PURDUE UNIVERSITY

We’ve got you Covered:
Type-Guided Repair of Incomplete Input Generators

Patrick LaFontaine

Zhe Zhou

Ashish Misra

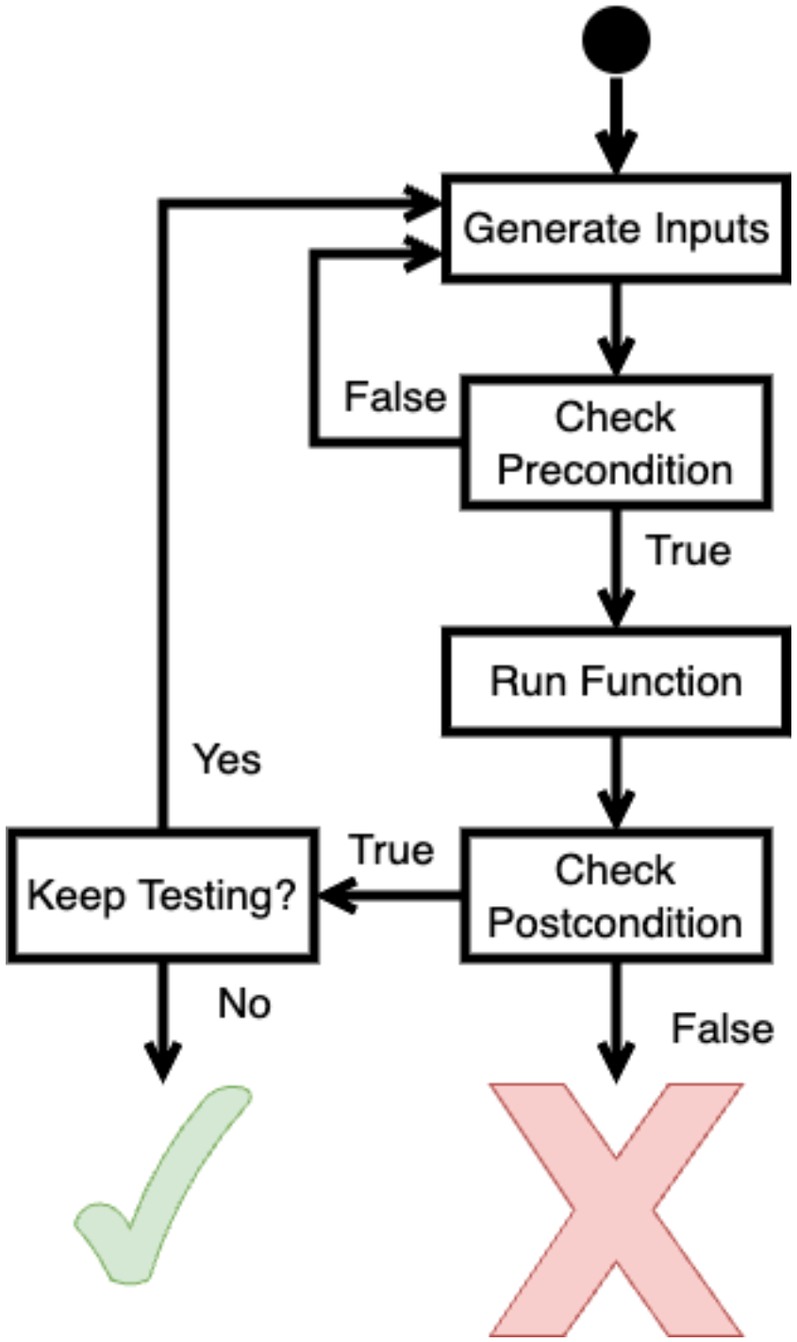
Suresh Jagannathan

Benjamin Delaware


Who wants to release bug-free code?


Who wants to release bug-free code?
Who wants to write tests?

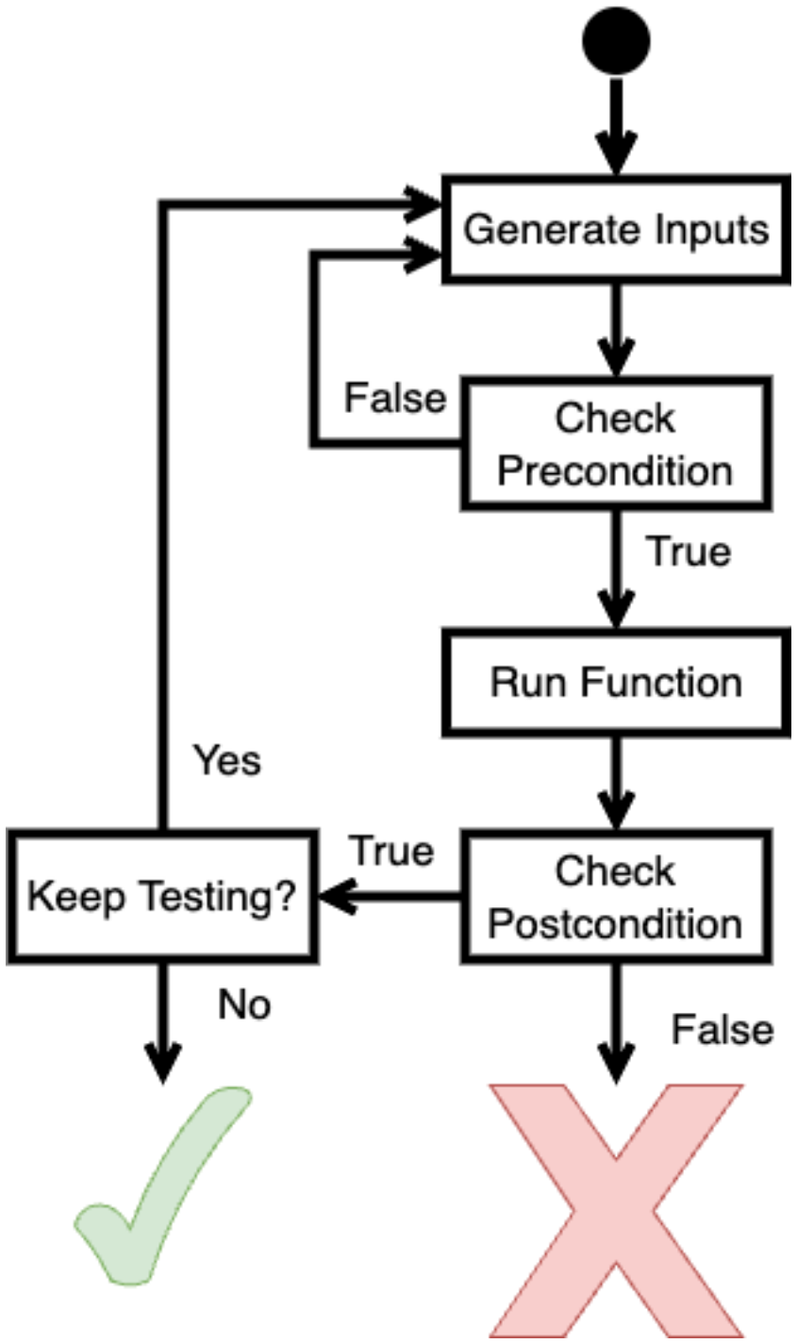
PBT in a Nutshell



Property-Based Testing is an **automated testing strategy**.

A variety of frameworks in all of your favorite languages!

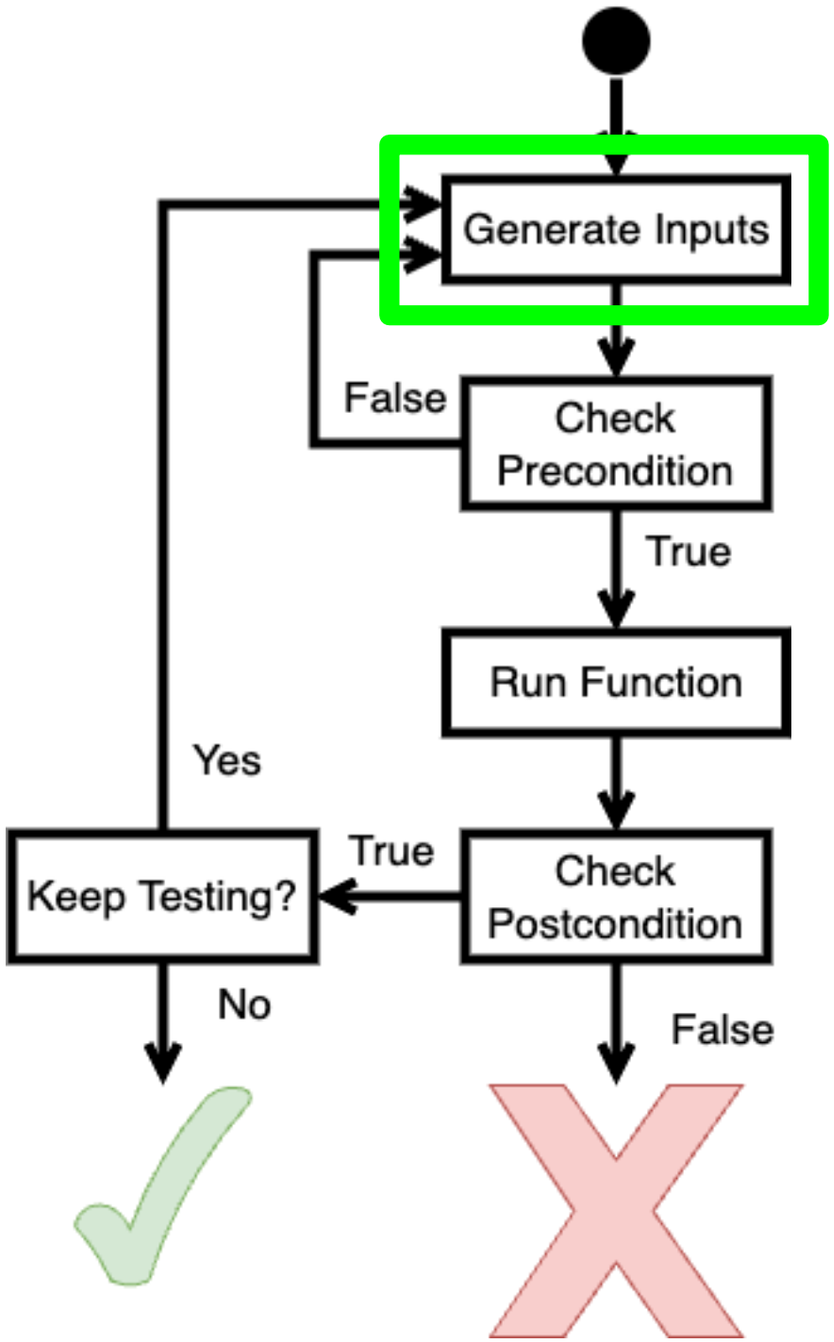
PBT in a Nutshell



Generate a bunch of random well-sized lists

```
let bad_test = Test.make
~count:20000
~name:"tests_R_us"
(default_gen ())
(fun (n, l) ->
  assume(List.length l <= n);
  func l n |> check_post)
```

PBT in a Nutshell

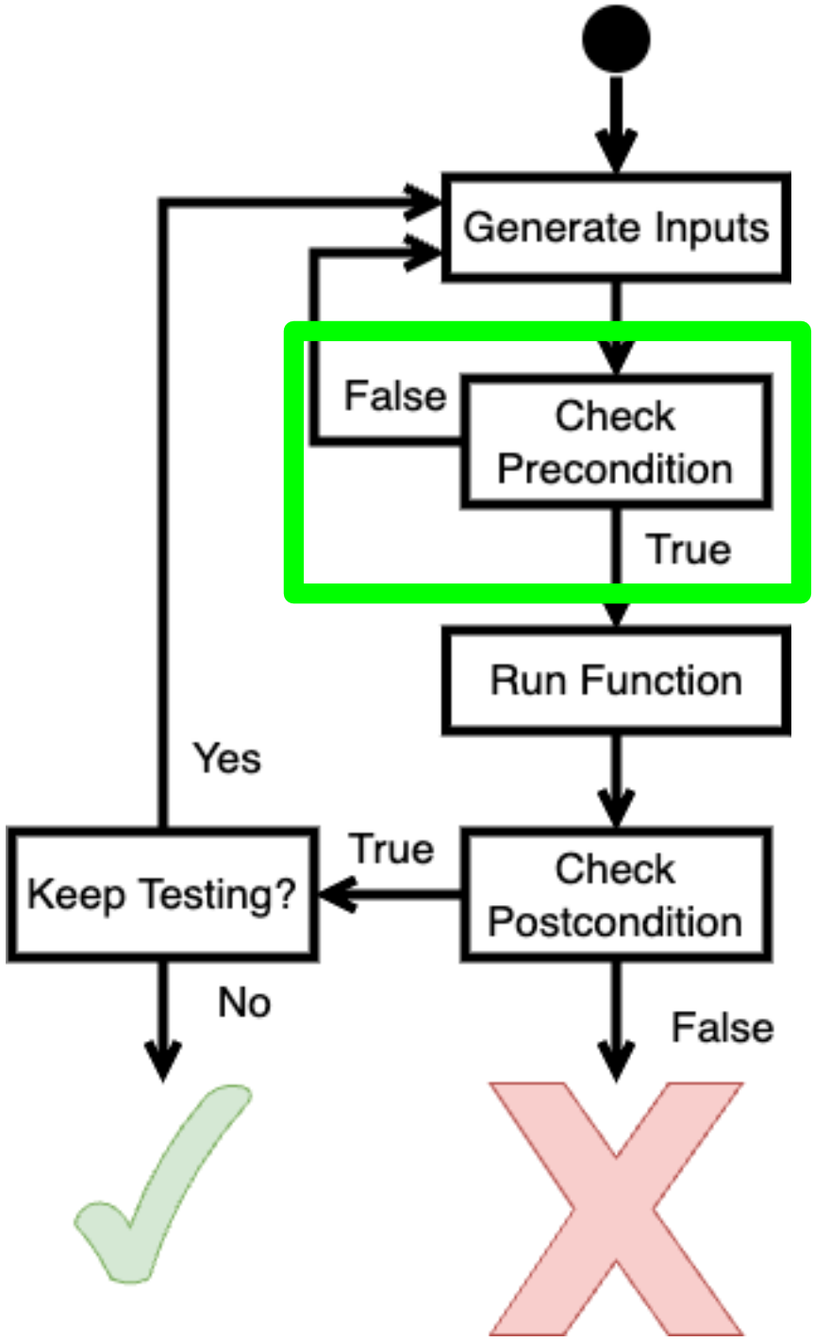


Built-in or are automatically derivable

`[@@ gen]` or `#[derive(Arbitrary)]`

```
let bad_test = Test.make
~count:20000
~name:"tests R us"
(default_gen ())
(fun (n, l) ->
  assume(List.length l <= n);
  func l n |> check_post)
```

