat_gen ()) + 1 covers 3.

$x : [v{:}int \mid v \geq 0] \vdash \quad [v{:}int \mid v = x + 1] <: [v{:}int \mid v = 3]$

Exists an assignment of x

$v = 3 \implies v = x + 1$

x:$[v{:}int \mid v \geq 0]$ interpreted as $\exists x . x \geq 0 \wedge$

Verification Condition $\forall v . v = 3 \implies \exists x . x \geq 0 \wedge v = x + 1$

Coverage Typing: Function Application

```
let prog () =
  let (a : int) = int_gen () in
  let (b : int) = int_gen () in
  bst_gen a b
```

$\texttt{bst_gen} : \texttt{lo}{:}\{v{:}int \mid \top\} \to \texttt{hi}{:}\{v{:}int \mid \texttt{lo} < v\} \to \ldots$

$\texttt{int_gen} : \texttt{unit} \to [v{:}int \mid \top]$

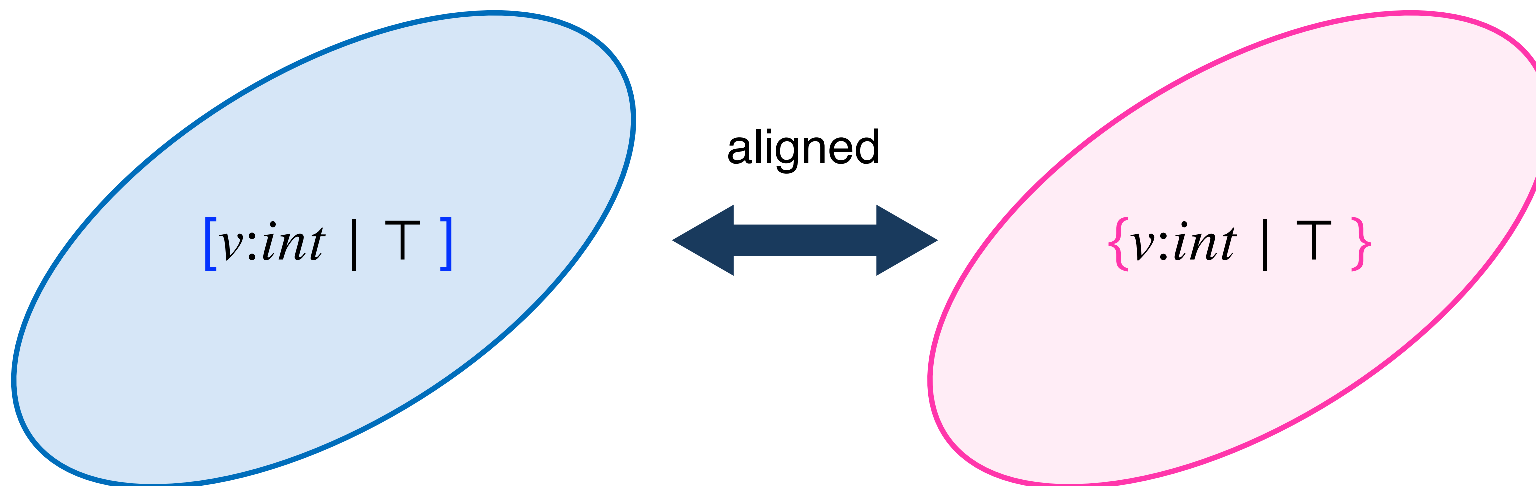
Coverage Typing: Function Application

Coverage types of arguments should be aligned with **parameter types**.

```
let prog () =
  let (a : int) = int_gen () in
  let (b : int) = int_gen () in
  let (f : int -> int) = bst_gen a in
  f b
```

$\texttt{bst_gen} : \texttt{lo}{:}\{v{:}int \mid \top\} \rightarrow \texttt{hi}{:}\{v{:}int \mid \texttt{lo} < v\} \rightarrow \ldots$

$\texttt{int_gen} : \texttt{unit} \rightarrow [v{:}int \mid \top]$



Coverage Typing: Function Application

Coverage types of arguments should be aligned with parameter types.

$$\frac{\Gamma \vdash v_x : [v{:}b \mid \phi] \qquad \Gamma \vdash e : (x{:}\{v{:}b \mid \phi\} \to \tau)}{\Gamma \vdash e\ v_x\ : \tau[x \mapsto v_x]}\ \textbf{TApp}$$

Coverage Typing: Function Application

Coverage types of arguments should be aligned with **parameter types**.

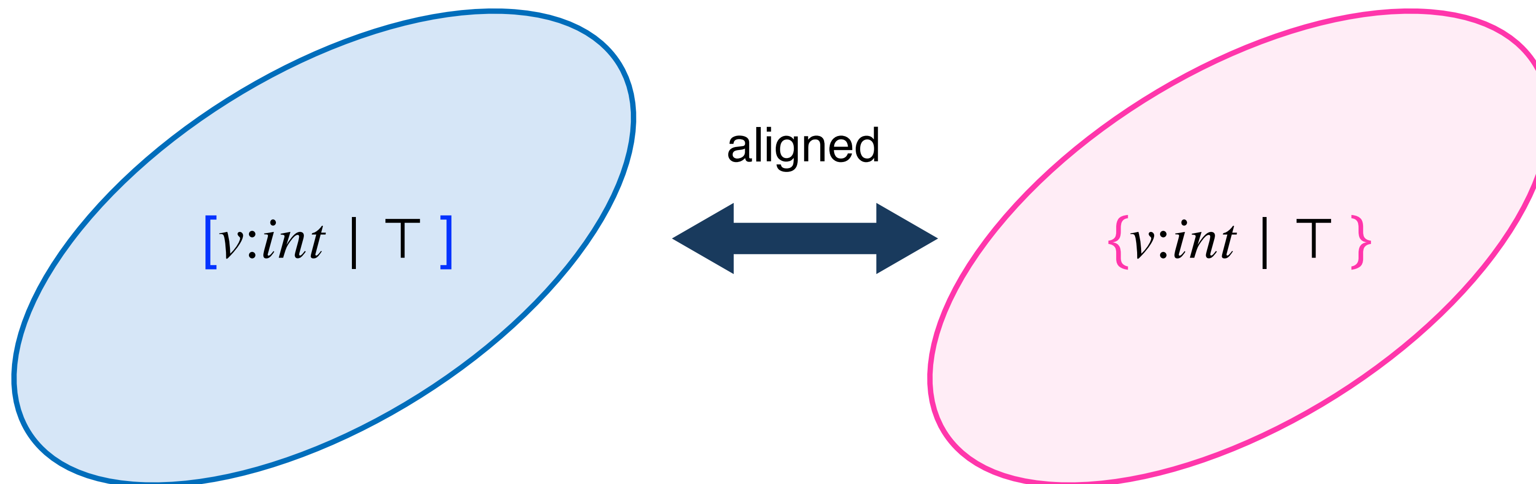
```
let prog () =
  let (a : int) = int_gen () in
  let (b : int) = int_gen () in
  let (f : int -> int) = bst_gen a in
  f b
```

