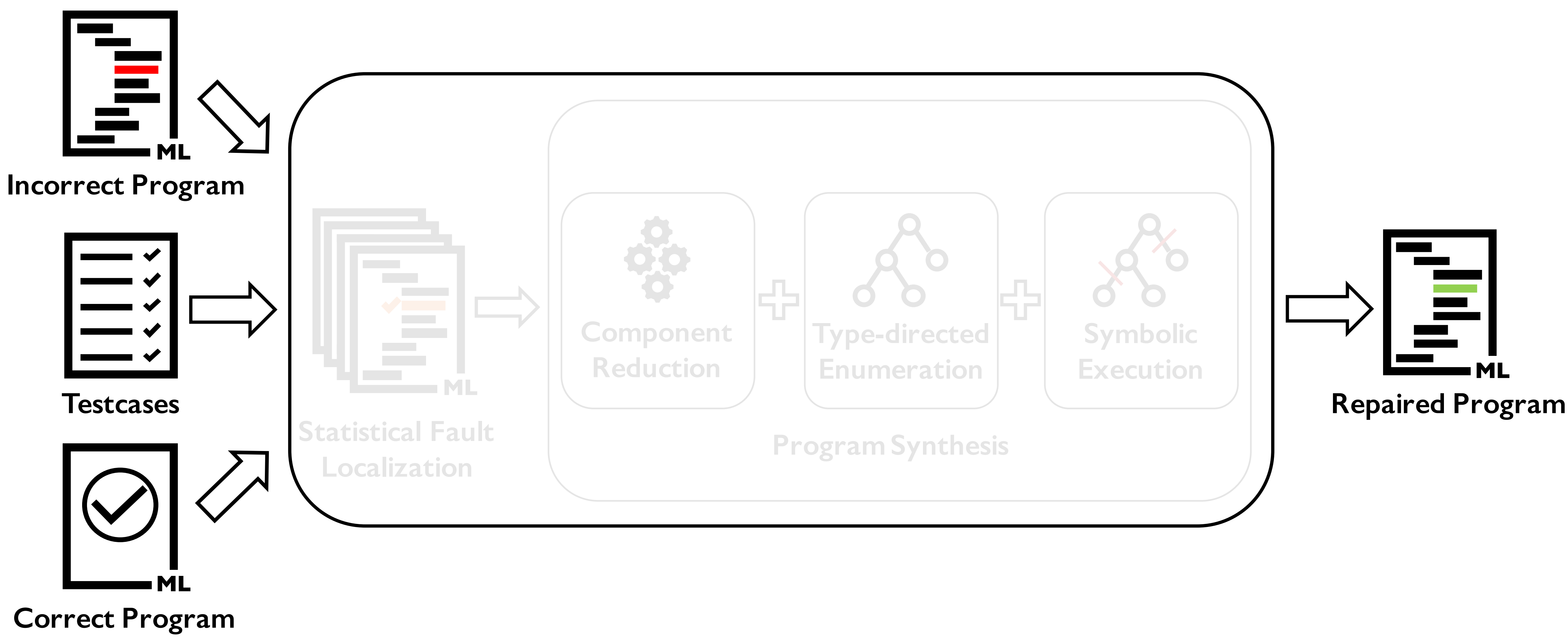
함수형 프로그래밍 과제 자동 채점 및 피드백 생성 시스템

2019.08.26

고려대학교 소프트웨어 분석 연구실
송도원

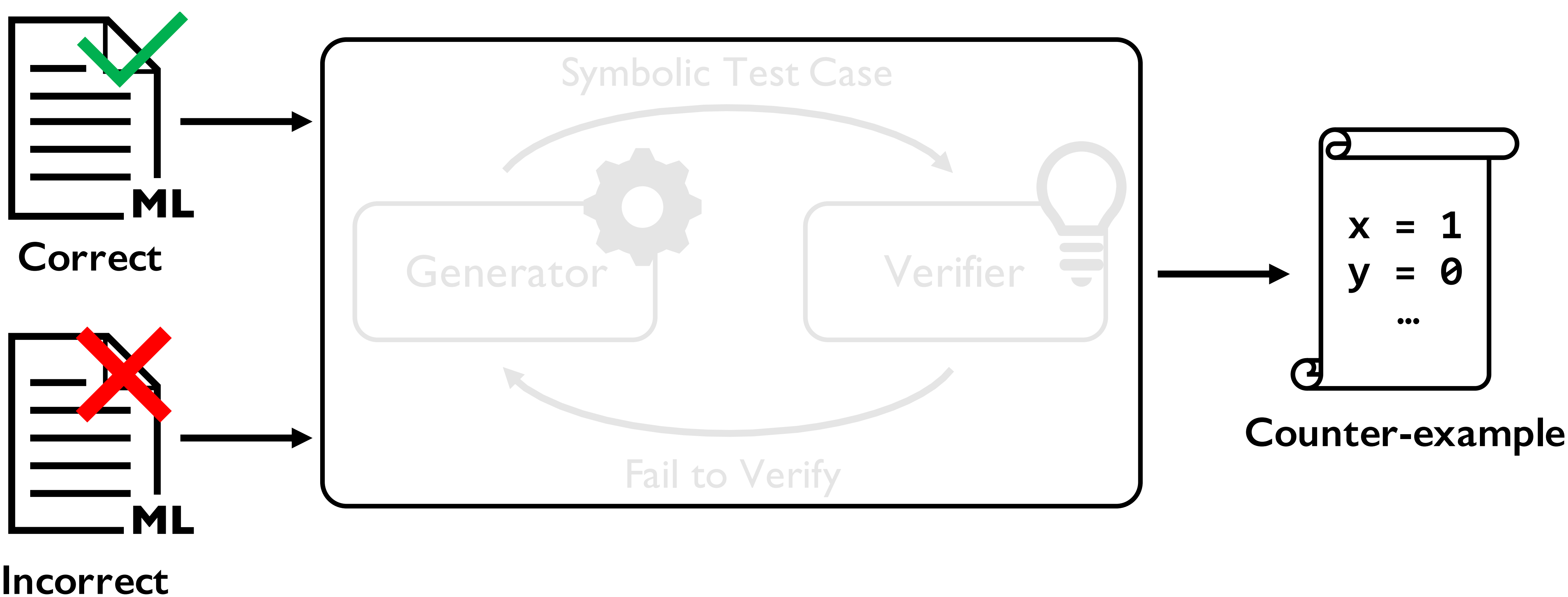
Today’s Talk: Part I

- Automatically feedback generation system for logical errors in functional programming assignment.



Today's Talk: Part2

- Automatic counter-example generation to detect incorrect submissions without human-designed test cases.