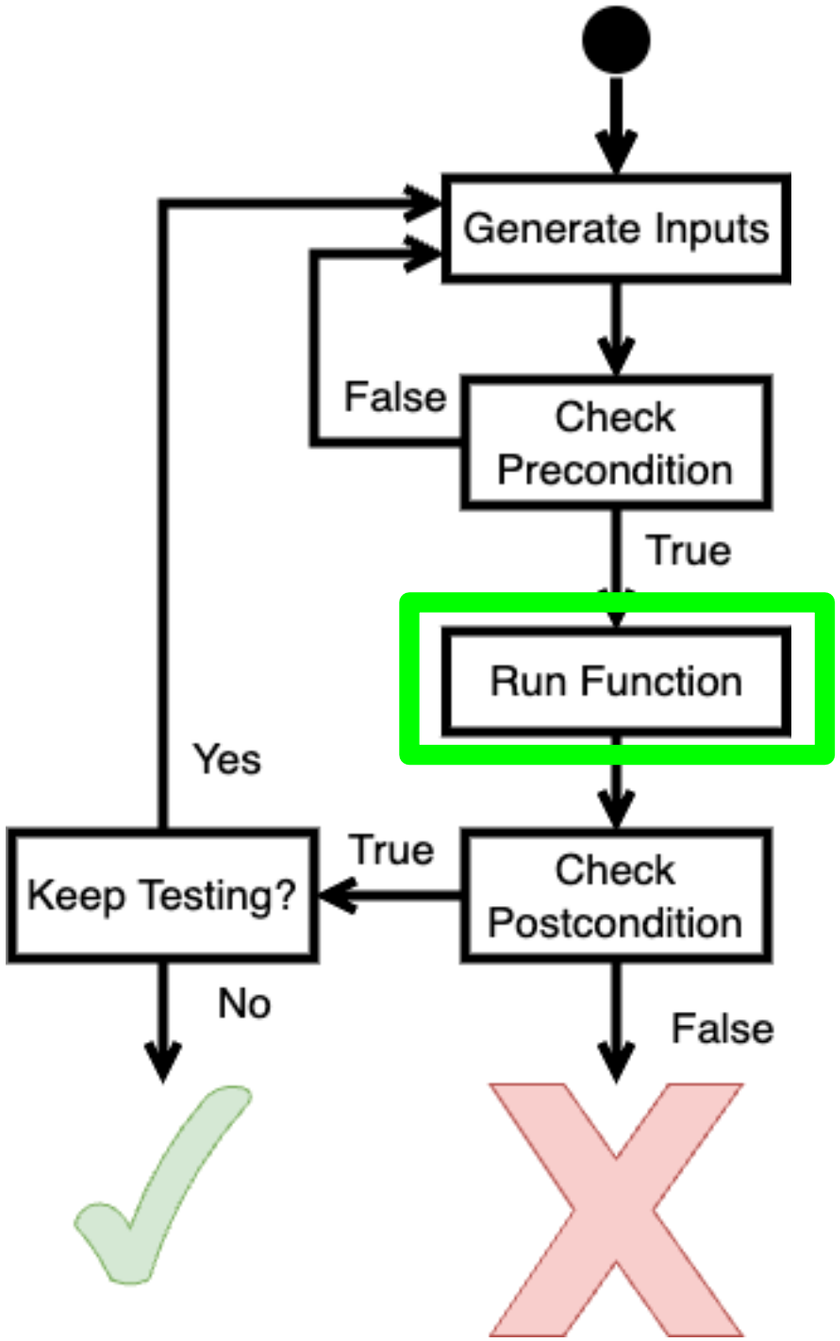
PBT in a Nutshell



Generated inputs must satisfy the precondition

```
let bad_test = Test.make
~count:20000
~name:"tests_R_us"
(default_gen ())
(fun (n, l) ->
  assume(List.length l <= n);
   func l n |> check_post)
```

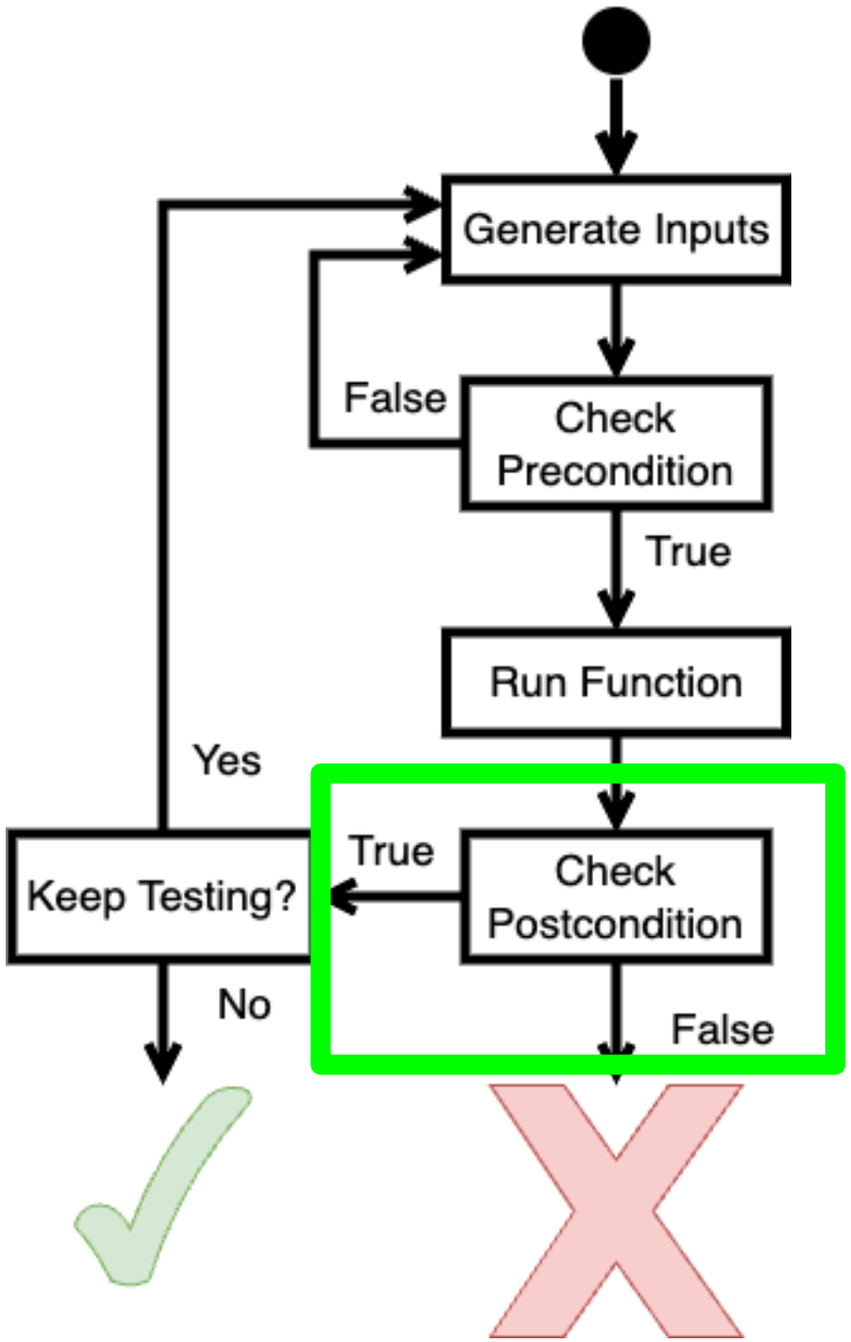
PBT in a Nutshell



User function under test

```
let bad_test = Test.make
~count:20000
~name:"tests_R_us"
(default_gen ())
(fun (n, l) ->
  assume(List.length l <= n);
  func l n |> check_post)
```

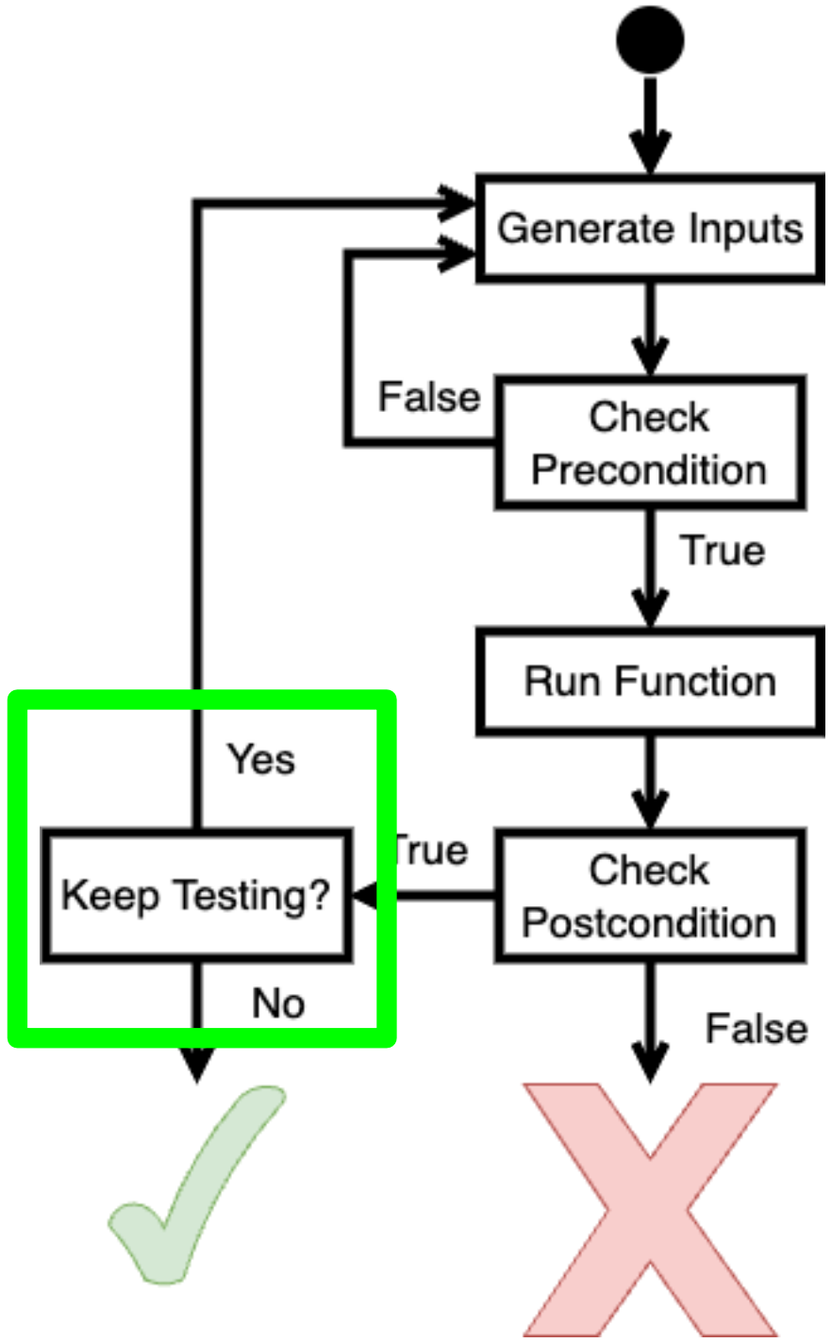
PBT in a Nutshell



User post-condition maps output to True/False

```
let bad_test = Test.make
~count:20000
~name:"tests_R_us"
(default_gen ())
(fun (n, l) ->
  assume(List.length l <= n);
  func l n |> check_post
```

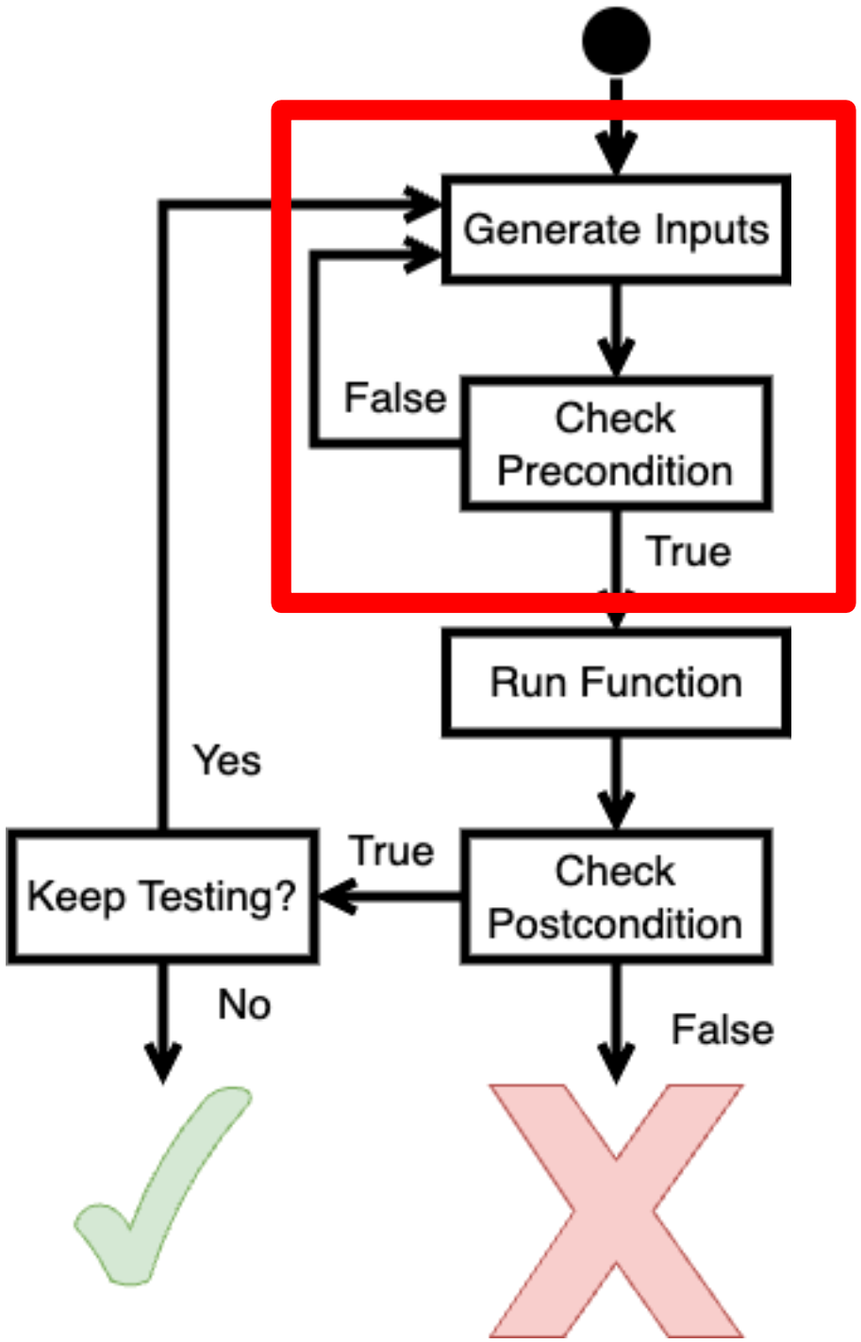
PBT in a Nutshell



Run for 20k attempts or until postcondition fails

```
let bad_test = Test.make
~count:20000
~name:"tests_R_us"
(default_gen ())
(fun (n, l) ->
  assume(List.length l <= n);
  func l n |> check_post)
```

PBT in a Nutshell



Focus of this talk

Wasted work where we aren't testing the user's function

So you think you can test?

How well does the default generator do?

```
let bad_test = QCheck.(Test.make
  ~count:20000
  ~name:"tests R us"
  (default_gen ())
   (fun (n, l) ->
      assume (List.length l <= n);
       func l n |> check_post))
```

So you think you can test?

Sized List: only about 50% of inputs generated inputs are valid

```
❯ dune exec example
generated error  fail  pass / total     time test name
[✓] 20200     0     0 10050 / 20000     0.3s tests_R_us
================================================================================
success (ran 1 tests)
```

```
assume (List.length l <= n);
func l n |> check_post))
```

So you think you can test?

More restrictive precondition -> more rejected inputs

```
let bad_test = QCheck.(Test.make
  ~count:20000
  ~name:"tests_R_us"
   (default_gen ())
   (fun (l) ->
      assume (is_even_list l);
      func l n |> check_post))
```

So you think you can test?