$\texttt{bst_gen} : \texttt{lo}{:}\{v{:}int \mid \top\} \rightarrow \texttt{hi}{:}\{v{:}int \mid \texttt{lo} < v\} \rightarrow \ldots$

$\texttt{int_gen} : \texttt{unit} \rightarrow [v{:}int \mid \top]$

$\texttt{f} : \texttt{hi}{:}\{v{:}int \mid \texttt{a} < v\} \rightarrow \ldots$

$$\frac{\Gamma \vdash v_x : [v{:}b \mid \phi] \quad \Gamma \vdash e : (x{:}\{v{:}b \mid \phi\} \rightarrow \tau)}{\Gamma \vdash e\ v_x : \tau[x \mapsto v_x]} \textbf{ TApp}$$



Coverage Typing: Function Application

Use subtyping to align augment types with the **parameter types**.

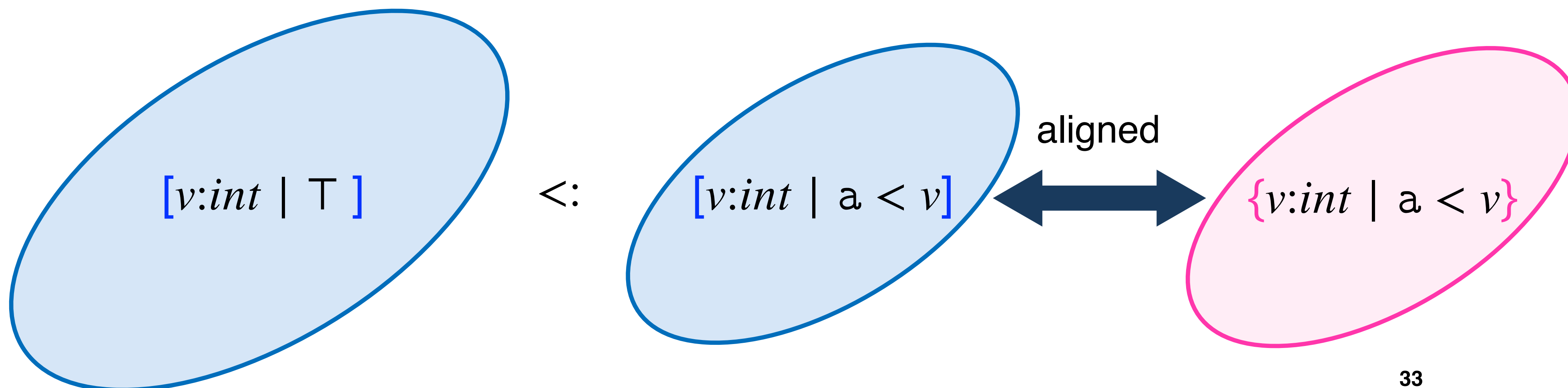
```
let prog () =
  let (a : int) = int_gen () in
  let (b : int) = int_gen () in
  let (f : int -> int) = bst_gen a in
  f b
```

$\texttt{bst_gen} : \texttt{lo}{:}\{v{:}int \mid \top\} \rightarrow \texttt{hi}{:}\{v{:}int \mid \texttt{lo} < v\} \rightarrow \ldots$

$\texttt{int_gen} : \texttt{unit} \rightarrow [v{:}int \mid \top]$

$\texttt{f} : \texttt{hi}{:}\{v{:}int \mid \texttt{a} < v\} \rightarrow \ldots$

$$\frac{\Gamma \vdash v_x : [v{:}b \mid \phi] \quad \Gamma \vdash e : (x{:}\{v{:}b \mid \phi\} \rightarrow \tau)}{\Gamma \vdash e\ v_x : \tau[x \mapsto v_x]} \ \textbf{TApp}$$



Coverage Typing: Control Flow

```
let rec bst_gen (lo : int) (hi : int) : int tree =
  if lo + 1 >= hi then Leaf
  else if bool_gen () then Leaf
  else
    let (mid : int) = int_range (lo, hi) in
    Node (mid, bst_gen lo mid, bst_gen mid hi)
```