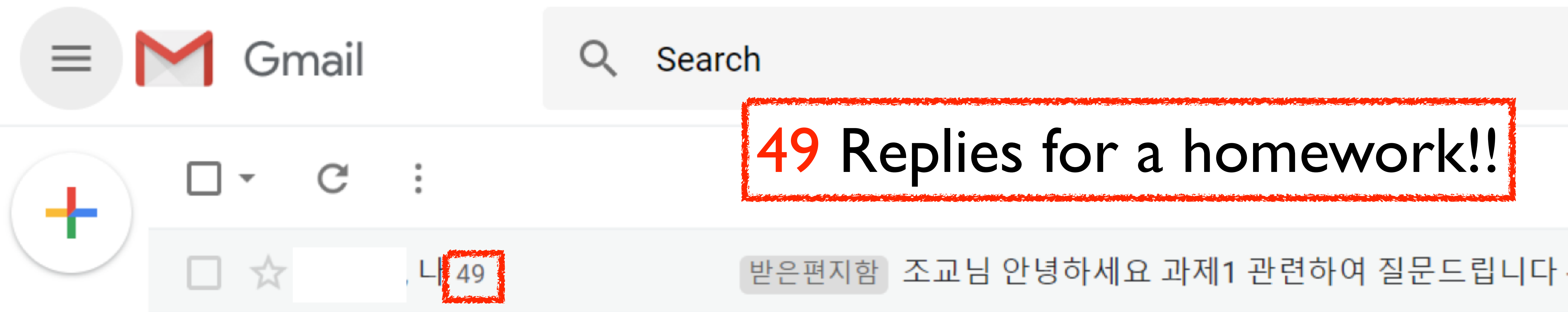
Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments

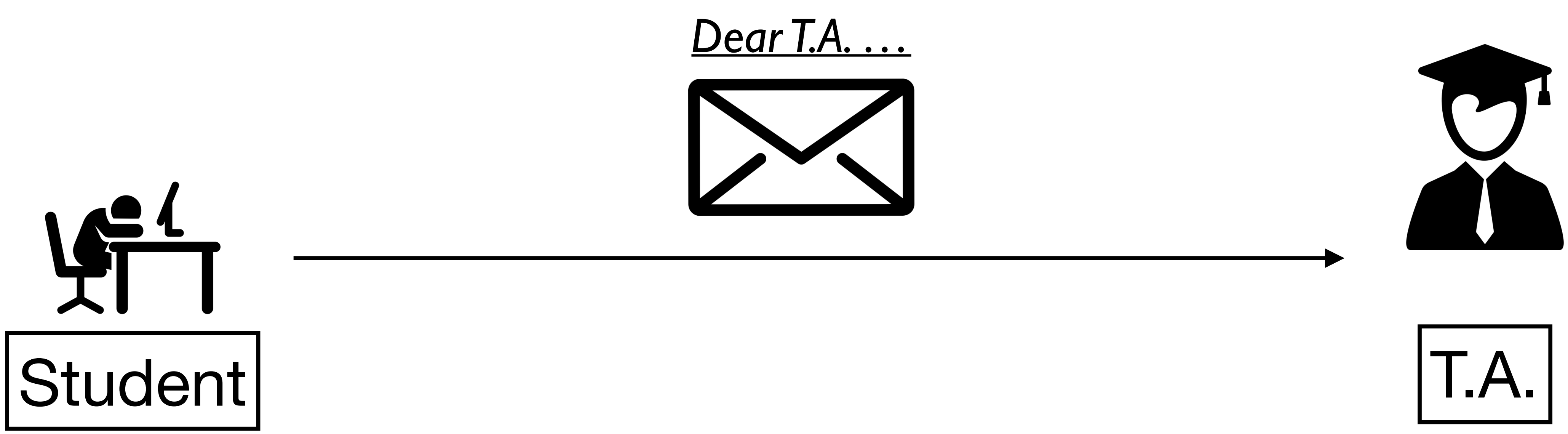
Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh
Korea University

9 November 2018
OOPSLA`18 @ Boston, U.S.A.

Motivation

- T.A. experience in functional programming course.
- A lot of e-mails about assignments





Motivation

Student's implementation:

```
type aexp =
   |CONST of int
   | VAR of string
   | POWER of string * int
   | TIMES of aexp list
   | SUM of aexp list

type env = (string * int * int) list

let diff : aexp * string -> aexp
= fun (aexp, x) ->

   let rec deployEnv : env -> int -> aexp list
   = fun env flag ->
   match env with
   | hd::tl ->
   (
    match hd with
    |(x, c, p) ->
      if (flag = 0 && c = 0) then deployEnv tl flag
      else if (x = "const" && flag = 1 && c = 1) then deployEnv tl flag
      else if (p = 0) then (CONST c)::(deployEnv tl flag)
      else if (c = 1 && p = 1) then (VAR x)::(deployEnv tl flag)
      else if (p = 1) then TIMES[CONST c; VAR x]::(deployEnv tl flag)
      else if (c = 1) then POWER(x, p)::(deployEnv tl flag)
      else TIMES [CONST c; POWER(x, p)]::(deployEnv tl flag)
 )
 | [] -> []
 in

 let rec updateEnv : (string * int * int) -> env -> int -> env
 = fun elem env flag ->
 match env with
 | (hd::tl) ->
    (
    match hd with
    | (x, c, p) ->
      (
      match elem with
      |(x2, c2, p2) ->
        if (flag = 0) then
          if (x = x2 && p = p2) then (x, (c + c2), p)::tl
          else hd::(updateEnv elem tl flag)
        else
          if (x = x2) then (x, (c*c2), (p + p2))::tl
          else hd::(updateEnv elem tl flag)
      )
    )
 | [] -> elem::[]
 in

 let rec doDiff : aexp * string -> aexp
 = fun (aexp, x) ->
 match aexp with
 | CONST _ -> CONST 0
 | VAR v ->
    if (x = v) then CONST 1
    else CONST 0
 | POWER (v, p) ->
    if (p = 0) then CONST 0
    else if (x = v) then TIMES ((CONST p)::POWER (v, p-1)::[])
    else CONST 0
 | TIMES lst ->
    (
      match lst with
          (
            match (hd, diff_hd, tl, diff_tl) with
            | (CONST p, CONST s, [CONST r], CONST q) -> CONST (p*q + r*s)
            | (CONST p, _, _, CONST q) ->
              if (diff_hd = CONST 0 || tl = [CONST 0]) then CONST (p*q)
              else SUM [CONST(p*q); TIMES(diff_hd::tl)]
            | (_, CONST s, [CONST r], _) ->
              if (hd = CONST 0 || diff_tl = CONST 0) then CONST (r*s)
              else  SUM [TIMES [hd; diff_tl]; CONST(r*s)]
            | _ ->
              if (hd = CONST 0 || diff_tl = CONST 0) then TIMES(diff_hd::tl)
              else if (tl = [CONST 0] || diff_hd = CONST 0) then TIMES [hd; diff_tl]
              else SUM [TIMES [hd; diff_tl]; TIMES (diff_hd::tl)]
          )
        | [] -> CONST 0
    )
 | SUM lst -> SUM(List.map (fun aexp -> doDiff(aexp, x)) lst)
 in

 let rec simplify : aexp -> env -> int -> aexp list
 = fun aexp env flag ->
 match aexp with
 | SUM lst ->
    (
      match lst with
      | (CONST c)::tl -> simplify (SUM tl) (updateEnv ("const", c, 0) env 0) 0
      | (VAR x)::tl -> simplify (SUM tl) (updateEnv (x, 1, 1) env 0) 0
      | (POWER (x, p))::tl -> simplify (SUM tl) (updateEnv (x, 1, p) env 0) 0
      | (SUM lst)::tl -> simplify (SUM (List.append lst tl)) env 0
      | (TIMES lst)::tl ->
        (
          let l = simplify (TIMES lst) [] 1 in
          match l with
          | h::t ->
            if (t = []) then List.append l (simplify (SUM tl) env 0)
            else List.append (TIMES l::[]) (simplify (SUM tl) env 0)
          | [] -> []
        )
      | [] -> deployEnv env 0
    )
 | TIMES lst ->
    (
      match lst with
      | (CONST c)::tl -> simplify (TIMES tl) (updateEnv ("const", c, 0) env 1) 1
      | (VAR x)::tl -> simplify (TIMES tl) (updateEnv (x, 1, 1) env 1) 1
      | (POWER (x, p))::tl -> simplify (TIMES tl) (updateEnv (x, 1, p) env 1) 1
      | (SUM lst)::tl ->
        (
          let l = simplify (SUM lst) [] 0 in
          match l with
          | h::t ->
            if (t = []) then List.append l (simplify (TIMES tl) env 1)
             else List.append (SUM l::[]) (simplify (TIMES tl) env 1)
           | [] -> []
         )
       | (TIMES lst)::tl -> simplify (TIMES (List.append lst tl)) env 1
       | [] -> deployEnv env 1
     )
 in

 let result = doDiff (aexp, x) in
 match result with
 | SUM _ -> SUM (simplify result [] 0)
 | TIMES _ -> TIMES (simplify result [] 1)
 | _ -> result
```