More restrictive precondition -> more rejected inputs

Even List: About 10% of inputs are valid

```
❯ dune exec example
generated error  fail  pass / total     time test name
[✓] 20200     0     0  2138 / 20000     0.3s tests_R_us
================================================================================
success (ran 1 tests)
```

```
assume (is_even_list l);
func l n |> check_post))
```

So you think you can test?

Specialized generators -> all inputs valid

```
let bad_test = QCheck.(Test.make
  ~count:20000
  ~name:"tests_R_us"
  (even_list_gen ())
  (fun (l) ->
     assume (is_even_list l);
     func l n |> check_post))
```

So you think you can test?

Specialized generators -> all inputs valid

```
❯ dune exec example
generated error  fail   pass / total     time test name
[✓] 20000     0     0  20000 / 20000     0.2s tests_R_us
================================================================================
success (ran 1 tests)
```

```
        (fun (l) ->
           assume (is_even_list l);
           func l n |> check_post))
```

So you think you can test?

Users need to write **specialized** generators to ensure enough **valid** inputs

```
let rec even_list_gen (s : int) : int list =
 if s == 0 then [ 2 * int_gen () ]
 else
   if bool_gen () then [ 2 * int_gen () ]
 else
   2 * int_gen () :: even_list_gen (s - 1)
```

So you think you can test?

Users need to write **specialized** generators to ensure enough **valid** inputs

```
let rec even_list_gen (s : int) : int list =
 if s == 0 then [ 2 * int_gen () ]
 else
   if bool_gen () then [ 2 * int_gen () ]
   else
     2 * int_gen () :: even_list_gen (s - 1)
```

Different precondition? Different generator needed


Who wants to release bug-free code?
Who wants to write test-input generators?

OB5: *Developers see writing generators as a distraction, preferring to use derived generators.* Since PBT is often done in the midst of development, developers are reluctant to slow down and write a generator; the task was seen as both difficult and time-consuming. Instead, as seen in §4.4, many participants opted to test with generators derived from the types of their programs.

Goldstein et al., "Property-Based Testing in Practice," ICSE 2024

Program Synthesis to the Rescue?

High Leverage, Low Investment Opportunity for Program Synthesis

Program Synthesis to the Rescue?

Program synthesis requires a **specification** of the users intent

`even_list_gen: "`**`Only`** `non-empty lists of even ints"`

Program Synthesis to the Rescue?

Program synthesis over an initial **template** with components

`even_list_gen: "Only non-empty lists of even ints"`

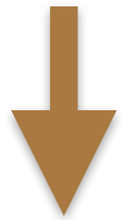
```
let rec even_list_gen (s : int) : int list =
  if s == 0 then Err
  else Err
```

Program Synthesis to the Rescue?

Many **valid** programs make for **disappointing** generators

`even_list_gen: "Only non-empty lists of even ints"`

```
let rec even_list_gen (s : int) : int list =
  if s == 0 then Err
  else Err
```



?

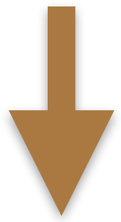
Always producing [0] is valid

```
let rec even_list_gen (s : int) : int list =
 if s == 0 then [ 0 ]
 else [ 0 ]
```

Program Synthesis to the Rescue?

even_list_gen:"**Every** non-empty lists of even ints"

```
let rec even_list_gen (s : int) : int list =
  if s == 0 then Err
  else Err
```



Good generators produce **every** valid input

```
let rec even_list_gen (s : int) : int list =
 if s == 0 then [ 2 * int_gen () ]
 else
   if bool_gen () then [ 2 * int_gen () ]
   else
     2 * int_gen () :: even_list_gen (s - 1)
```

Missing ingredient: “Coverage”

```
even_list_gen:“Every non-empty lists of even ints”

even_list_gen: {s:int | v >= 0}
-> [int list | all_evens(v) && 0<len(v)<=s+1]
```

“Coverage Types”

Formally:

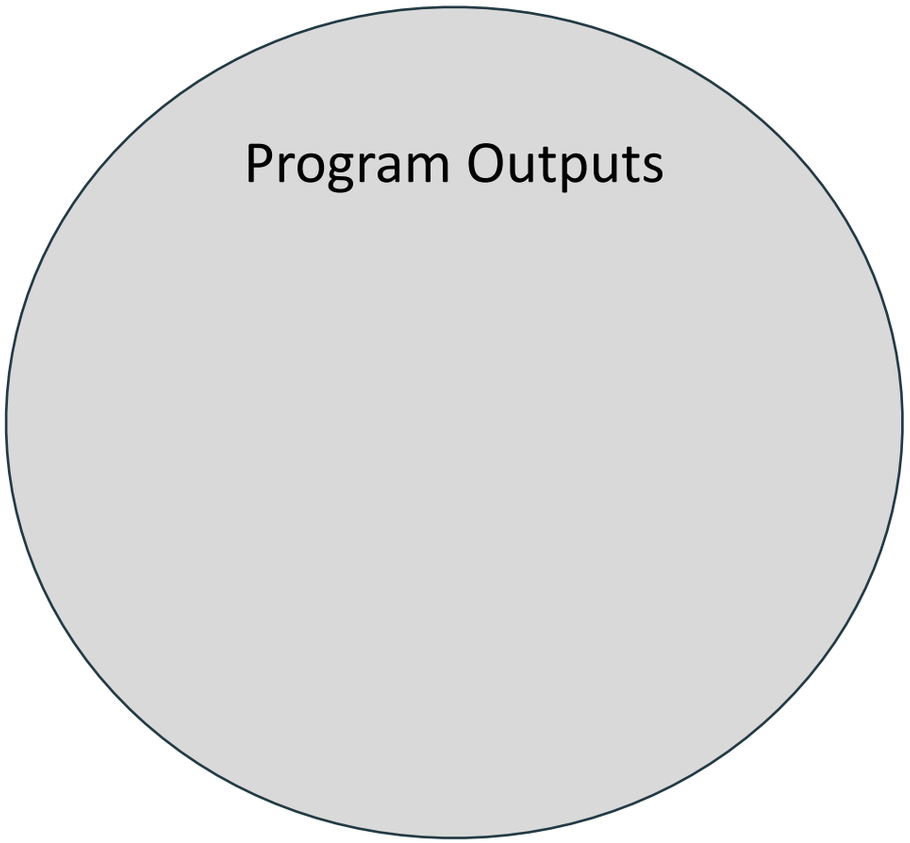
“only” -> safety property

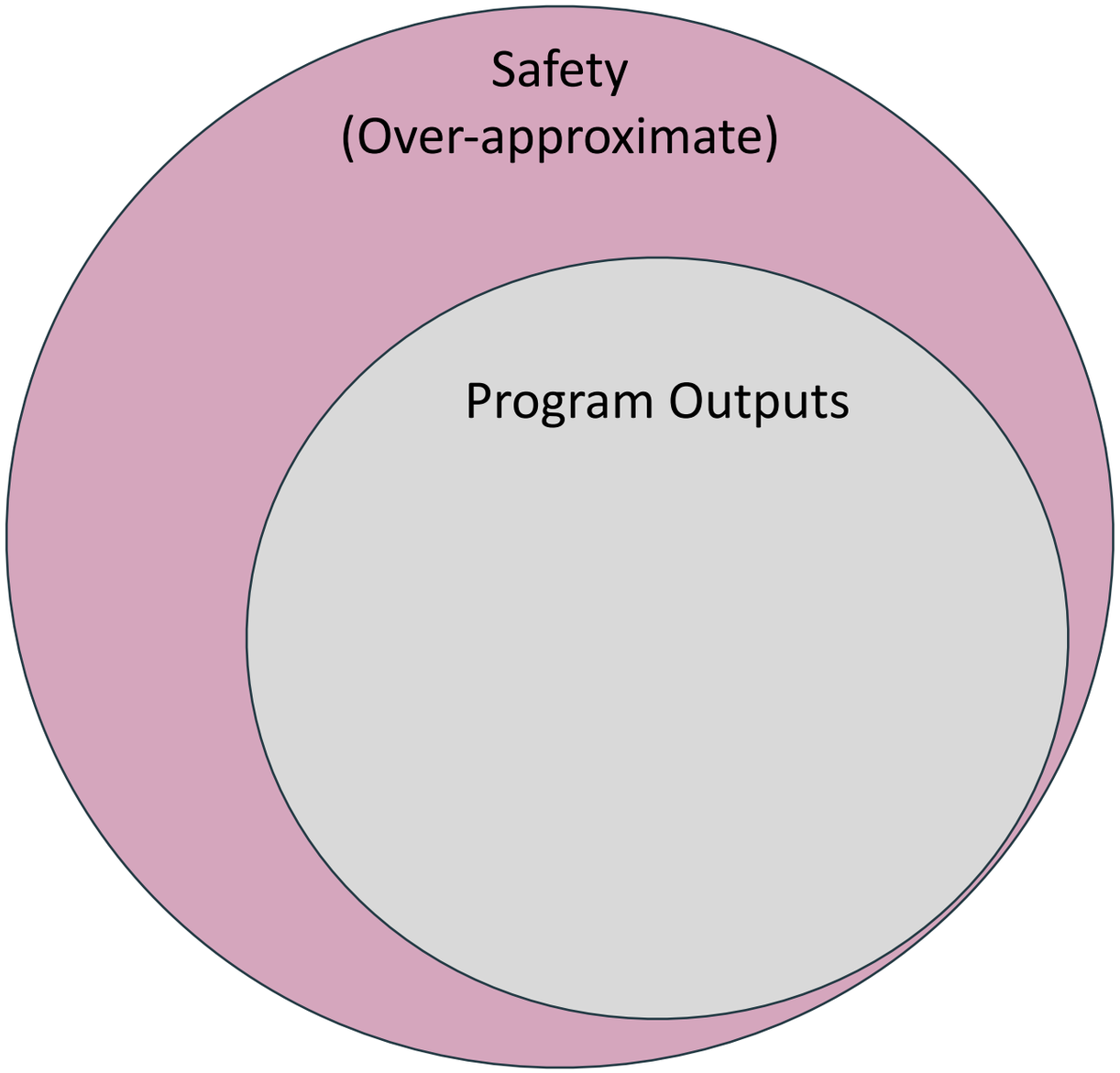
“every” -> coverage property

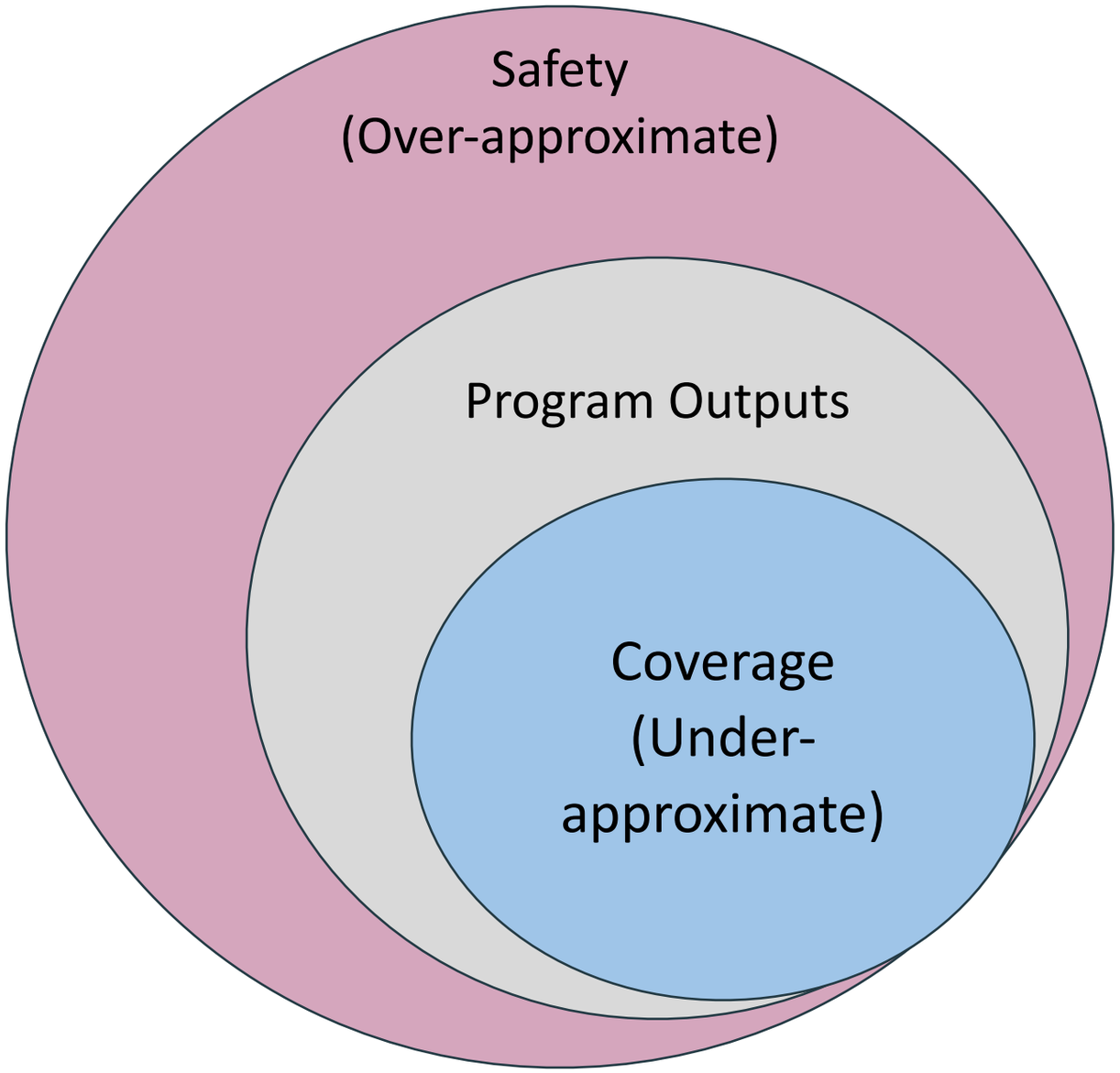
Covering All the Bases: Type-Based Verification of Test Input Generators