$[v{:}int\ tree \mid \phi(v, \mathtt{lo}, \mathtt{hi})]$

bst_gen 0 0 $\hookrightarrow^*$

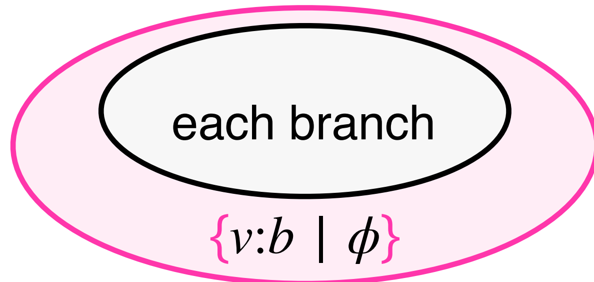
bst_gen 0 3 $\hookrightarrow^*$

Coverage Typing: Control Flow

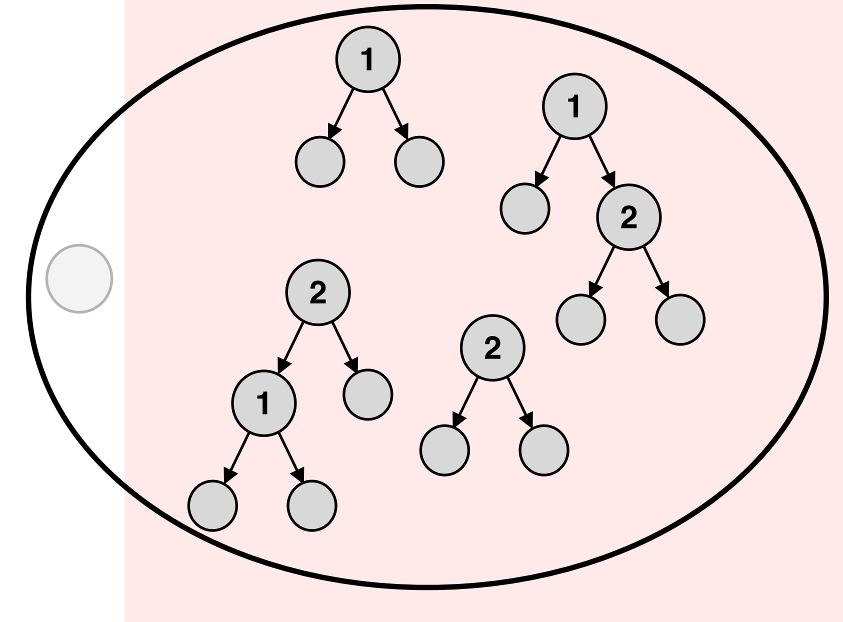
Single branch **doesn't cover** all expected values

```
let rec bst_gen (lo : int) (hi : int) : int tree =
  if lo + 1 >= hi then Leaf
  else if bool_gen () then Leaf
  else
    let (mid : int) = int_range (lo, hi) in
    Node (mid, bst_gen lo mid, bst_gen mid hi)
```

$[v{:}int\ tree \mid \phi(v, \texttt{lo}, \texttt{hi})]$



bst_gen 0 3 $\hookrightarrow^*$

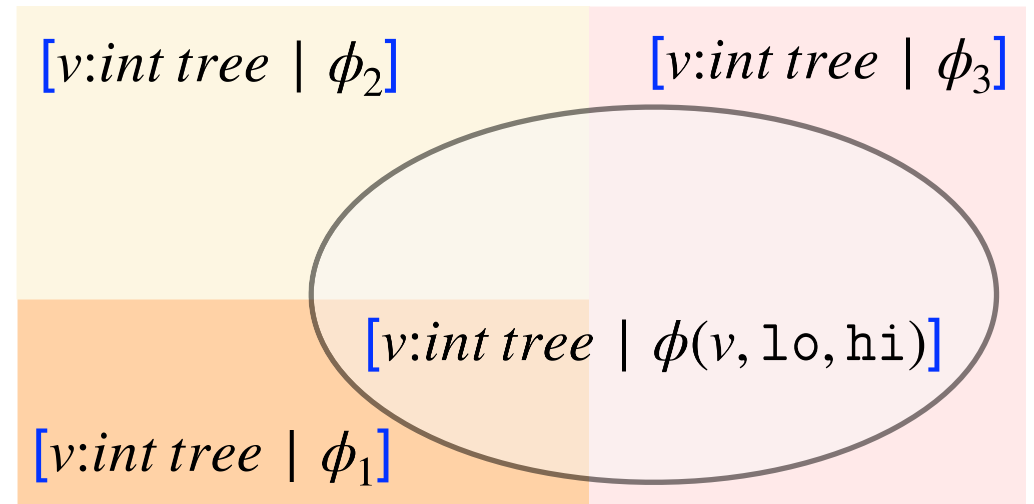


Coverage Typing: Control Flow

Need to ensure **combination** of branches cover all expected values.

```
let rec bst_gen (lo : int) (hi : int) : int tree =
  if lo + 1 >= hi then Leaf
  else if bool_gen () then Leaf
  else
    let (mid : int) = int_range (lo, hi) in
    Node (mid, bst_gen lo mid, bst_gen mid hi)
```

$$\phi \implies \phi_1 \vee \phi_2 \vee \phi_3$$



Coverage Typing: Control Flow

Type check each branch **separately**, then **merge** them together.

$$\frac{\Gamma \vdash x : [v{:}bool \mid v = \texttt{true}] \qquad \Gamma \vdash e_1[x \mapsto \texttt{true}] : \tau}{\Gamma \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : [v{:}b \mid \phi \land x = \texttt{true}]}\ \textbf{TIf1}$$

$$\frac{\Gamma \vdash x : [v{:}bool \mid v = \texttt{false}] \qquad \Gamma \vdash e_2[x \mapsto \texttt{false}] : \tau}{\Gamma \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : [v{:}b \mid \phi \land x = \texttt{false}]}\ \textbf{TIf2}$$

$$\frac{\Gamma \vdash e : [v{:}b \mid \phi_1] \qquad \Gamma \vdash e : [v{:}b \mid \phi_2]}{\Gamma \vdash e : [v{:}b \mid \phi_1 \lor \phi_2]}\ \textbf{TMerge}$$

Coverage Typing: Control Flow

Type check each branch **separately**, then **merge** them together.

```
let rec bst_gen (lo : int) (hi : int) : int tree =
  if lo + 1 >= hi then Leaf
  else if bool_gen () then Leaf
  else
    let (mid : int) = int_range (lo, hi) in
    Node (mid, bst_gen lo mid, bst_gen mid hi)
```

$[v{:}\mathit{int\ tree} \mid \mathit{emp}(v) \land \texttt{lo} + 1 \geq \texttt{hi}]$

$[v{:}\mathit{int\ tree} \mid \mathit{emp}(v) \land \texttt{lo} + 1 < \texttt{hi}]$

$$\frac{\Gamma \vdash x : [v{:}\mathit{bool} \mid v = \texttt{true}] \quad \Gamma \vdash e_1[x \mapsto \texttt{true}] : \tau}{\Gamma \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : [v{:}b \mid \phi \land x = \texttt{true}]}\ \textbf{TIf1}$$