Solution:

```
let rec diff : aexp * string -> aexp
= fun (e, x) ->
  match e with
  | Const n -> Const 0
  | Var a -> if (a <> x) then Const 0 else Const 1
  | Power (a, n) -> if (a <> x) then Const 0 else Times [Const n; Power (a, n-1)]
  | Times l ->
    begin
    match l with
    | [] -> Const 0
    | hd::tl -> Sum [Times ((diff (hd, x))::tl); Times [hd; diff (Times tl, x)]]
    end
  | Sum l -> Sum (List.map (fun e -> diff (e,x)) l)
```

TA:
Hard to provide feedback!

Students:
Solution is meaningless…

Goal

Student's implementation:

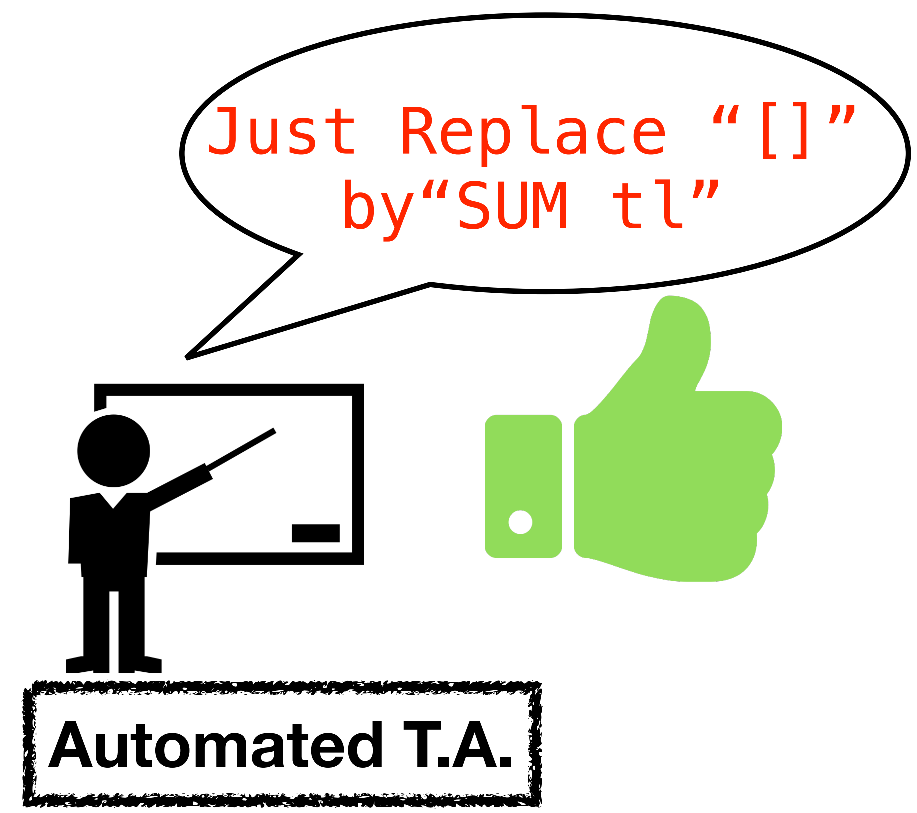
```
type aexp =
   |CONST of int
   | VAR of string
   | POWER of string * int
   | TIMES of aexp list
   | SUM of aexp list

type env = (string * int * int) list

let diff : aexp * string -> aexp
= fun (aexp, x) ->

   let rec deployEnv : env -> int -> aexp list
   = fun env flag ->
   match env with
   | hd::tl ->
   (
    match hd with
    |(x, c, p) ->
      if (flag = 0 && c = 0) then deployEnv tl flag
      else if (x = "const" && flag = 1 && c = 1) then deployEnv tl flag
      else if (p = 0) then (CONST c)::(deployEnv tl flag)
      else if (c = 1 && p = 1) then (VAR x)::(deployEnv tl flag)
      else if (p = 1) then TIMES[CONST c; VAR x]::(deployEnv tl flag)
      else if (c = 1) then POWER(x, p)::(deployEnv tl flag)
      else TIMES [CONST c; POWER(x, p)]::(deployEnv tl flag)
 )
 | [] -> []
 in

 let rec updateEnv : (string * int * int) -> env -> int -> env
 = fun elem env flag ->
 match env with
 | (hd::tl) ->
   (
   match hd with
   | (x, c, p) ->
     (
     match elem with
     |(x2, c2, p2) ->
       if (flag = 0) then
         if (x = x2 && p = p2) then (x, (c + c2), p)::tl
         else hd::(updateEnv elem tl flag)
       else
         if (x = x2) then (x, (c*c2), (p + p2))::tl
         else hd::(updateEnv elem tl flag)
     )
   )
 | [] -> elem::[]
 in

 let rec doDiff : aexp * string -> aexp
 = fun (aexp, x) ->
 match aexp with
 | CONST _ -> CONST 0
 | VAR v ->
   if (x = v) then CONST 1
   else CONST 0
 | POWER (v, p) ->
   if (p = 0) then CONST 0
   else if (x = v) then TIMES ((CONST p)::POWER (v, p-1)::[])
   else CONST 0
 | TIMES lst ->
   (
     match lst with
```

```
          (
            match (hd, diff_hd, tl, diff_tl) with
            | (CONST p, CONST s, [CONST r], CONST q) -> CONST (p*q + r*s)
            | (CONST p, _, _, CONST q) ->
              if (diff_hd = CONST 0 || tl = [CONST 0]) then CONST (p*q)
              else SUM [CONST(p*q); TIMES(diff_hd::tl)]
            | (_, CONST s, [CONST r], _) ->