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


Program Outputs


Safety
(Over-approximate)
Program Outputs

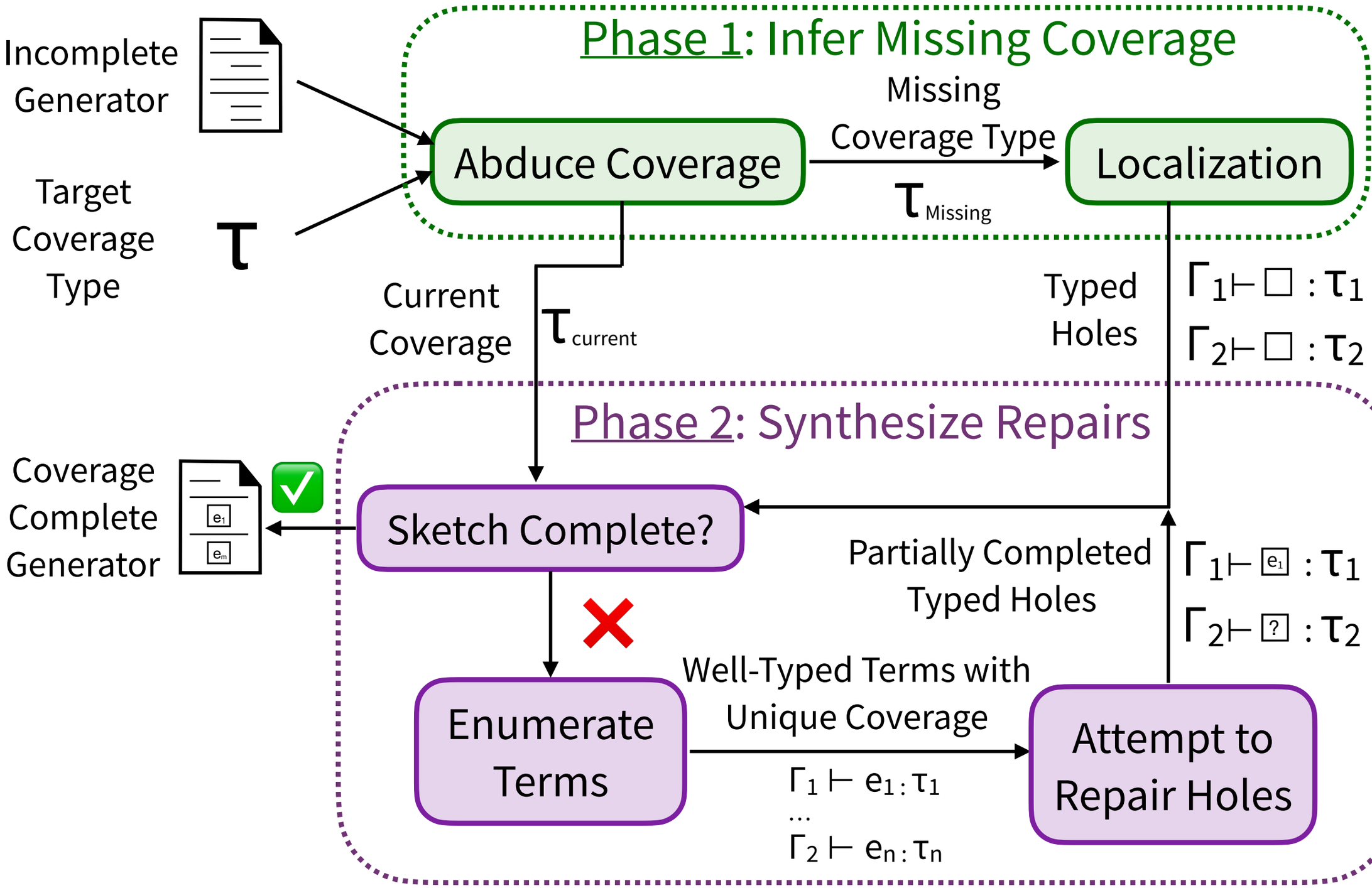

Safety
(Over-approximate)
Program Outputs
Coverage
(Under-
approximate)

Coverage + Synthesis = Cobb

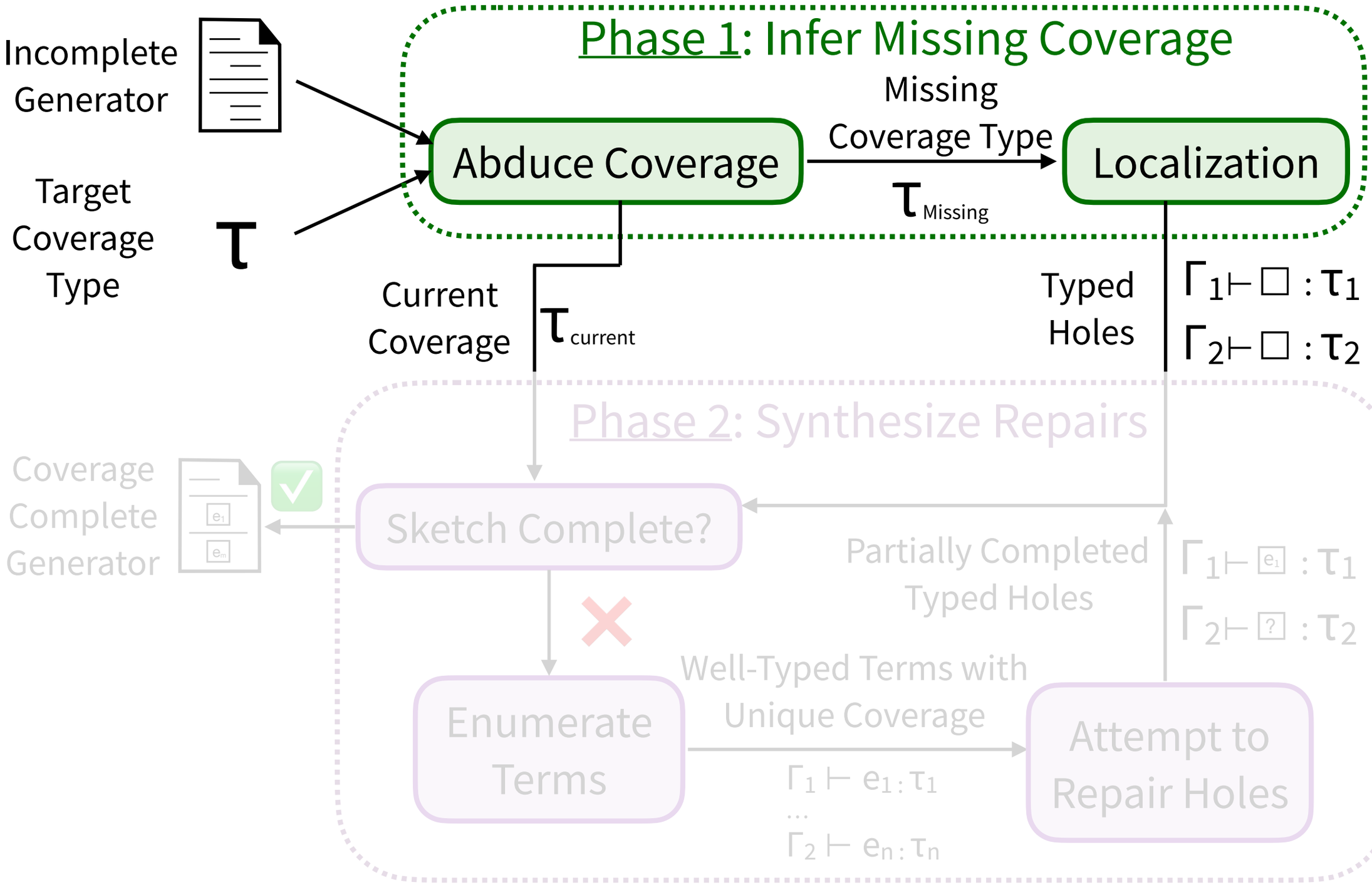
	Safety	Coverage
Type-Based Verification	Liquid Haskell Flux RefinedRust RefinedC refined(scala) ...	Poirot
Type-Based Synthesis	Synquid Cobalt ReSyn SuSLik ...	**Cobb (This work)**

So how does Cobb work?

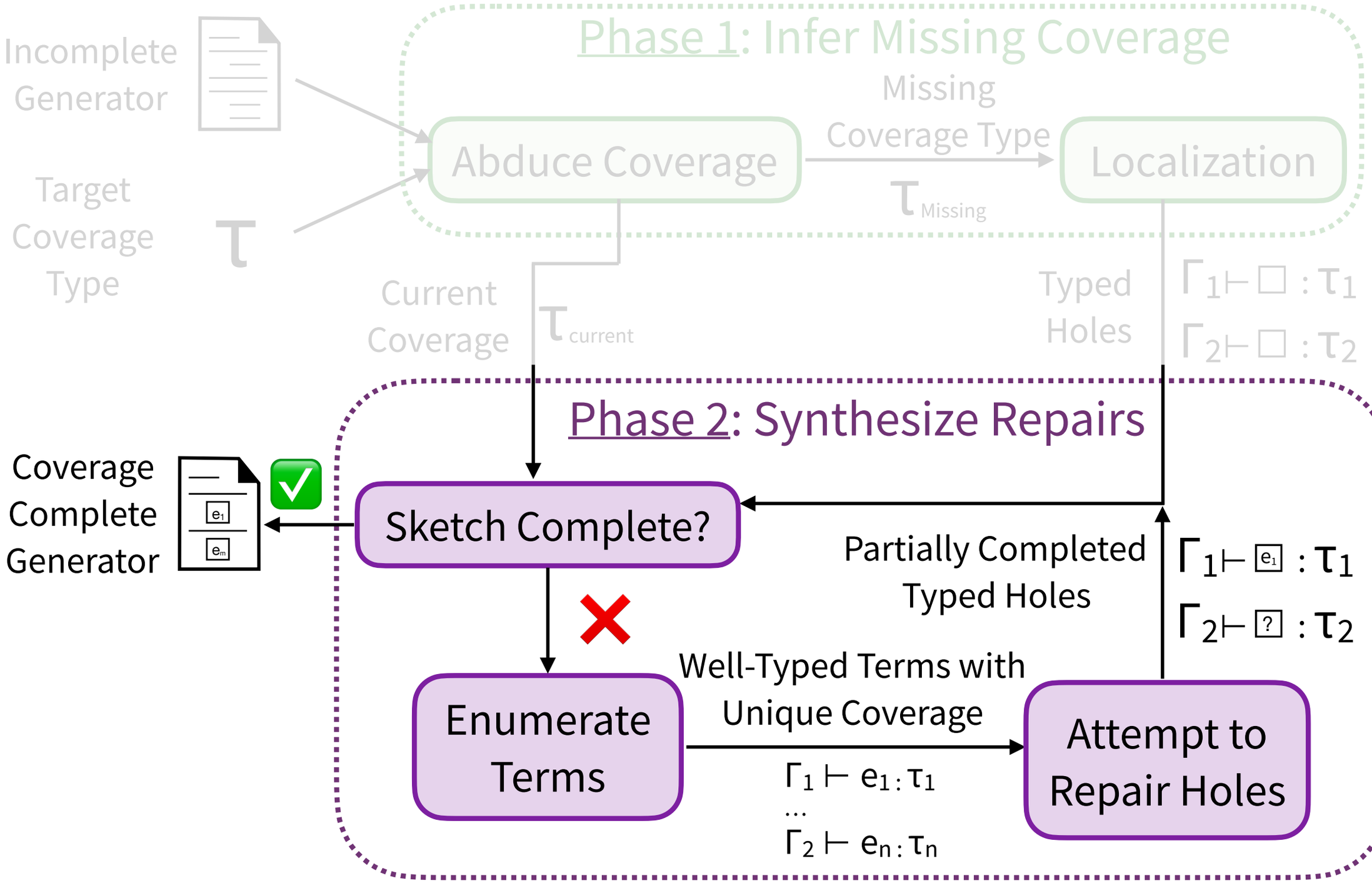
Cobb in a Nutshell



Cobb in a Nutshell



Cobb in a Nutshell



Setting up Synthesis

Users provide a **Coverage Specification** and an **incomplete generator**

```
even_list_gen: {s:int | v >= 0}
-> [int list |
  all_evens(v) && 0<len(v)<=s+1]
```

```
let rec even_list_gen
(s : int) : int list =
  if s == 0 then Err
  else Err
```

Setting up Synthesis

Phase One: Augments the sketch with **typed holes**

```
even_list_gen: {s:int | v >= 0}
-> [int list |
  all_evens(v) && 0<len(v)<=s+1]
```

```
let rec even_list_gen
(s : int) : int list =
  if s == 0 then Err
  else Err
```

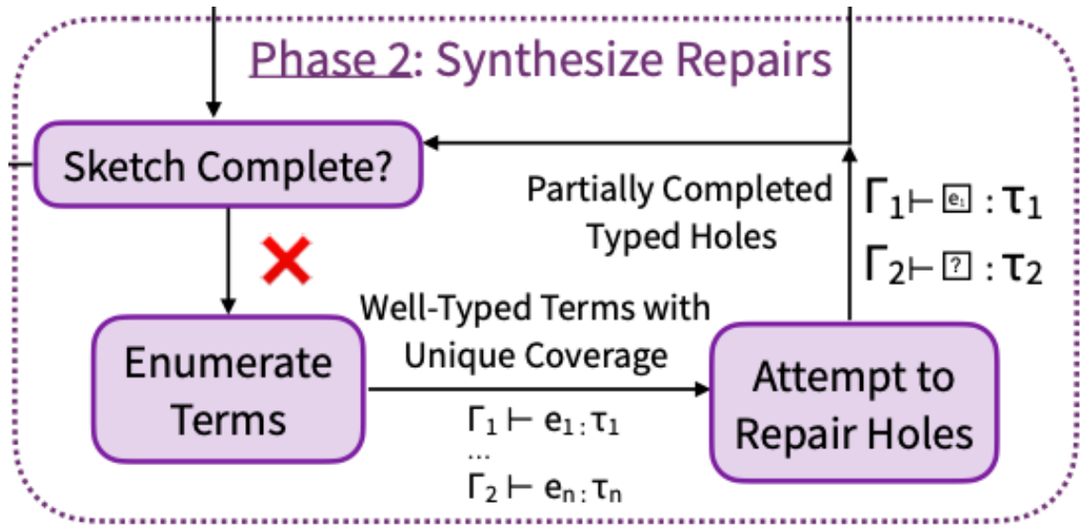
Phase 1

```
let rec even_list_gen (s : int) : int list =
  if s == 0 then Γ1 ⊢ □ : τmissing_1
 else Γ2 ⊢ □ : τmissing_2
```

Setting up Synthesis

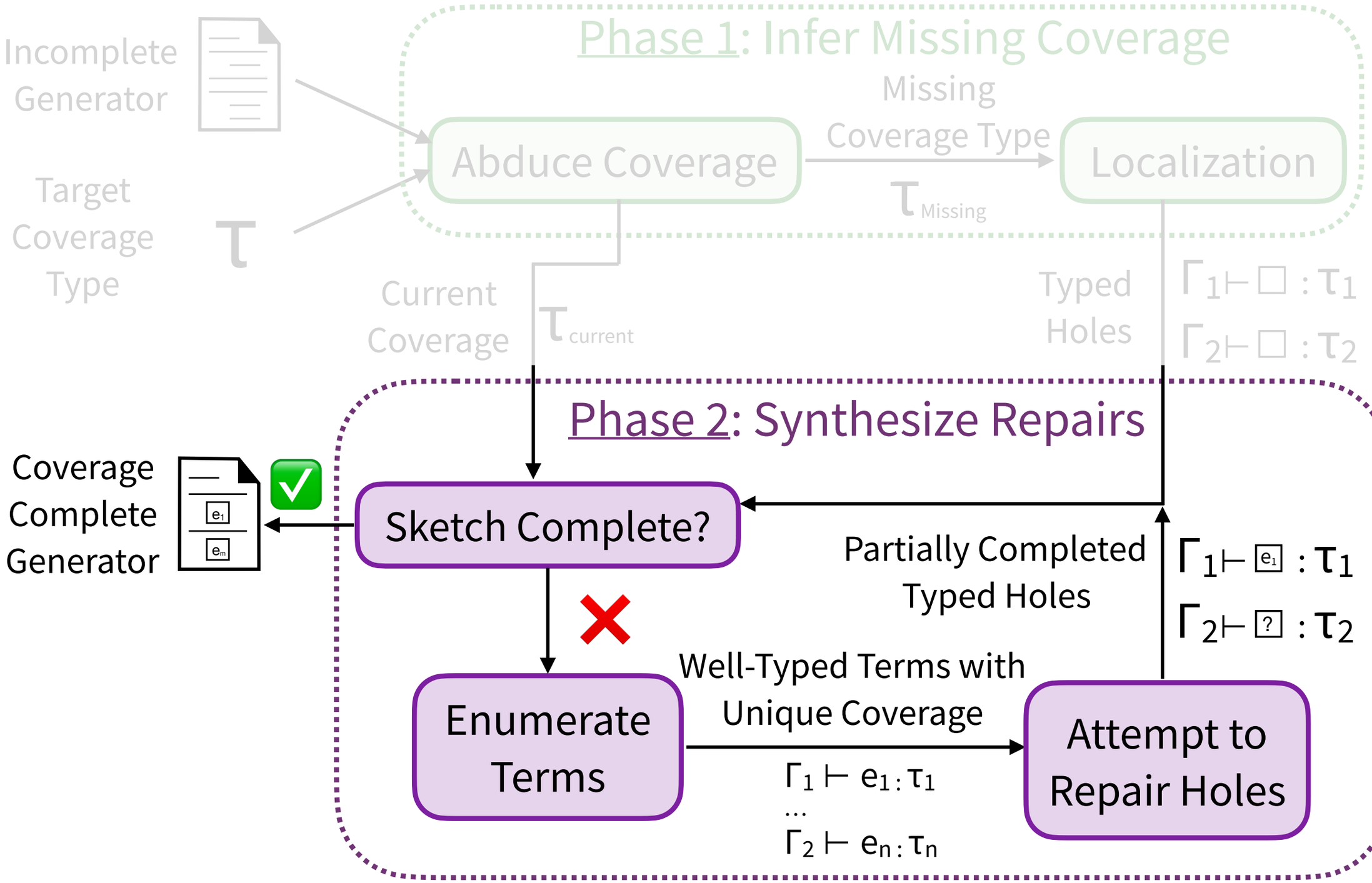
Phase Two: **Component-based Synthesis** to fill holes

```
even_list_gen: {s:int | v >= 0}
-> [int list |
   all_evens(v) && 0<len(v)<=s+1]
```