$[v{:}\mathit{int\ tree} \mid \neg \mathit{emp}(v) \land \phi(v, \texttt{lo}, \texttt{hi}) \land \texttt{lo} + 1 < \texttt{hi}]$

Coverage Typing: Control Flow

Type check each branch **separately**, then **merge** them together.

$$\phi(v,\texttt{lo},\texttt{hi}) \doteq \underbrace{bst(v)}_{\text{binary search tree}} \land \underbrace{\forall n{:}int\,.\, mem(v,n) \implies \texttt{lo} < n < \texttt{hi}}_{\text{all elements are bound by the range (lo, hi)}}$$

$$emp(v) \implies \phi(v,\texttt{lo},\texttt{hi})$$

$$\frac{\Gamma \vdash e : [v{:}b \mid \phi_1] \qquad \Gamma \vdash e : [v{:}b \mid \phi_2]}{\Gamma \vdash e : [v{:}b \mid \phi_1 \lor \phi_2]}\ \textbf{TMerge}$$

$[v{:}int\ tree \mid emp(v) \land \texttt{lo} + 1 \geq \texttt{hi}]$

TMerge $[v{:}int\ tree \mid \phi(v,\texttt{lo},\texttt{hi})]$

$[v{:}int\ tree \mid emp(v) \land \texttt{lo} + 1 < \texttt{hi}]$

TMerge $[v{:}int\ tree \mid \phi(v,\texttt{lo},\texttt{hi}) \land \texttt{lo} + 1 < \texttt{hi}]$

$[v{:}int\ tree \mid \neg emp(v) \land \phi(v,\texttt{lo},\texttt{hi}) \land \texttt{lo} + 1 < \texttt{hi}]$

Outlines

- Coverage v.s. Safety Refinement Types
- Coverage Typing
- Soundness & Algorithm & Implementation
- Evaluation

Formalization

We have formalized a core calculus with coverage types

Type Soundness

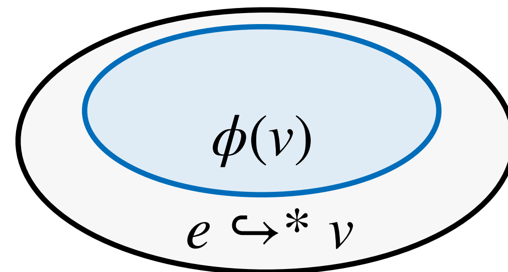
$\forall e, \varnothing \vdash e : [v{:}t \mid \phi(v)] \implies$

well-typed term e

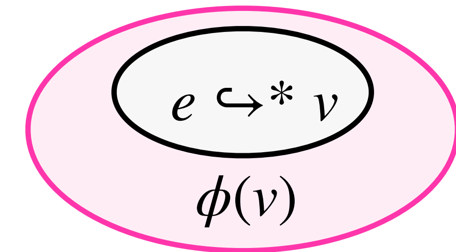
$\forall v{:}t, \quad \phi(v) \implies e \hookrightarrow^* v$

all values satisifying ϕ — can be reached from e

$e \hookrightarrow^* v \implies \phi(v)$



Coverage refinement type system



Safety refinement type system

Typing Algorithm & Implementation

Typing Algorithm

- Bidirectional typing algorithm

  Type Synthesis $\boxed{\Gamma \vdash e \Rightarrow \tau}$ Type Check $\boxed{\Gamma \vdash e \Leftarrow \tau}$

- Decidable SMT-Based Subtyping Algorithm

  SMT Queries are in Effective Propositional Logic (EPR) $\boxed{\exists^* \forall^* \phi}$

Implementation

We implement a coverage type checker **Poirot** for pure OCaml programs.

In 11K line of OCaml code

Z3 backend SMT solver

Outlines

- Coverage v.s. Safety Refinement Types
- Coverage Typing
- Soundness & Algorithm & Implementation
- Evaluation

Effectiveness Evaluation

Can **<u>Poirot</u>** verify input generators over <u>diverse datatypes</u> and <u>non-trivial properties</u>?

- **Realistic input generators**
  - **14** hand-written test input generators for PBT
    From QuickCheck[ICFP'00], QuickChick[ITP'15], CGPT[OOPSLA'19], and Elrond.[OOPSLA'21]
- **Diverse datatypes**
  Lists, trees, queues, streams, heaps, and sets.
- **Non-trivial properties**
  Normal: membership, size
  Structural: non-duplicate, sorting, complete tree, balance red black tree