              if (hd = CONST 0 || diff_tl = CONST 0) then CONST (r*s)
              else  SUM [TIMES [hd; diff_tl]; CONST(r*s)]
            | _ ->
              if (hd = CONST 0 || diff_tl = CONST 0) then TIMES(diff_hd::tl)
              else if (tl = [CONST 0] || diff_hd = CONST 0) then TIMES [hd; diff_tl]
              else SUM [TIMES [hd; diff_tl]; TIMES (diff_hd::tl)]
          )
        | [] -> CONST 0
      )
  | SUM lst -> SUM(List.map (fun aexp -> doDiff(aexp, x)) lst)
  in

  let rec simplify : aexp -> env -> int -> aexp list
  = fun aexp env flag ->
  match aexp with
  | SUM lst ->
    (
      match lst with
      | (CONST c)::tl -> simplify (SUM tl) (updateEnv ("const", c, 0) env 0) 0
      | (VAR x)::tl -> simplify (SUM tl) (updateEnv (x, 1, 1) env 0) 0
      | (POWER (x, p))::tl -> simplify (SUM tl) (updateEnv (x, 1, p) env 0) 0
      | (SUM lst)::tl -> simplify (SUM (List.append lst tl)) env 0
      | (TIMES lst)::tl ->
        (
          let l = simplify (TIMES lst) [] 1 in
          match l with
          | h::t ->
            if (t = []) then List.append l (simplify (SUM tl) env 0)
            else List.append (TIMES l::[]) (simplify (SUM tl) env 0)
          | [] -> []
        )
      | [] -> deployEnv env 0
    )
  | TIMES lst ->
    (
      match lst with
      | (CONST c)::tl -> simplify (TIMES tl) (updateEnv ("const", c, 0) env 1) 1
      | (VAR x)::tl -> simplify (TIMES tl) (updateEnv (x, 1, 1) env 1) 1
      | (POWER (x, p))::tl -> simplify (TIMES tl) (updateEnv (x, 1, p) env 1) 1
      | (SUM lst)::tl ->
        (
          let l = simplify (SUM
          match l with
          | h::t ->
            if (t = []) then Li
             else List.append (S
           | [] -> []
        )
       | (TIMES lst)::tl -> simplify (TIMES (List.append lst tl)) env 1
       | [] -> deployEnv env 1
    )
  in

  let result = doDiff (aexp, x) in
  match result with
  | SUM _ -> SUM (simplify result [] 0)
  | TIMES _ -> TIMES (simplify result [] 1)
  | _ -> result
```

Time: 3.4 sec

Solution:

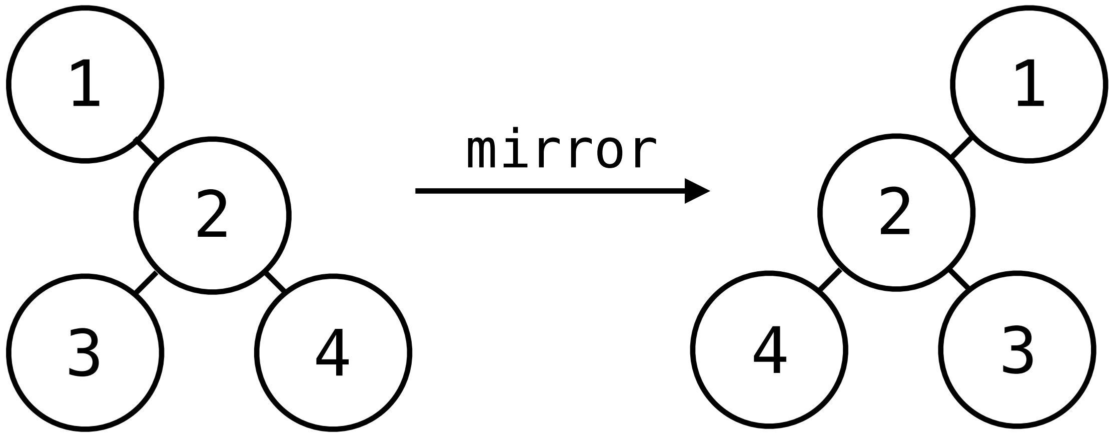
```
let rec diff : aexp * string -> aexp
= fun (e, x) ->
  match e with
  | Const n -> Const 0
  | Var a -> if (a <> x) then Const 0 else Const 1
  | Power (a, n) -> if (a <> x) then Const 0 else Times [Const n; Power (a, n-1)]
  | Times l ->
    begin
    match l with
    | [] -> Const 0
    | hd::tl -> Sum [Times ((diff (hd, x))::tl); Times [hd; diff (Times tl, x)]]
    end
  | Sum l -> Sum (List.map (fun e -> diff (e,x)) l)
```



Example1: Mirroring Tree

- Warming up!