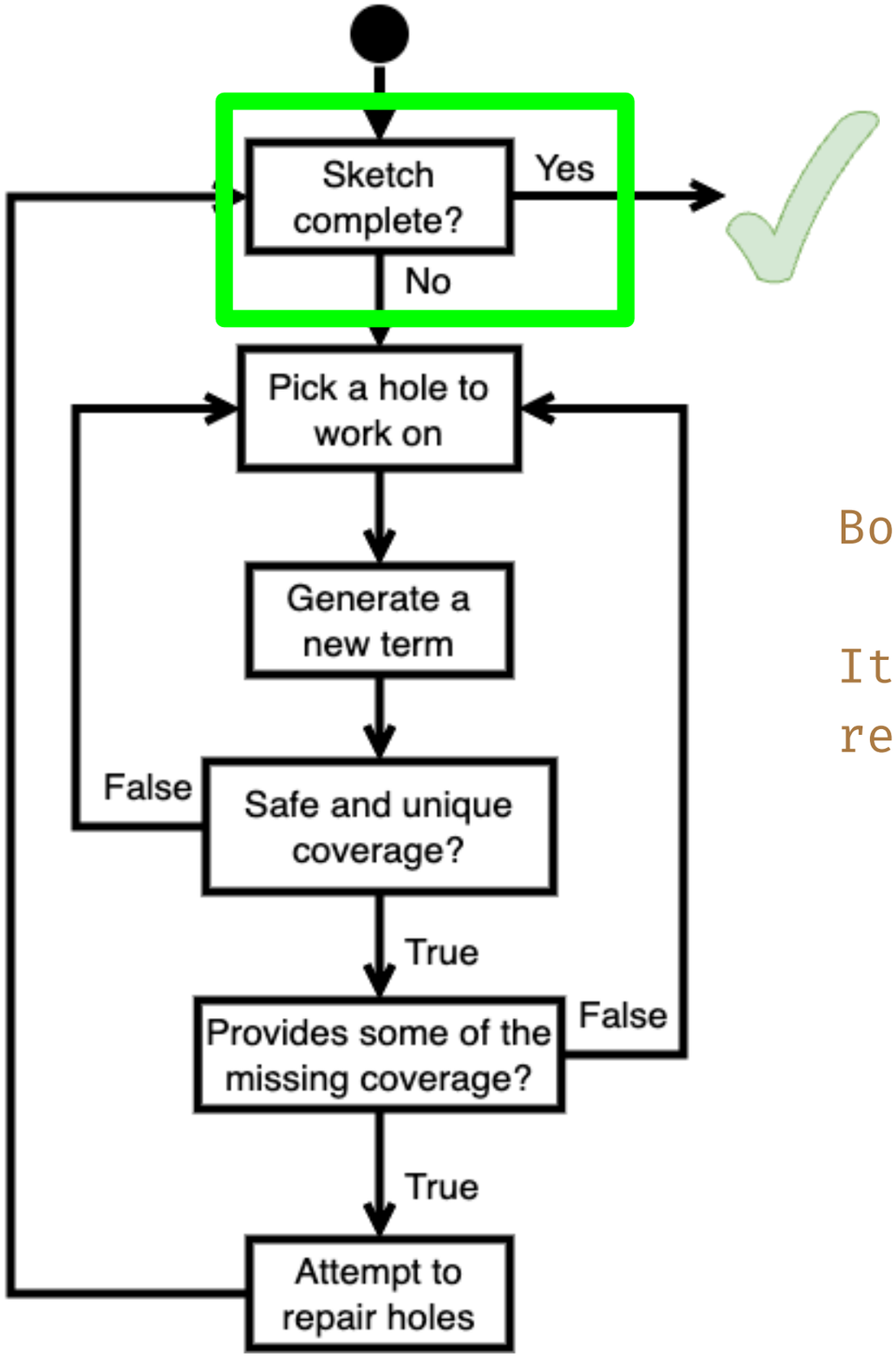



```
let rec even_list_gen (s : int) : int list =
  if s == 0 then Γ1 ⊢ □ : τmissing_1
 else Γ2 ⊢ □ : τmissing_2
```

Cobb in a Nutshell



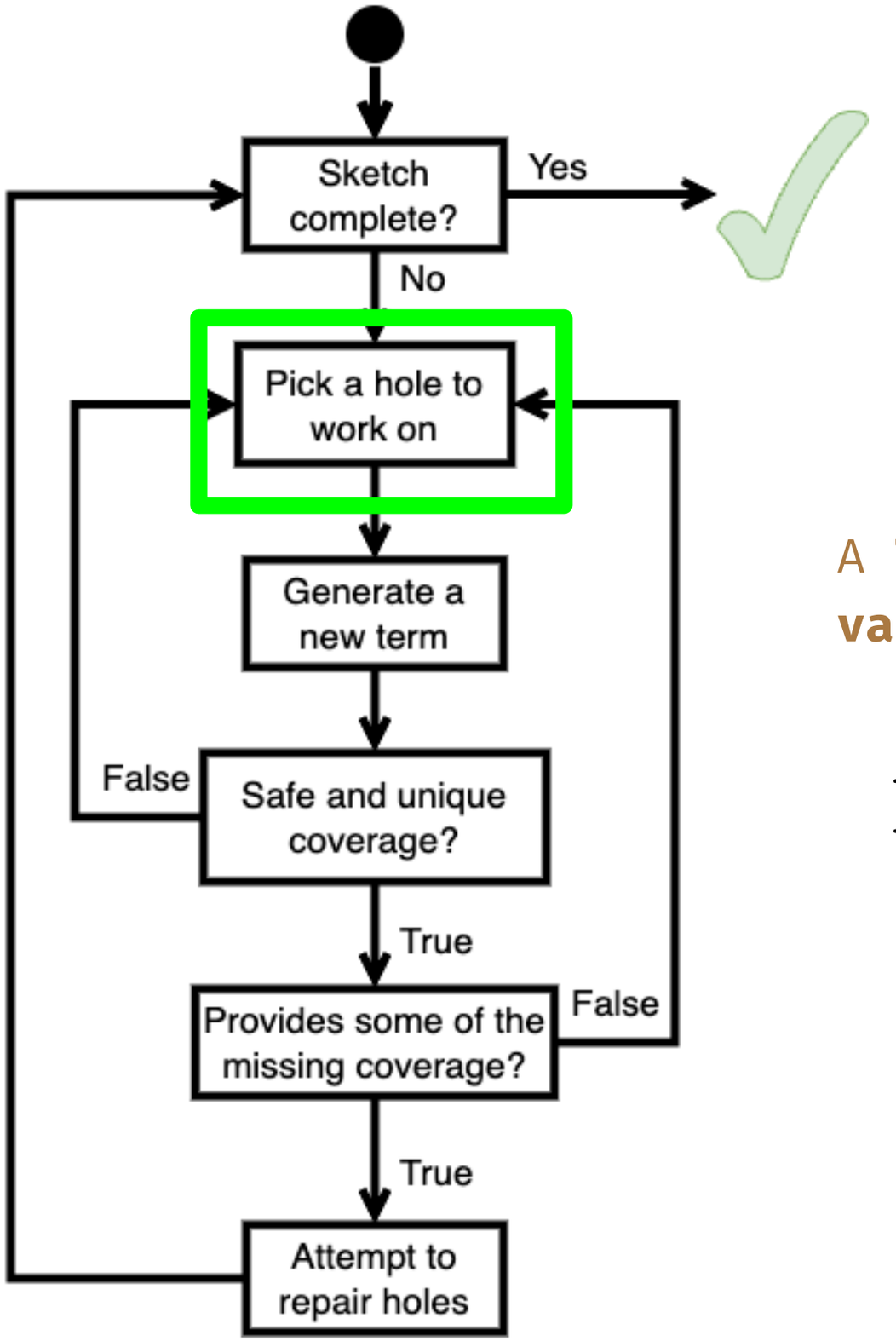
Enumerate Terms



Bottom-up, Cost-based Enumeration

Iterate until sketch is fully repaired or max-bound is hit

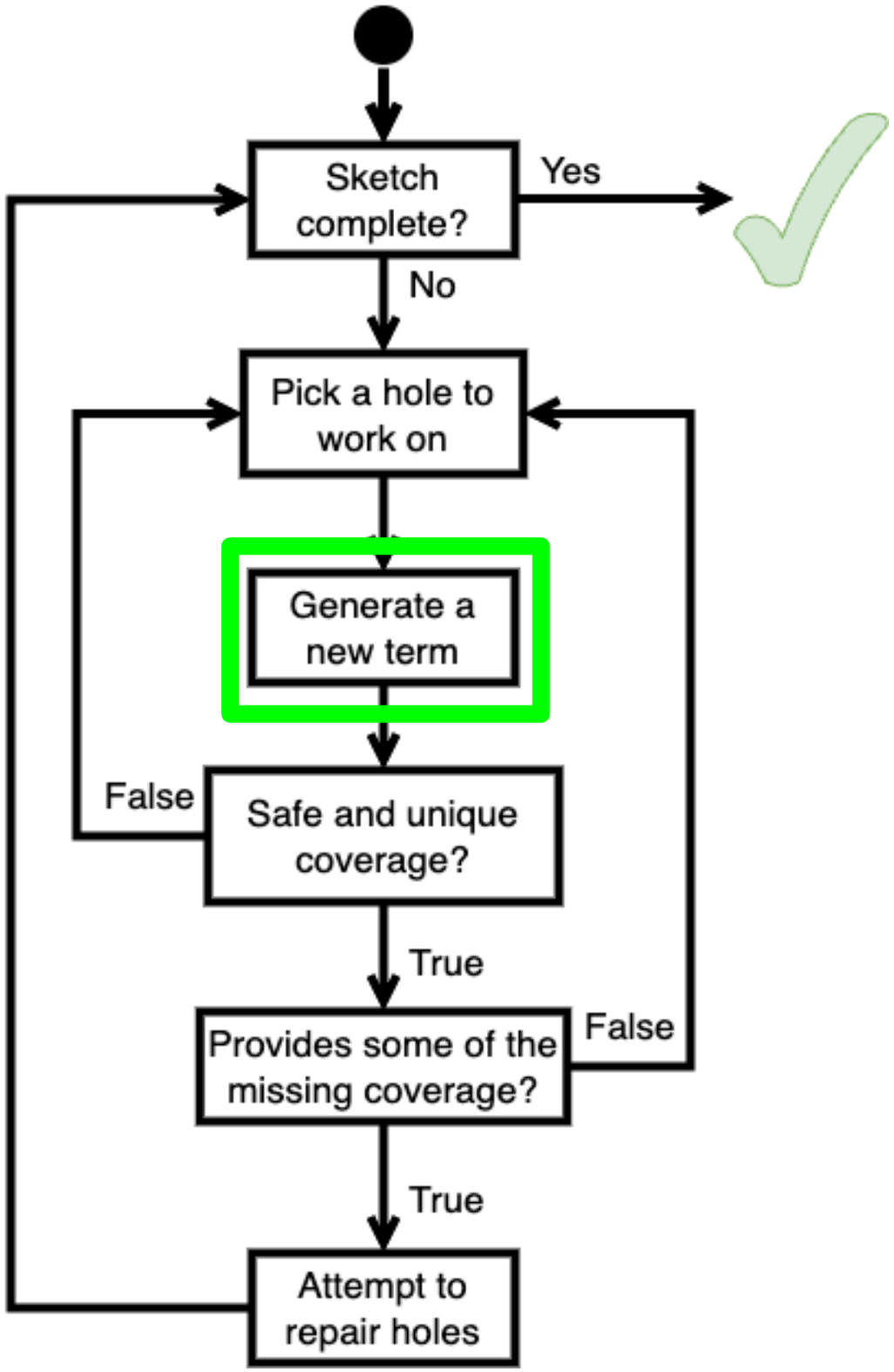
Enumerate Terms



A **Typing Context** and set of **local variables** per path

$$\Gamma_1 \vdash \square : \tau_{missing\_1}$$

Enumerate Terms



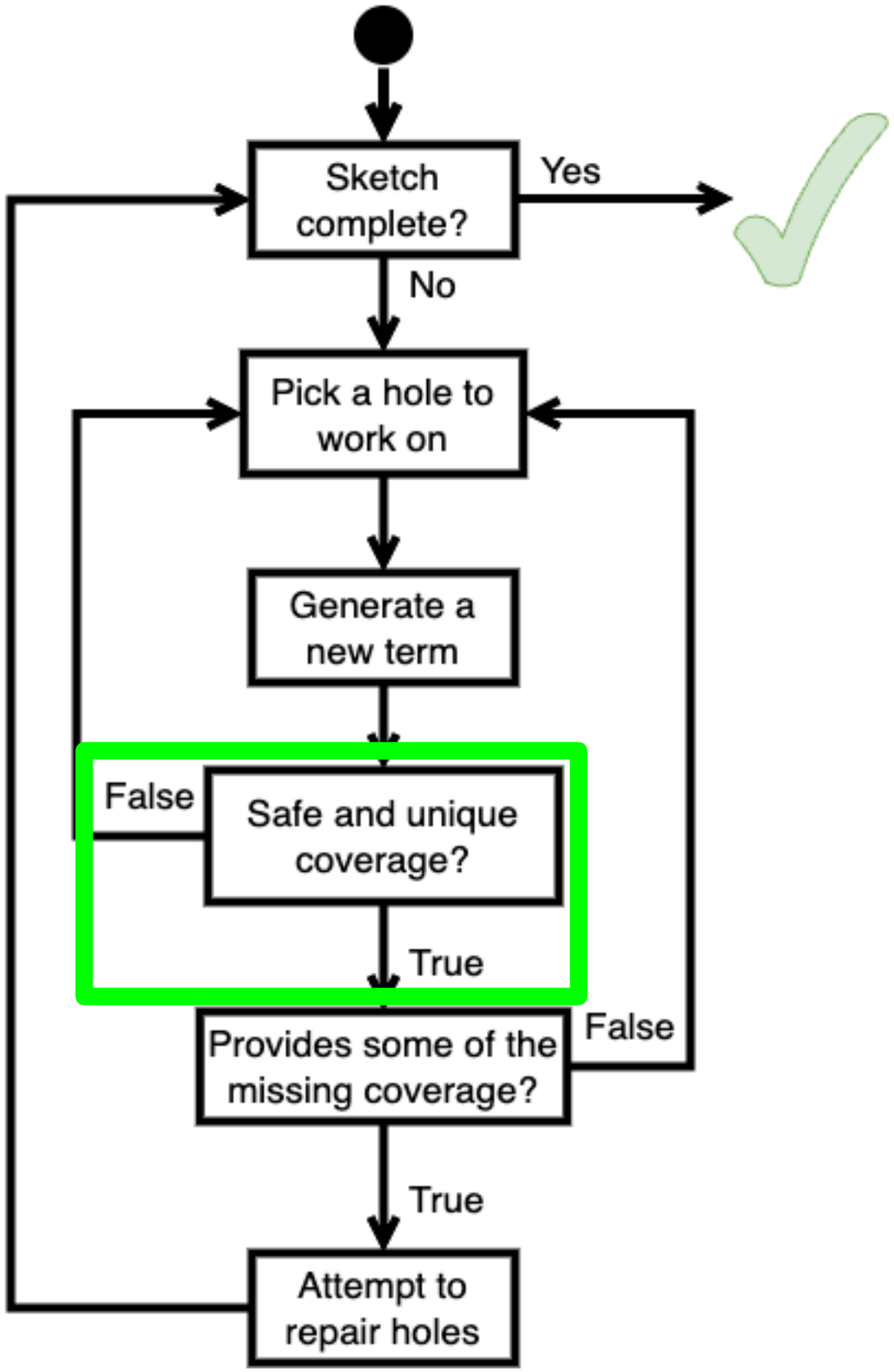
Term cost:α

s-1

2*int_gen()