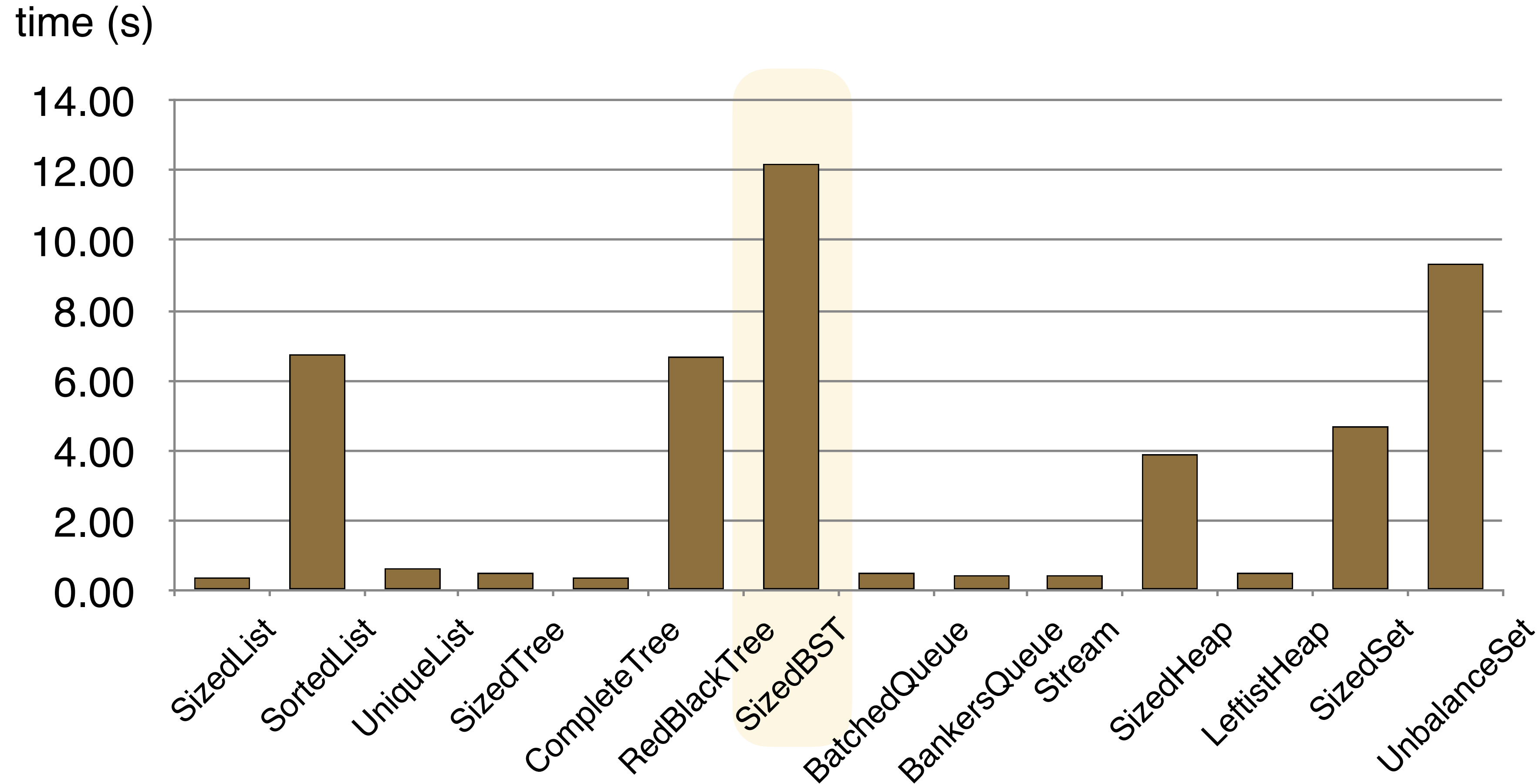
$\vdash \texttt{sized_list_gen}:$

$$\texttt{size}{:}\{v{:}int \mid \underbrace{v \geq 0}_{\text{natural number}}\} \rightarrow [v{:}int\ list \mid \underbrace{len(v) \leq \texttt{size}}_{\text{has length less equal than size}}]$$

[ICFP'00] QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs
[ITP'15] Foundational Property-Based Testing
[OOPSLA'19] Coverage Guided, Property Based Testing
[OOPSLA'21] Data-driven inference of representation invariants

Efficiency Evaluation

Can **Poirot** verify input generators in a reasonable amount of time?



SizedBST

5 branches
20 local variables

$$\phi \Longrightarrow \phi_1 \vee \phi_2 \vee \phi_3$$

Type checking time ranges from 0.35 to 12.20 seconds

Scalability Evaluation

Can **Poirot** verify complex input generators?

Simply-Typed Lambda Calculus (STLC) term generator

From QuickChick[ITP'15], consists of **13** auxiliary functions

Non-trivial inductive property

Cover all **well-typed** terms

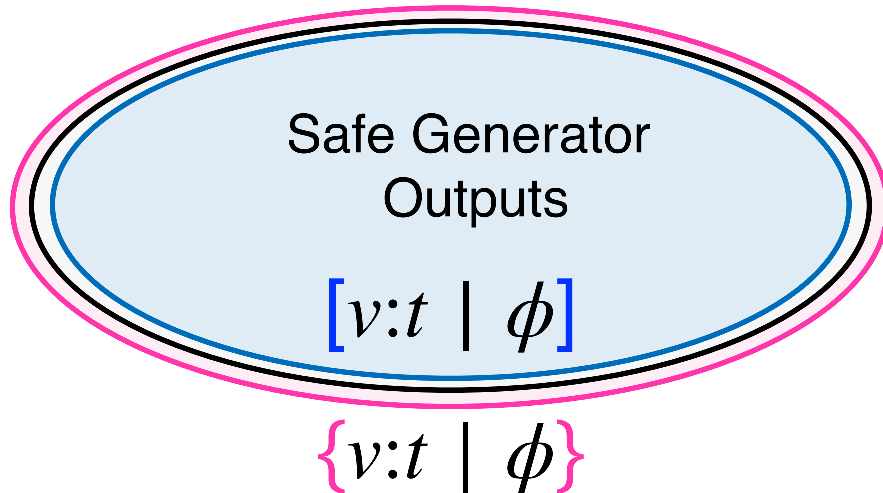
$$\vdash \texttt{gen_term_size} : \underbrace{\texttt{n}:\{v{:}int \mid 0 \le v\}}_{\text{maximum term size}} \rightarrow \underbrace{\texttt{t}:\{v{:}ty \mid \top\}}_{\text{type of term}} \rightarrow \underbrace{\Gamma:\{v{:}tyctx \mid \top\}}_{\text{typing context}} \rightarrow$$

$$\underbrace{[v{:}term \mid has_ty(\Gamma, v, t) \land app_num(v) \le n]}_{\text{result is well-typed and has at most n applications}}$$

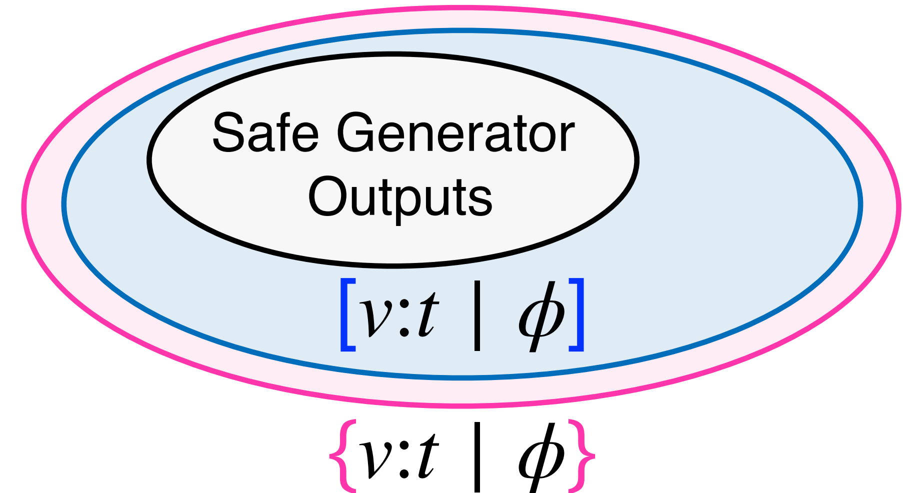
Utility Evaluation

Are safe generators always coverage complete?