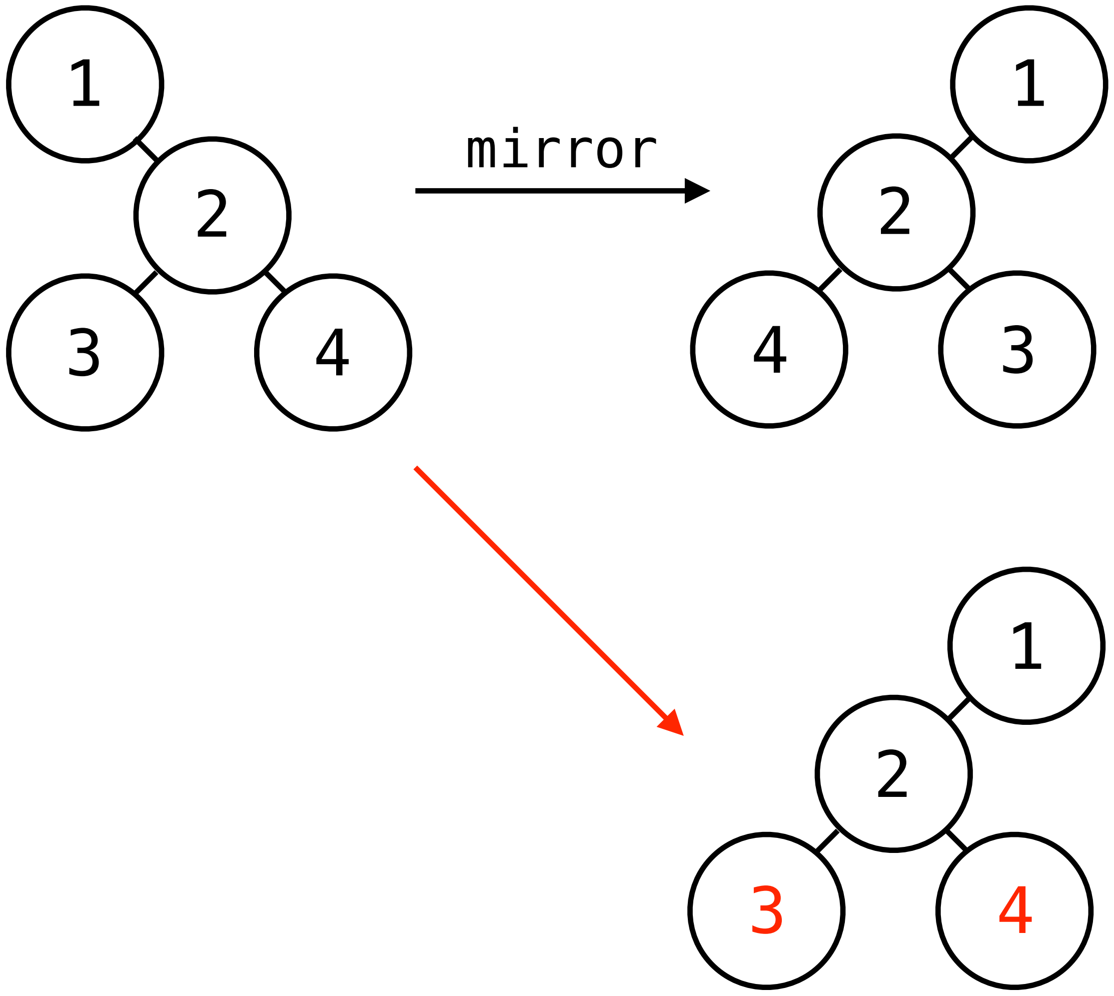
```
type btree =
  | Empty
  | Node of int * btree * btree

let rec mirror tree =
  match tree with
  | Empty -> Empty
  | Node (n,l,r) -> Node (n,r,l)
```



Example 1 : Mirroring Tree

- Warming up!

```
type btree =
  | Empty
  | Node of int * btree * btree

let rec mirror tree =
  match tree with
  | Empty -> Empty
  | Node (n,l,r) -> Node (n,r,l)
```



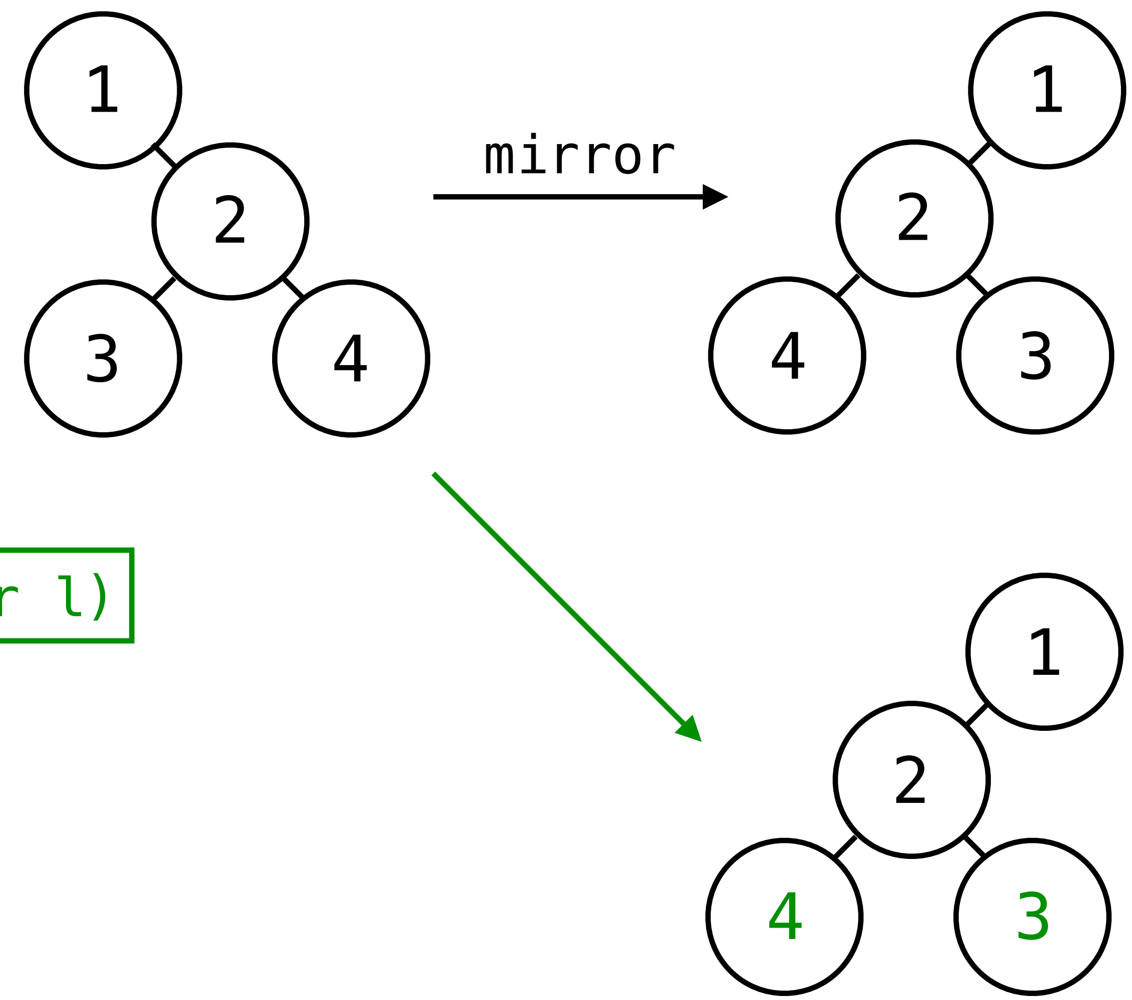
Example1: Mirroring Tree

- Warming up!

```
type btree =
  | Empty
  | Node of int * btree * btree

let rec mirror tree =
  match tree with
  | Empty -> Empty
  | Node (n,l,r) -> Node (n,r,l)
```

```
FixML: Node (n, mirror r, mirror l)
```