2*s

Small terms that are initially enumerated



Enumerate Terms

Term cost:α

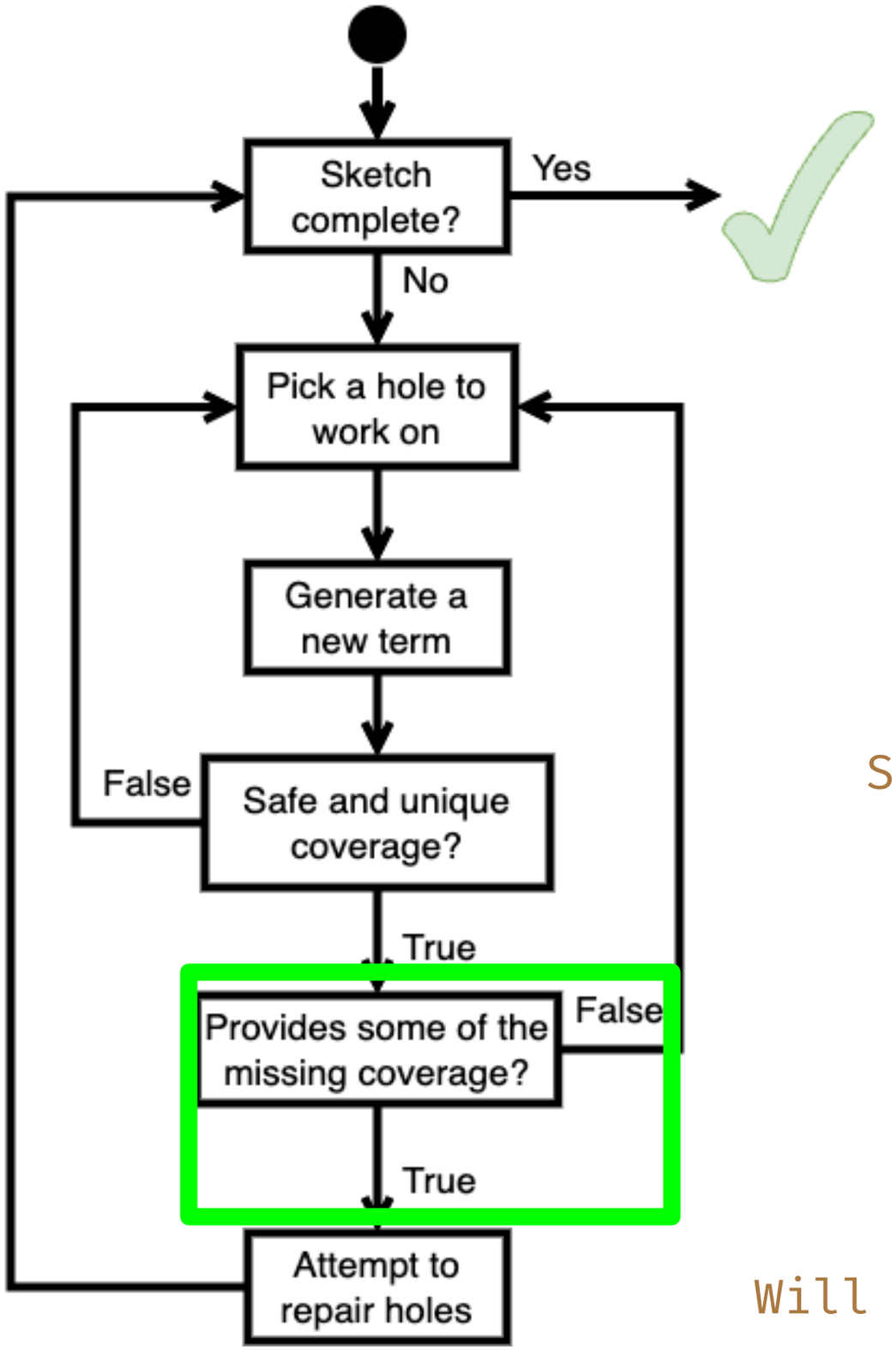
$\Gamma_1 \vdash$ s-1 : τ_1 ✓

$\Gamma_1 \vdash$ 2*int_gen(): τ_2 ✓

$\Gamma_1 \vdash$ 2*s : τ_3 ✓

Infer a coverage type for each term

Typecheck: safe
Subtyping check: unique



Enumerate Terms

Term cost:α

$\Gamma_1 \vdash$ s-1 : τ_1

$\Gamma_1 \vdash$ 2*int_gen(): τ_2

$\Gamma_1 \vdash$ 2*s : τ_3

Subtype of missing coverage?

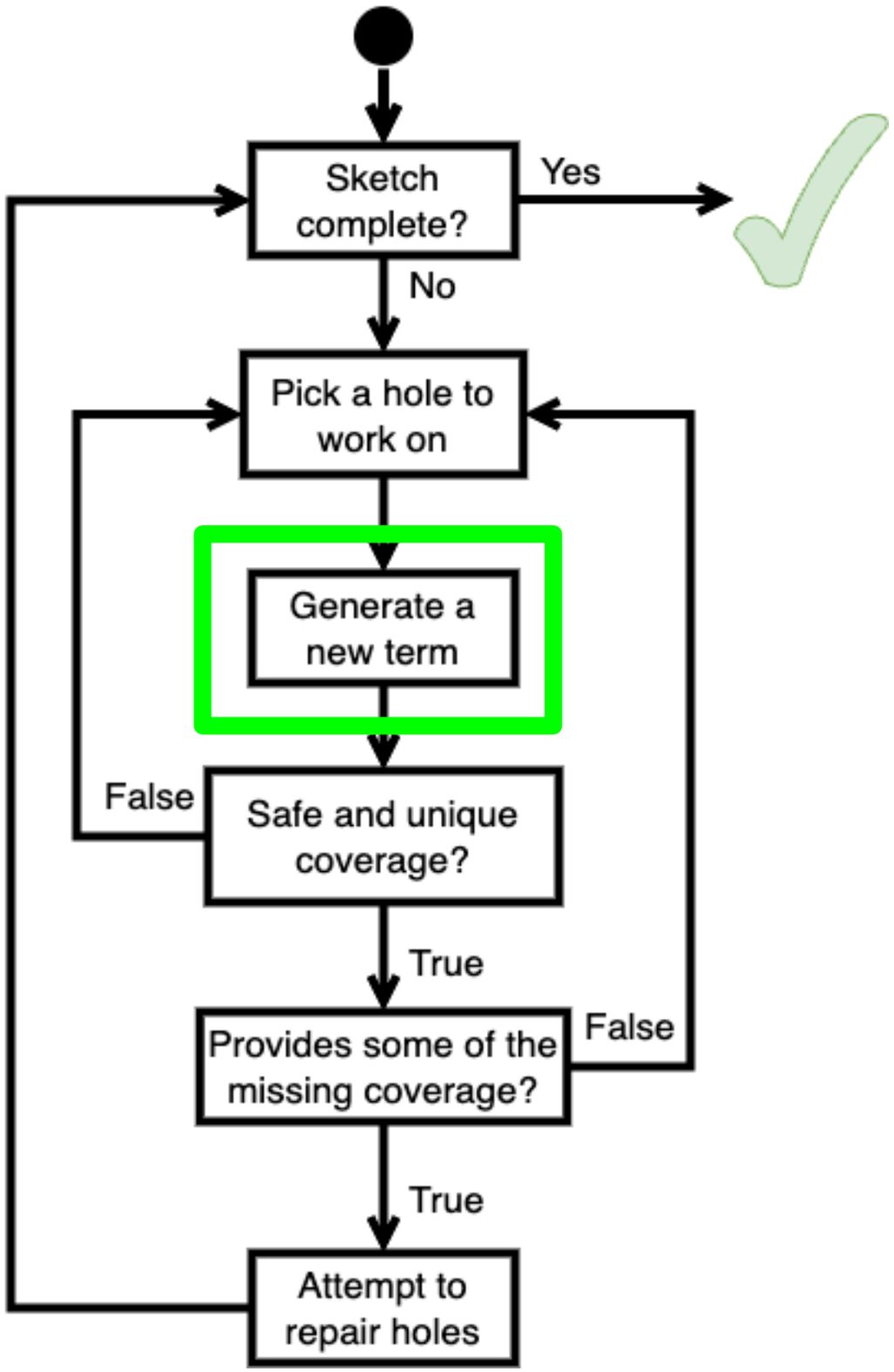
$\tau_{missing\_1}$ <: τ_1 ✗

$\tau_{missing\_1}$ <: τ_2 ✗

$\tau_{missing\_1}$ <: τ_3 ✗

Will be used to enumerate larger terms

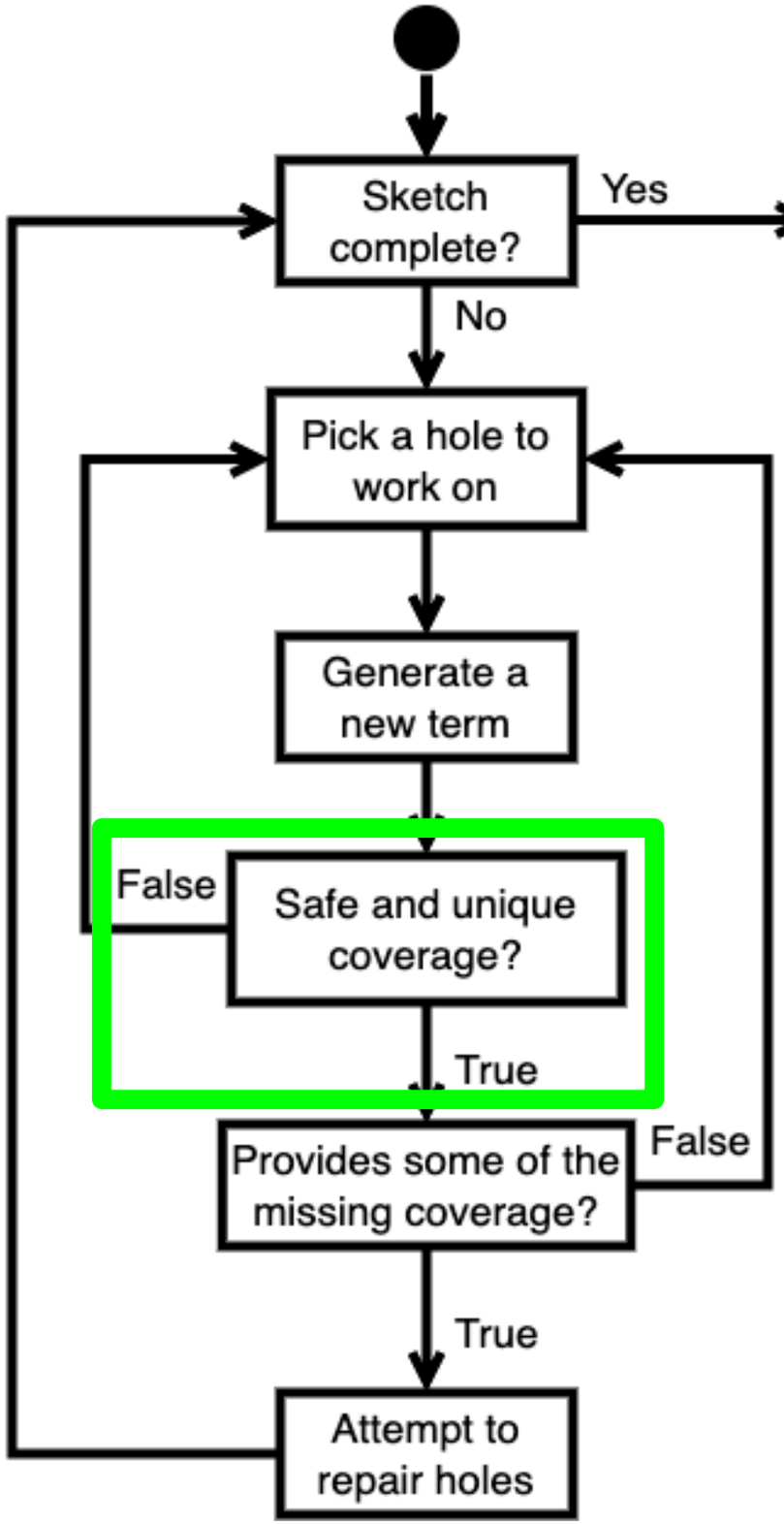
Enumerate Terms



Term cost:α′

```
even_list_gen(0)
even_list_gen(int_gen())
even_list_gen(s - 1)
```