Already coverage complete



Lose coverage



If safe generators are always coverage complete, we don't need Poirot

Utility Evaluation

We do need **coverage** type check

- **Benchmarks**

  Reuse PBT benchmarks

- **Synthesize multiple safe programs**
  - Cobalt [OOPSLA'22] refinement type based program synthesizer

$$\vdash \texttt{sized_list_gen} : \texttt{size}{:}\{v{:}int \mid v \geq 0\} \rightarrow \{v{:}int\ list \mid len(v) \leq \texttt{size}\}$$

- **Type check safe synthesized programs with coverage type**

$$\vdash \texttt{sized_list_gen} : \texttt{size}{:}\{v{:}int \mid v \geq 0\} \rightarrow [v{:}int\ list \mid len(v) \leq \texttt{size}]$$

76% ~ 99% safe synthesized generators **don't** pass **coverage type checking**

Future Work: Logic Foundation

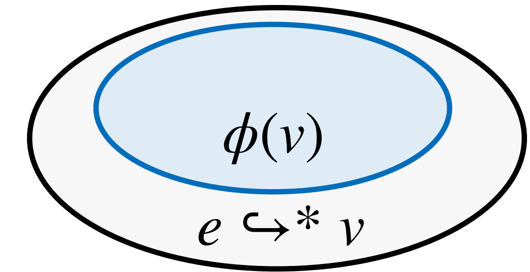
	Safety	Coverage
Program Logic	**Hoare Logic** $\{P\}\ e\ \{Q\}$	**Incorrectness Logic** $[P]\ e\ [Q]$ [POPL'19]
Types	**Refinement Types** $\Gamma \vdash e : \{v : t \mid \phi\}$	**Coverage Types** $\Gamma \vdash e : [v : t \mid \phi]$
	overapproximation	underapproximation

Summary and Thank You!

- **Coverage verification**
  - Coverage correct generators
  - Underapproximation
  - Must-style automated verification

- **Coverage types**
  - Invert subtyping relationship
  - Use overapproximate parameter type and
    underapproximate return type

- **Coverage Typing**
  - Coverage bindings are interpreted as **existential** quantifiers
  - Arguments types should be **aligned** with parameter types
  - **Combine** all branches to cover expected values



Covering All the Bases: Type-Based Verification of Test Input Generators

ZHE ZHOU, Purdue University, USA
ASHISH MISHRA, Purdue University, USA
BENJAMIN DELAWARE, Purdue University, USA
SURESH JAGANNATHAN, Purdue University, USA

Test input generators are an important part of property-based testing (PBT) frameworks. Because PBT is intended to test deep semantic and structural properties of a program, the outputs produced by these generators can be complex data structures, constrained to satisfy properties the developer believes is most relevant to testing the function of interest. An important feature expected of these generators is that they be capable of producing *all* acceptable elements that satisfy the function's input type and generator-provided constraints. However, it is not readily apparent how we might validate whether a particular generator's output satisfies this *coverage* requirement. Typically, developers must rely on manual inspection and post-mortem analysis of test runs to determine if the generator is providing sufficient coverage; these approaches are error-prone and difficult to scale as generators become more complex. To address this important concern, we present a new refinement type-based verification procedure for validating the coverage provided by input test generators, based on a novel interpretation of types that embeds "*must*-style" underapproximate reasoning principles as a fundamental part of the type system. The types associated with expressions now capture the set of values *guaranteed* to be produced by the expression, rather than the typical formulation that uses types to represent the set of values an expression *may* produce. Beyond formalizing the notion of *coverage types* in the context of a rich core language with higher-order procedures and inductive datatypes, we also present a detailed evaluation study to justify the utility of our ideas.

CCS Concepts: • **Software and its engineering** → **Language types**; **Verification and validation**.

Additional Key Words and Phrases: refinement types, property-based testing, underapproximate reasoning

ACM Reference Format:
Zhe Zhou, Ashish Mishra, Benjamin Delaware, and Suresh Jagannathan. 2023. Covering All the Bases: Type-Based Verification of Test Input Generators. *Proc. ACM Program. Lang.* 7, PLDI, Article 157 (June 2023), 24 pages.
https://doi.org/10.1145/3591271

THANK YOU

Overview: Higher-Order Function

$\vdash$ `sized_list_gen` :

`int_gen`:($\{v{:}unit \mid \top\} \to [v{:}int \mid \top]$) $\to$

`size`:$\{v{:}int \mid \underbrace{v \geq 0}_{\text{natural number}}\} \to [v{:}int\ list \mid \underbrace{len(v) \leq \texttt{size}}_{\text{has length less equal than size}}]$

Refinement types don't qualify functions directly.