Time: 0.1 sec



Example2: Natural Numbers

- More complicated program

```
type nat =
  |ZERO
  |SUCC of nat

let rec natadd n1 n2 =
  match n1 with
  |ZERO -> ZERO
  |SUCC n -> SUCC (natadd n n2)

let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1’ ->
    SUCC( match n2 with
      | ZERO -> ZERO
      | SUCC ZERO -> SUCC ZERO
      | SUCC n2’ -> SUCC (natmul n1’ (natmul n1 n2’))
    )
```

```
Test cases :
  natmul (ZERO) (SUCC ZERO) = ZERO
  natmul (SUCC ZERO) (SUCC ZERO) = SUCC ZERO
  natmul (SUCC(SUCC ZERO)) (SUCC(SUCC(SUCC ZERO)))
    = SUCC(SUCC(SUCC(SUCC(SUCC(SUCC ZERO)))))
```

Example2: Natural Numbers

- More complicated program

```
type nat =
  |ZERO
  |SUCC of nat

let rec natadd n1 n2 =
  match n1 with
  |ZERO -> ZERO
  |SUCC n -> SUCC (natadd n n2)

let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1’ ->
   SUCC( match n2 with
     | ZERO -> ZERO
     | SUCC ZERO -> SUCC ZERO
     | SUCC n2’ -> SUCC (natmul n1’ (natmul n1 n2’))
   )
```

```
Test cases :
 natmul (ZERO) (SUCC ZERO) = ZERO
 natmul (SUCC ZERO) (SUCC ZERO) = SUCC ZERO
 natmul (SUCC(SUCC ZERO)) (SUCC(SUCC(SUCC ZERO)))
   = SUCC(SUCC(SUCC(SUCC(SUCC(SUCC ZERO)))))
```

Wrong formula:

$$\boxed{2 + (n_1 - 1) \times (n_1 \times (n_2 - 1))}$$

Example2: Natural Numbers

- More complicated program

```
type nat =
  |ZERO
  |SUCC of nat

let rec natadd n1 n2 =
  match n1 with
  |ZERO -> ZERO
  |SUCC n -> SUCC (natadd n n2)

let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1’ ->
   SUCC( match n2 with
     | ZERO -> ZERO
     | SUCC ZERO -> SUCC ZERO
     | SUCC n2’ -> SUCC (natmul n1’ (natmul n1 n2’))
   )
```

```
Test cases :
 natmul (ZERO) (SUCC ZERO) = ZERO
 natmul (SUCC ZERO) (SUCC ZERO) = SUCC ZERO
 natmul (SUCC(SUCC ZERO)) (SUCC(SUCC(SUCC ZERO)))
  = SUCC(SUCC(SUCC(SUCC(SUCC(SUCC ZERO)))))
```

Wrong formula:

$$\boxed{2 + (n_1 - 1) \times (n_1 \times (n_2 - 1))}$$

Correct formula:

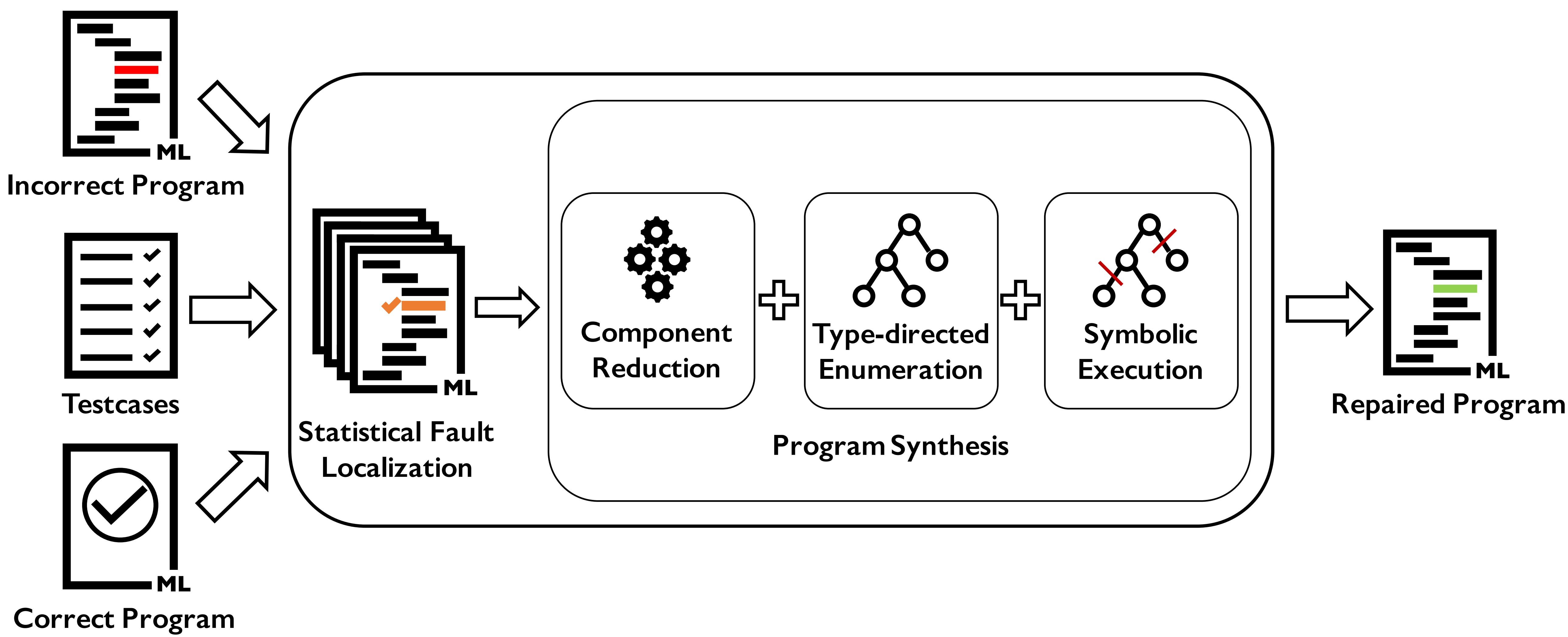
$$n_1 \times n_2 = \begin{cases} 0 & n_1 = 0 \\ \boxed{n_2 + (n_1 - 1) \times n_2 \quad n_1 \neq 0} \end{cases}$$

```
FixML:
natadd n2(natmul n1’ n2)
```