Recursive terms are prioritized



Enumerate Terms

Term cost:α′

~~$\Gamma_1 \vdash$ even_list_gen(**0**) : τ_4~~ ✗

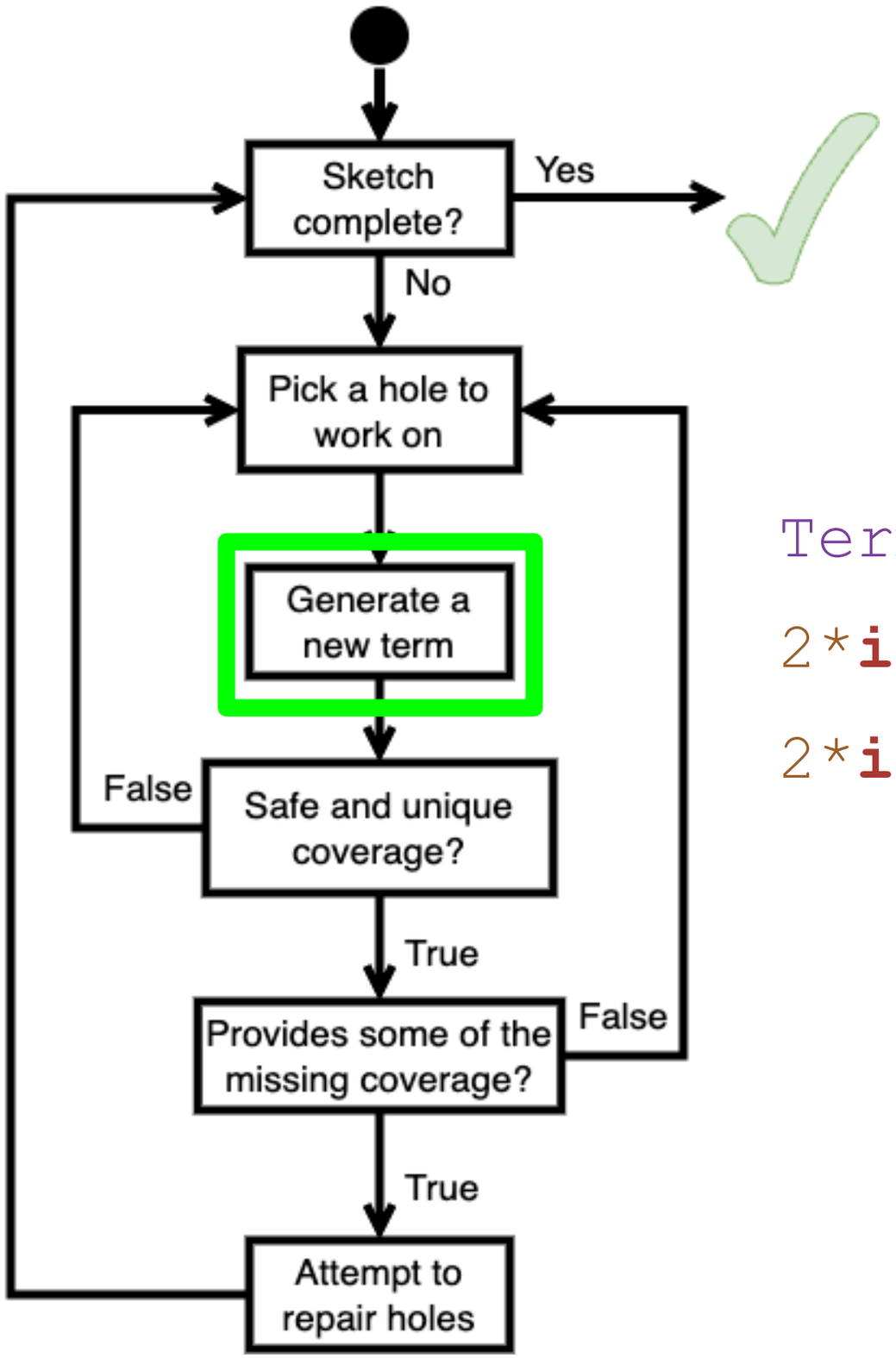
~~$\Gamma_1 \nvdash$ even_list_gen(**int_gen**()) : τ_5~~ ✗

$\Gamma_1 \vdash$ even_list_gen(s - 1) : τ_6 ✓

Some terms are rejected for being **unsafe** or **“coverage equivalent”**

- **int_gen**() not less than s
- even_list_gen(**0**) equivalent to the base case

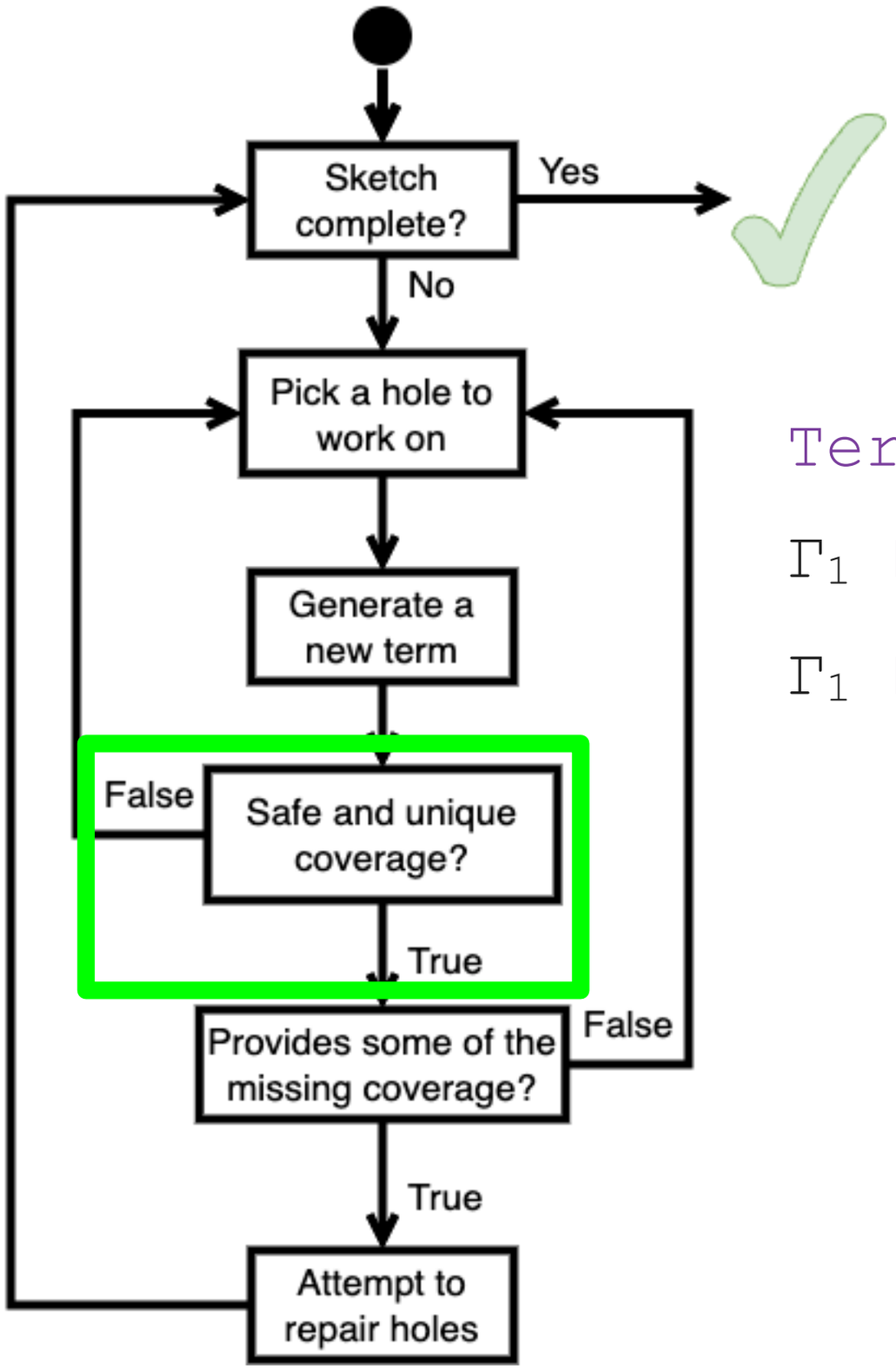
Enumerate Terms




```
Term cost:α''

2*int_gen()::[]

2*int_gen():: even_list_gen(s-1)
```



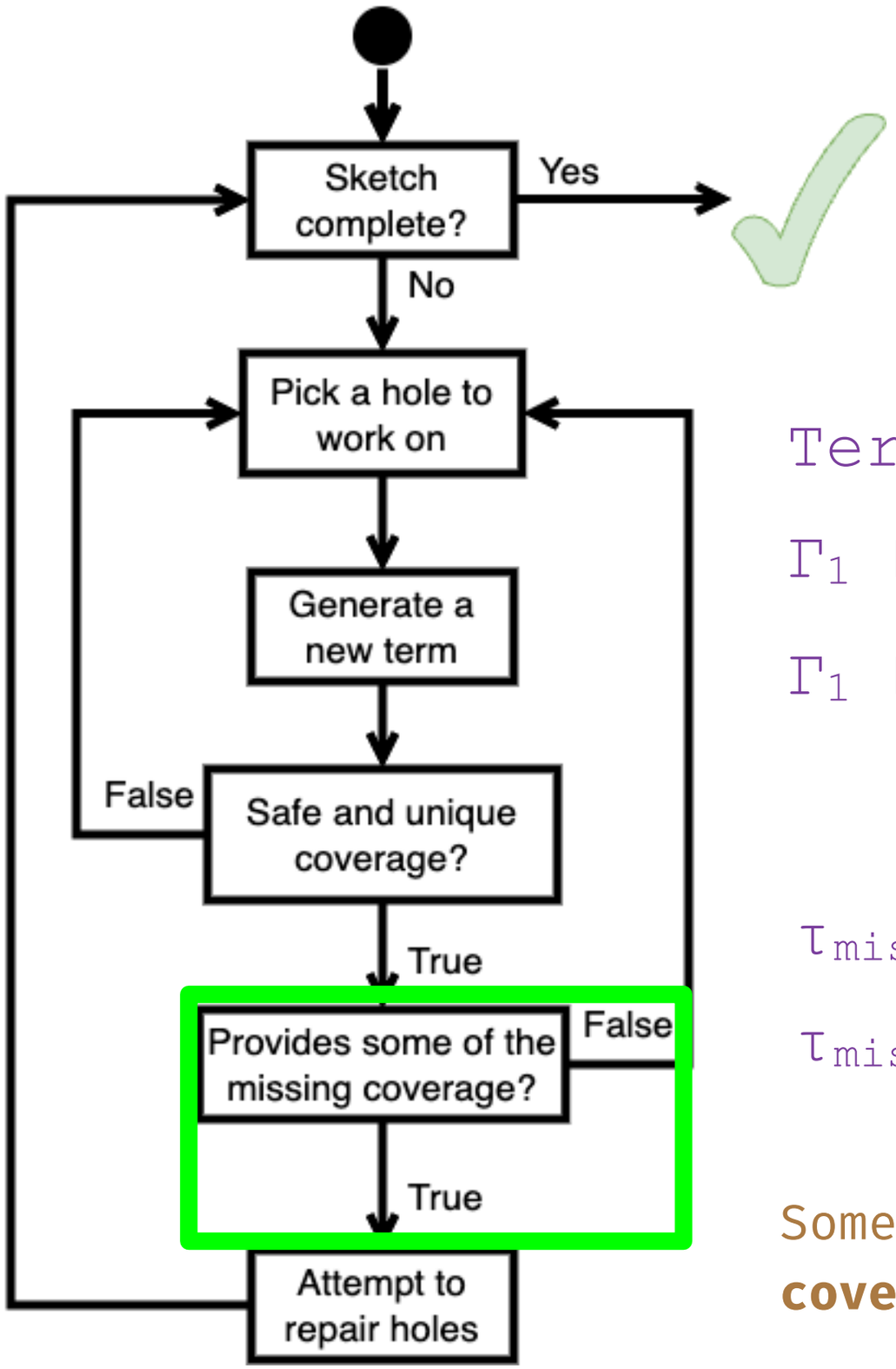
Enumerate Terms

Term cost:α''

$\Gamma_1 \vdash$ `2*int_gen()::[]` : τ_7 ✓

$\Gamma_1 \vdash$ `2*int_gen()` :: ✓

`even_list_gen(s-1)` : τ_8



Enumerate Terms

Term cost:α''

$\Gamma_1 \vdash$ `2*int_gen()::[]` : τ_7 ✓

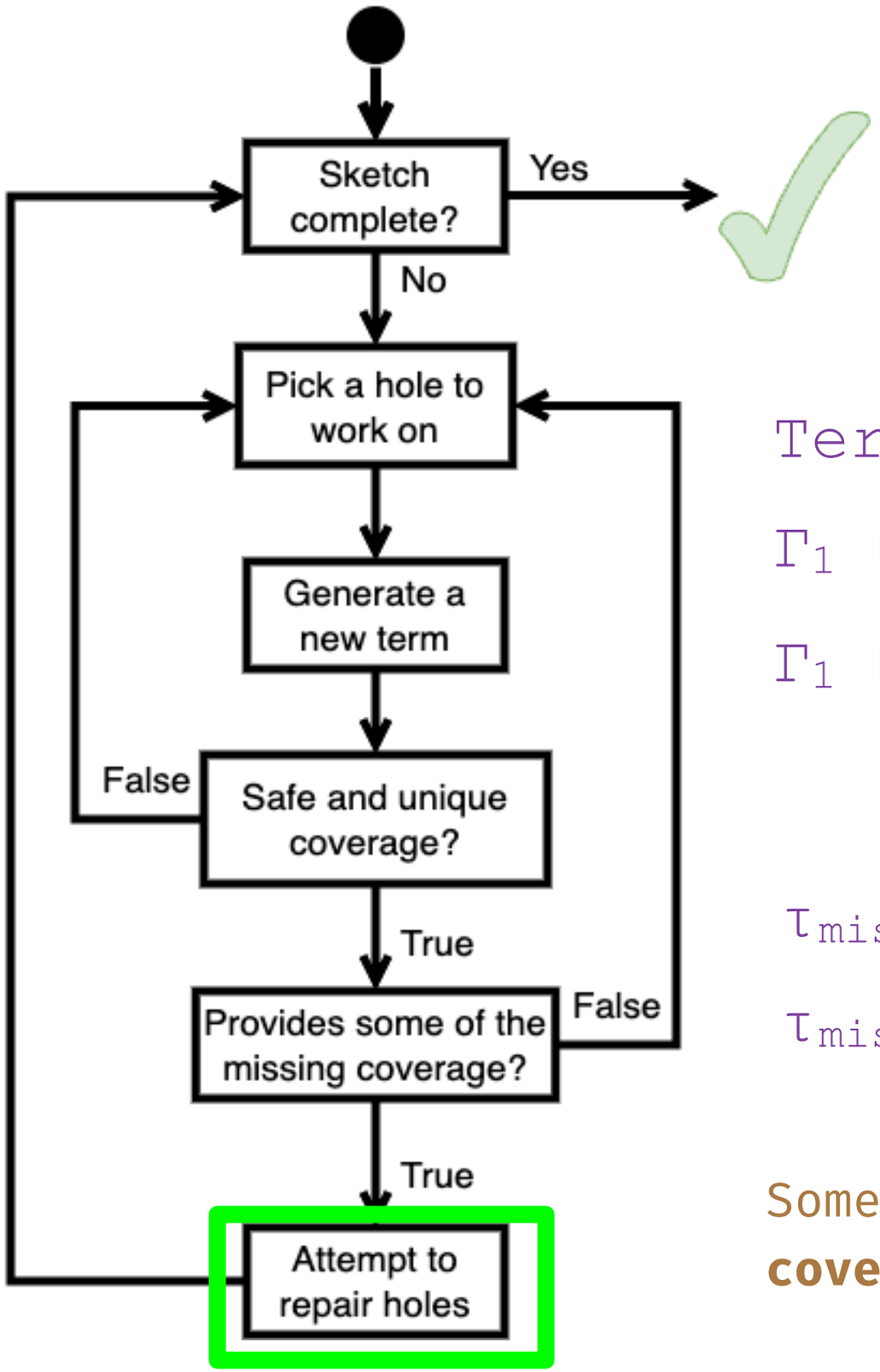
$\Gamma_1 \vdash$ `2*int_gen()` :: ✓

`even_list_gen(s-1)` : τ_8

$\tau_{missing\_1}$ <: τ_7 ✓

$\tau_{missing\_1}$ <: τ_8 ✓

Some terms are a **sub-type** of our **missing coverage**, thus could repair the hole