Overview: Function Type Subtyping

Contravariant **Covariant**

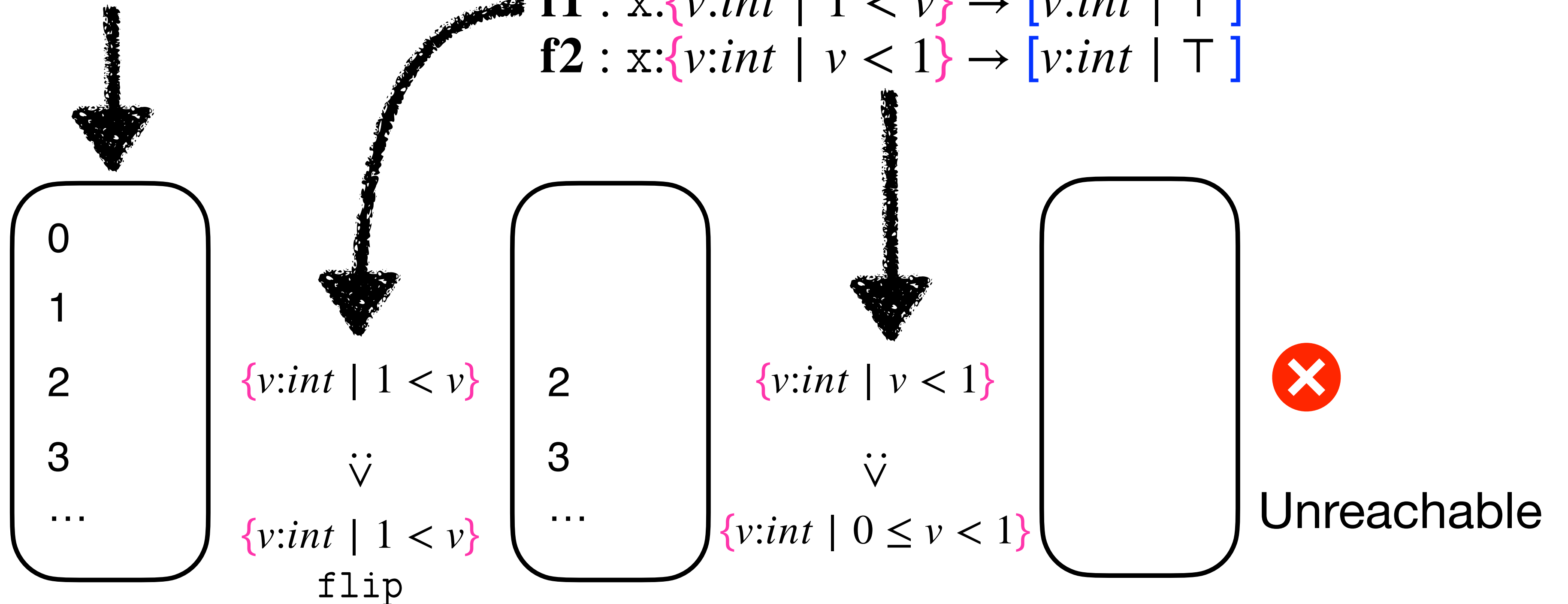
$$\frac{\Gamma \vdash \tau_{21} <: \tau_{11} \qquad \Gamma, x{:}\tau_{21} \vdash \tau_{12} <: \tau_{22}}{\Gamma \vdash x{:}\tau_{11} \rightarrow \tau_{12} <: x{:}\tau_{21} \rightarrow \tau_{22}} \textbf{ SubArr}$$

Coverage Typing: Subsumption

We cannot use subsumption rule arbitrarily.

```
let foo () =
    let (x: int) = nat_gen () in
    let _ = f1 x in
    let _ = f2 x in
    …
```

$$\frac{\emptyset \vdash e : \tau_1 \quad \emptyset \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$



flip

$x{:}[v{:}int \mid 0 \leq v] \vdash [v{:}int \mid v = x] <: [v{:}int \mid 1 < v]$ ✗

$x{:}[v{:}int \mid 0 \leq v] \vdash [v{:}int \mid v = x] <: [v{:}int \mid 0 \leq v < 1]$ ✗

Coverage Typing: Subsumption

Allow subsumption only when a coverage type is **closed.**

$$\frac{\emptyset \vdash e : \tau_1 \qquad \emptyset \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \textbf{TSub}$$

Coverage Typing: Subsumption

We cannot use subsumption rule arbitrarily.

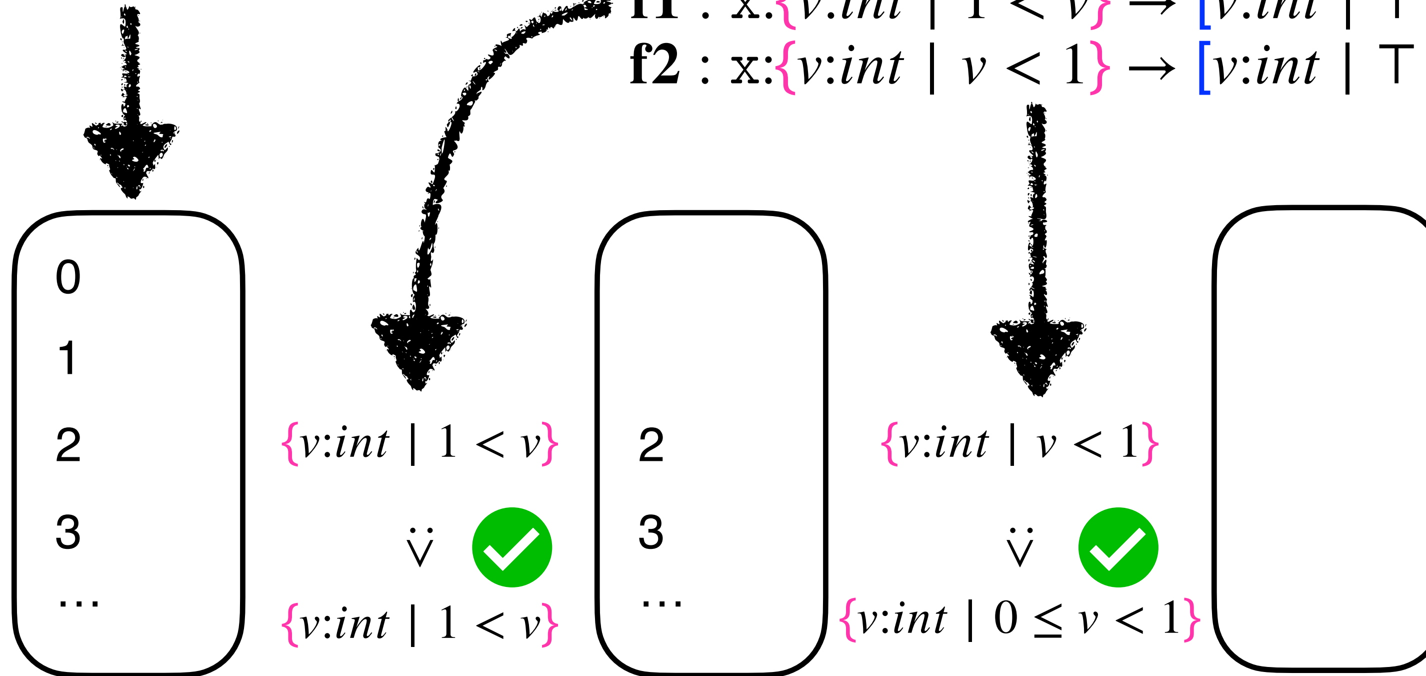
```
let foo () =
    let (x: int) = nat_gen () in
    let _ = f1 x in
    let _ = f2 x in
    ...
```