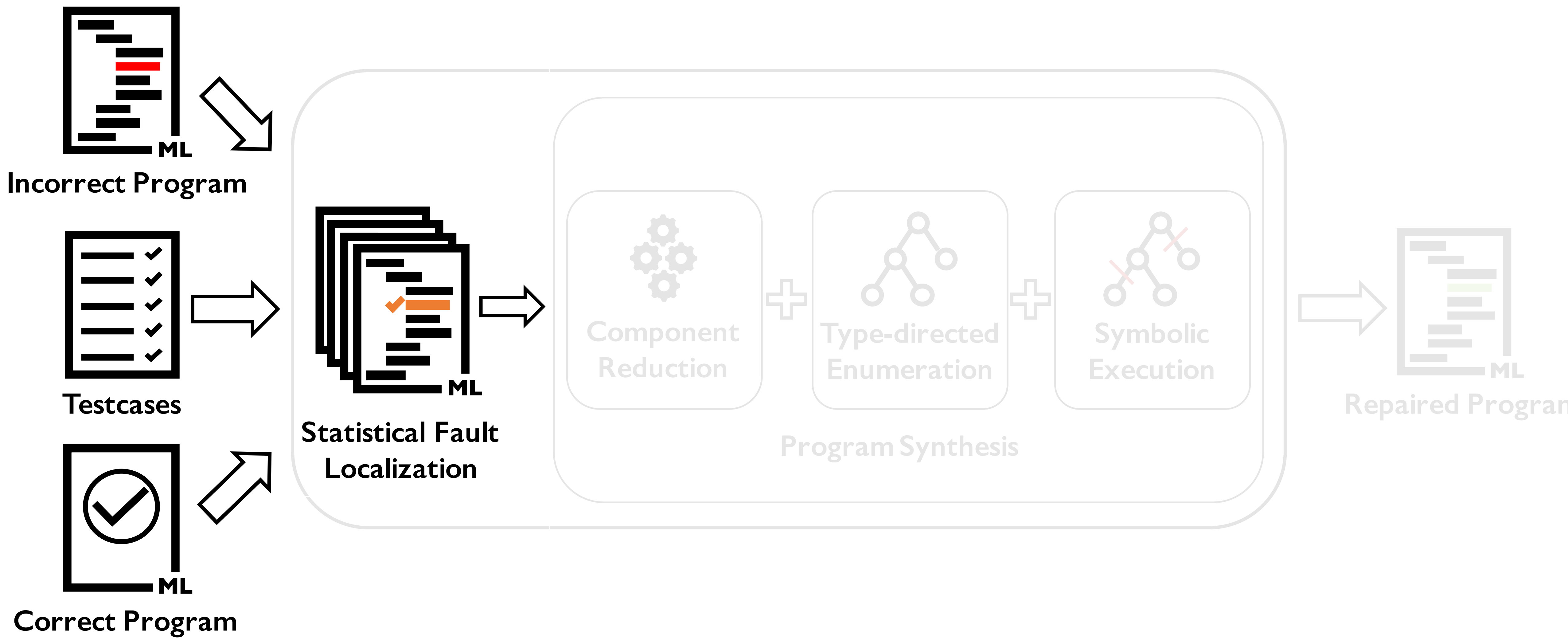
Time: 22 sec

FixML

- Given solution and test cases, our system automatically fixes the student submissions.



Error Localization



- Given buggy program and test cases, return a set of partial programs with suspicious score.

Statistical Fault Localization

Student’s program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1’ ->
    SUCC( match n2 with
      | ZERO -> ZERO
      | SUCC ZERO -> SUCC ZERO
      | SUCC n2’ -> SUCC (natmul n1’ (natmul n1 n2’))
    )
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC ZERO)) ZERO = ZERO
```

Statistical Fault Localization

Student's program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' ->
    SUCC( match n2 with
      | ZERO -> ZERO
      | SUCC ZERO -> SUCC ZERO
      | SUCC n2' -> SUCC (natmul n1' (natmul n1 n2'))
    )
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC ZERO)) ZERO = ZERO
```

The program satisfies the test case => Positive

Statistical Fault Localization

Student's program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' ->
    SUCC( match n2 with
      | ZERO -> ZERO
      | SUCC ZERO -> SUCC ZERO
      | SUCC n2' -> SUCC (natmul n1' (natmul n1 n2'))
    )
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC ZERO)) ZERO = ZERO
```

The program cannot satisfy the test case => Negative

Statistical Fault Localization

Student's program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' ->
    SUCC( match n2 with
      | ZERO -> ZERO
      | SUCC ZERO -> SUCC ZERO
      | SUCC n2' -> SUCC (natmul n1' (natmul n1 n2'))
    )
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC ZERO)) ZERO = ZERO
```

Only positive

Positive + negative

Only negative

Statistical Fault Localization

Student's program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' ->
   SUCC( match n2 with
     | ZERO -> ZERO
     | SUCC ZERO -> SUCC ZERO
     | SUCC n2' -> SUCC (natmul n1' (natmul n1 n2'))
   )
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC ZERO)) ZERO = ZERO
```

- Only positive
- Positive + negative
- Only negative

More negative, less positive => highly suspicious

Statistical Fault Localization

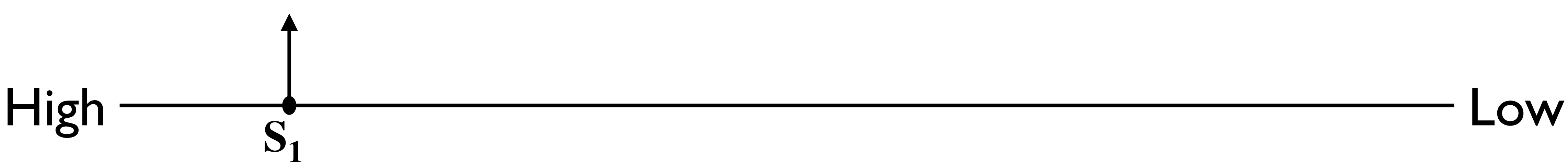
Student's program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' ->
    SUCC( match n2 with
      | ZERO -> ZERO
      | SUCC ZERO -> SUCC ZERO
      | SUCC n2' -> SUCC (natmul n1' (natmul n1 n2'))
    )
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC ZERO)) ZERO = ZERO
```

P_1

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```



Statistical Fault Localization

Student's program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' ->
    SUCC( match n2 with
      | ZERO -> ZERO
      | SUCC ZERO -> SUCC ZERO
      | SUCC n2' -> SUCC (natmul n1' (natmul n1 n2'))
    )
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC ZERO)) ZERO = ZERO
```

P_1

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

...

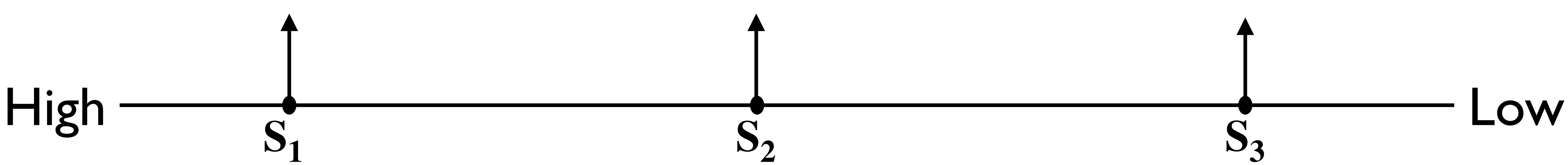
P_2

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> ?
  | SUCC n1' -> …
```

...