Enumerate Terms

Term cost:α''

$\Gamma_1 \vdash$ `2*int_gen()::[]` : τ_7 ✓

$\Gamma_1 \vdash$ `2*int_gen()` :: ✓

`even_list_gen(s-1)` : τ_8

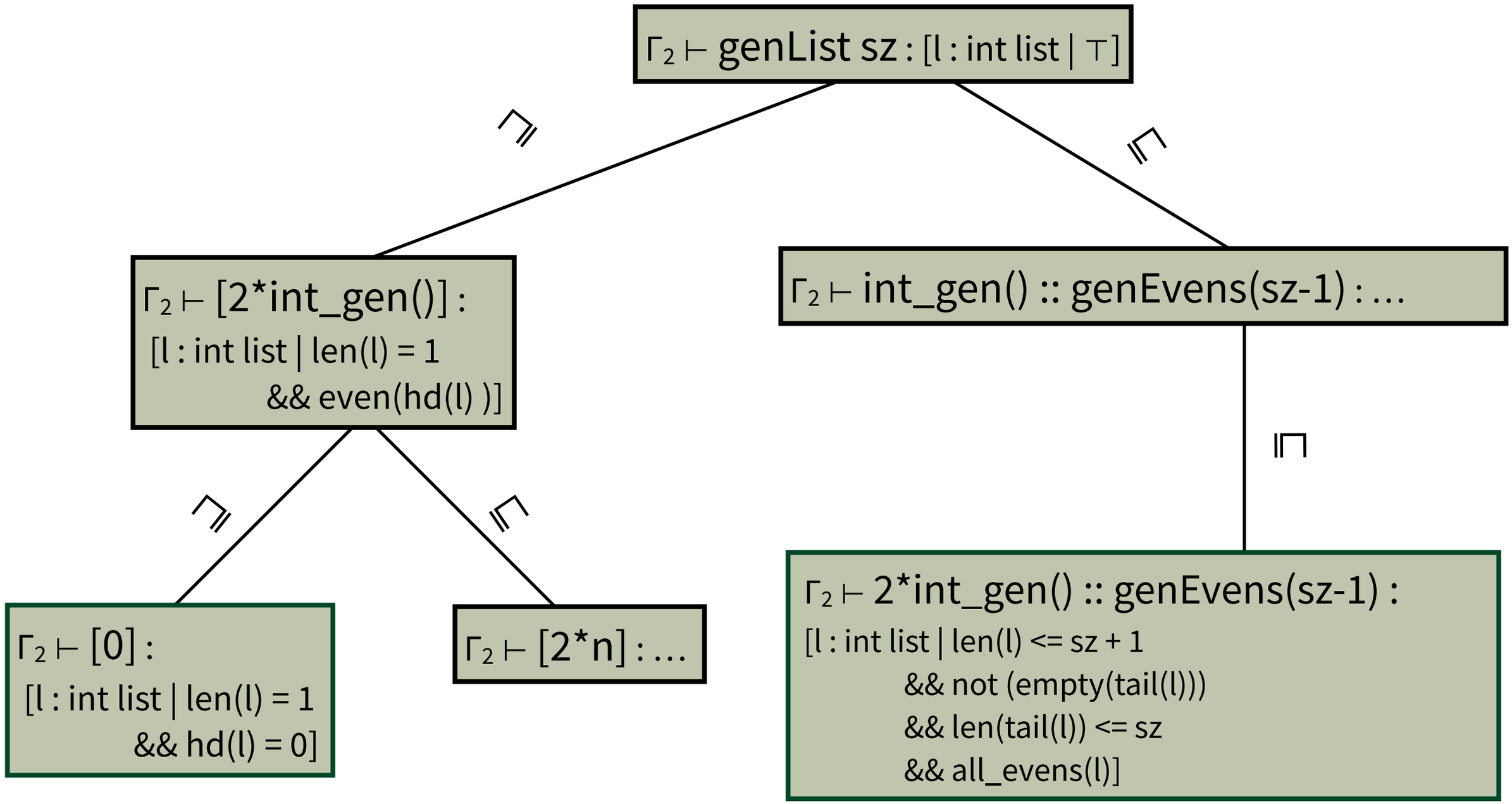
$\tau_{missing\_1}$ <: τ_7 ✓

$\tau_{missing\_1}$ <: τ_8 ✓

Some terms are a **sub-type** of our **missing coverage**, thus could repair the hole

Extract the optimal repair

Attempt to Repair Holes



Attempt to Repair Holes

$\Gamma_2 \vdash$ genList sz : [l : int list | $\top$]

$\Gamma_2 \vdash$ int_gen() :: genEvens(sz-1) : …

$\Gamma_2 \vdash \square$: [l : int list | 0 < len(l) <= sz+1 && all_evens(l)]

$\Gamma_2 \vdash$ [2*int_gen()] : [l : int list | len(l) = 1 && even(hd(l))]

$\Gamma_2 \vdash$ [0] : [l : int list | len(l) = 1 && hd(l) = 0]

$\Gamma_2 \vdash$ [2*n] : …

$\Gamma_2 \vdash$ 2*int_gen() :: genEvens(sz-1) : [l : int list | len(l) <= sz + 1 && not (empty(tail(l))) && len(tail(l)) <= sz && all_evens(l)]

That was cool and all but show me some numbers!

Evaluation: Benchmarks

4.5k lines of  (https://github.com/Pat-Lafon/Cobb)

Evaluate Cobb on a variety of :

- Datatypes: Lists, Trees, Red-BlackTrees, Simply Typed Lambda Calculus
- Properties: Sized, Sorted, Duplicate, Unique, Valid, Well-Typed
- Typing contexts: Branch conditions, Local variables

Evaluation: Benchmarks

4.5k lines of  (https://github.com/Pat-Lafon/Cobb)

Evaluate Cobb on a variety of :

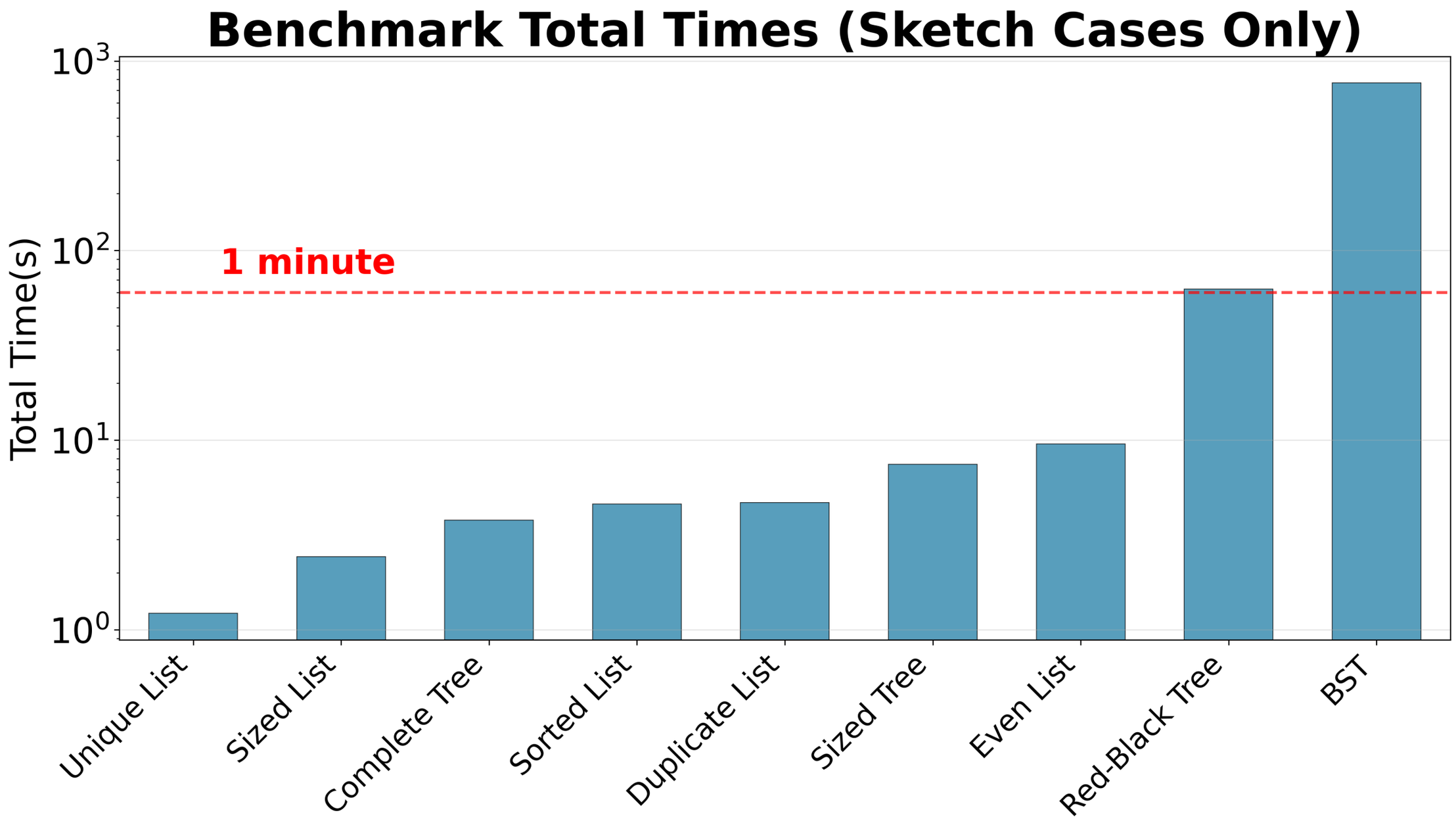
- Datatypes: Lists, Trees, Red-BlackTrees, Simply Typed Lambda Calculus
- Properties: Sized, Sorted, Duplicate, Unique, Valid, Well-Typed
- Typing contexts: Branch conditions, Local variables

Using benchmarks of generators from the literature, created a set of incomplete generator variations

A “Sketch” is then the most general form, only simple control flow and holes

Evaluation: Cobb is pretty fast!

Cobb satisfies the precise coverage < 1 minute



Evaluation: Cobb vs **Coverage-only** repair?

Trivial solution: the default generator is always coverage complete!

```
let rec even_list_gen (s : int) : int list =
 if s == 0 then Γ1 ⊢ □ : τmissing_1
 else Γ2 ⊢ □ : τmissing_2
```

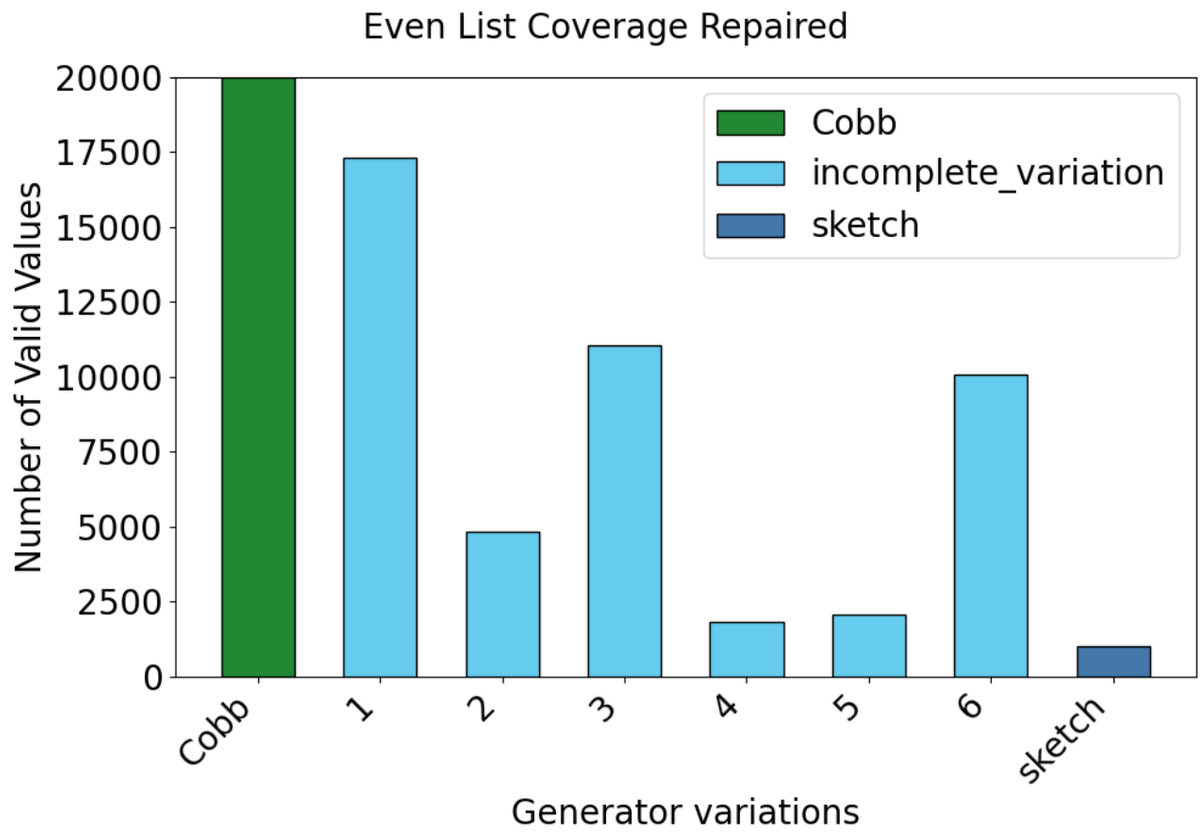
```
let rec even_list_gen (s : int) : int list =
 if s == 0 then default_list_gen ()
 else default_list_gen ()
```

Evaluation: Cobb vs **Coverage-only** repair?

Trivial solution: the default generator is always coverage complete!

Count how many outputs are valid (**higher** is better)

```
let rec even_list_gen
  (s : int) : int list =
  if s == 0 then
    default_list_gen ()
  else
    default_list_gen ()
```



See Paper(or come talk to me) for more!

- Simply Typed Lambda Calculus evaluation
- More detailed evaluation numbers/analysis
- Comparison with a safety-based repair strategy
- Comparison with dynamic test input generation
  - Dynamic Constraint solving is at least an order of magnitude slower
- Sensitivity to the set of user-provided components
- Soundness Theorem

Conclusion

- Problem: Users don’t want to write input generators

	Safety	Coverage
Type-Based Verification	Liquid Haskell Flux RefinedRust RefinedC refined(scala) ...	Poirot
Type-Based Synthesis	Synquid Cobalt ReSyn SuSLik ...	**Cobb (This work)**

Conclusion

- Problem: Users don’t want to write input generators
- Solution: Coverage Type-Guided Program Repair
  - Phase 1: Characterizes missing coverage
  - Phase 2: Uses coverage to guide repair

	Safety	Coverage
Type-Based Verification	Liquid Haskell Flux RefinedRust RefinedC refined(scala) ...	Poirot
Type-Based Synthesis	Synquid Cobalt ReSyn SuSLik ...	**Cobb (This work)**

Conclusion

- Problem: Users don't want to write input generators
- Solution: Coverage Type-Guided Program Repair
  - Phase 1: Characterizes missing coverage
  - Phase 2: Uses coverage to guide repair
- Results: Efficient, one-time synthesis task
- https://github.com/Pat-Lafon/Cobb

	Safety	Coverage
Type-Based Verification	Liquid Haskell Flux RefinedRust RefinedC refined(scala) ...	Poirot
Type-Based Synthesis	Synquid Cobalt ReSyn SuSLik ...	**Cobb (This work)**