nat_gen : $\texttt{unit} \rightarrow [v{:}int \mid 0 \leq v] <: \texttt{unit} \rightarrow [v{:}int \mid 1 < v]$

f1 : $x{:}\{v{:}int \mid 1 < v\} \rightarrow [v{:}int \mid \top]$

f2 : $x{:}\{v{:}int \mid v < 1\} \rightarrow [v{:}int \mid \top]$

$$\frac{\emptyset \vdash e : \tau_1 \quad \emptyset \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$



- What are allowed?
- Is it too strict?

Coverage Typing: Subsumption

We cannot use subsumption rule arbitrarily.

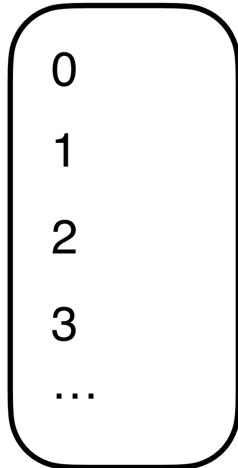
```
let foo () =
  let (x: int) = nat_gen () in
  let _ = f1 x in
  let _ = f2 x in
  …
```

$\textbf{nat_gen} : \texttt{unit} \to [v{:}int \mid 0 \leq v] \ <: \ \texttt{unit} \to [v{:}int \mid 2 < v]$

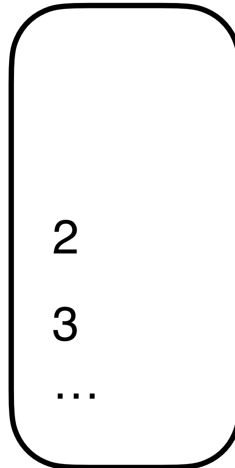
$\textbf{f1} : x{:}\{v{:}int \mid 1 < v\} \to [v{:}int \mid \top] <: x{:}\{v{:}int \mid 2 < v\} \to [v{:}int \mid \top]$

$\textbf{f2} : x{:}\{v{:}int \mid 2 < v\} \to [v{:}int \mid \top]$

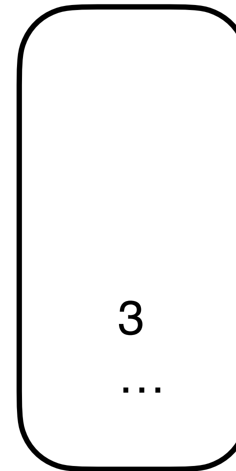
$[v{:}int \mid 0 \leq v]$



$\{v{:}int \mid 1 < v\}$

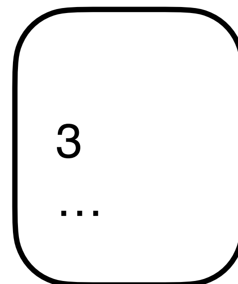


$\{v{:}int \mid 2 < v\}$

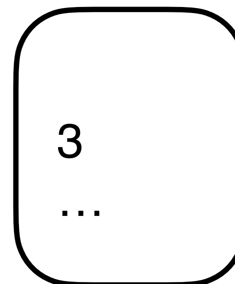


$$\frac{\emptyset \vdash e : \tau_1 \quad \emptyset \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

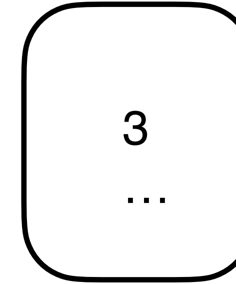
$[v{:}int \mid 2 < v]$



$\{v{:}int \mid 2 < v\}$



$\{v{:}int \mid 2 < v\}$



Coverage Typing: Type Context Invariant

Preserve the type context invariant: all coverage types are not bottom qualified.

$$\frac{\Gamma \not\vdash [v{:}b \mid \phi] <: [v{:}b \mid \bot]}{\Gamma \vdash [v{:}b \mid \phi]}\ \textbf{WF}$$

Checked by SMT solver

Coverage Typing: Control Flow

Need to ensure **combination** of branches cover all expected values.

$$\frac{\Gamma \vdash e : [v{:}b \mid \phi_1] \qquad \Gamma \vdash e : [v{:}b \mid \phi_2]}{\Gamma \vdash e : [v{:}b \mid \phi_1 \vee \phi_2]}\ \textbf{TMerge}$$

Coverage Typing: Recursion

Need to ensure **reachability**

```
let rec loop (n : nat) : int = loop n
```

$: n{:}\{v{:}int \mid \top\} \rightarrow [v{:}b \mid v = 3]$

$n{:}\{v{:}int \mid v < n\} \rightarrow [v{:}b \mid v = 3]$

Well-founded relation

$$\frac{\Gamma, x{:}\{v{:}b \mid \phi\}, f{:}(x{:}\{v{:}b \mid v \prec x \land \phi\} \rightarrow \tau) \vdash e : \tau}{\Gamma \vdash \mathtt{fix}\ f\,.\,\lambda x\,.\,e : x{:}\{v{:}b \mid \phi\} \rightarrow \tau}\ \textbf{TFix}$$