P_3

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ?
  | SUCC ZERO -> n2
  | SUCC n1' -> …
```



Statistical Fault Localization

Student's program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' ->
    SUCC( match n2 with
      | ZERO -> ZERO
      | SUCC ZERO -> SUCC ZERO
      | SUCC n2' -> SUCC (natmul n1' (natmul n1 n2'))
    )
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC ZERO)) ZERO = ZERO
```

Return a set of scored partial programs

P_1

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

…

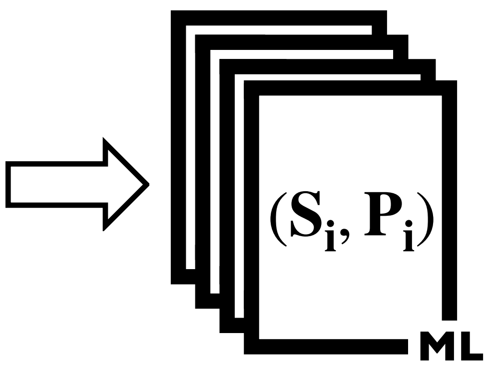
P_2

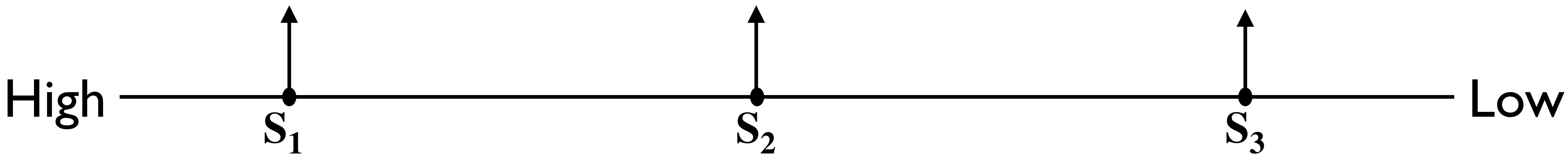
```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> ?
  | SUCC n1' -> …
```

…

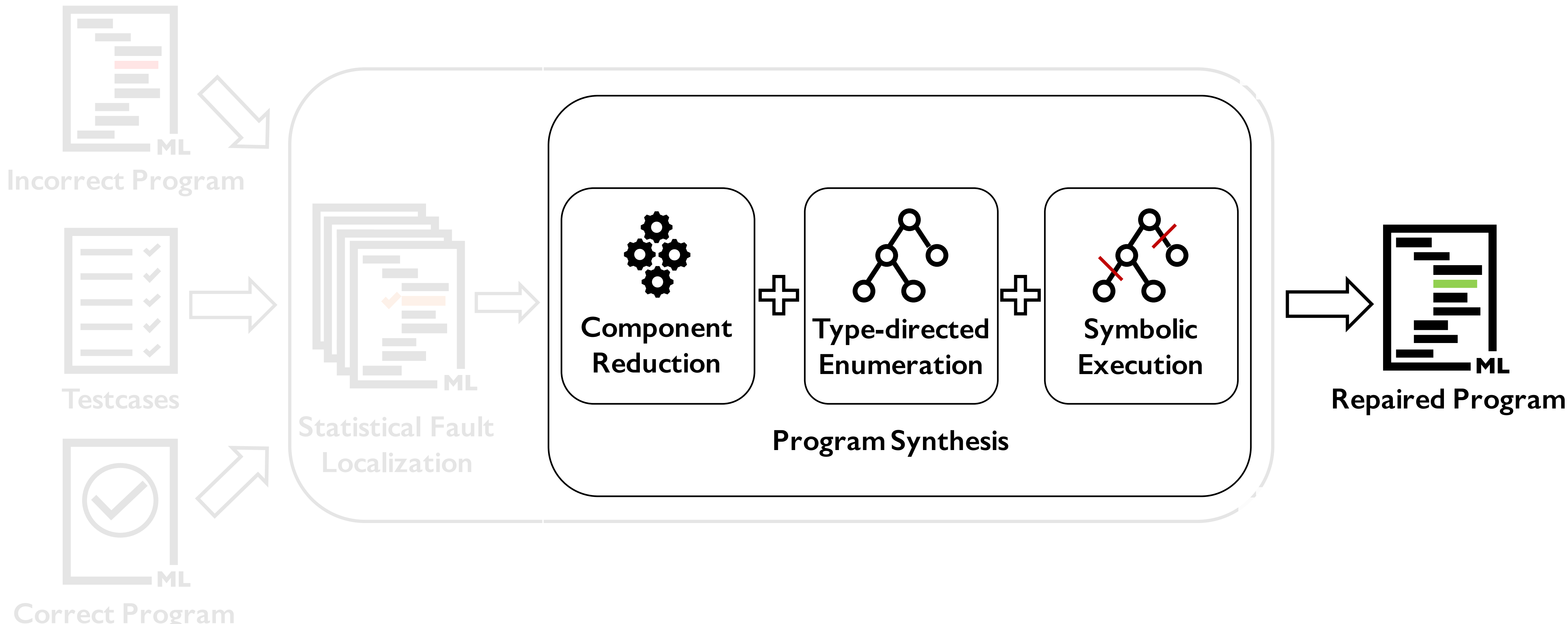
P_3

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ?
  | SUCC ZERO -> n2
  | SUCC n1' -> …
```





Program Synthesis



- Given the set of scored partial program, it generates a repaired program.

Baseline: Enumerative Search

- Enumerating all expressions in the language

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

Baseline: Enumerative Search

- Enumerating all expressions in the language

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> ?
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> SUCC ?
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> (fun x -> ?)
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> n1
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> if ? then ? else ?
```

...

Baseline: Enumerative Search

- Enumerating all expressions in the language

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1' -> SUCC ?
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1' -> (fun x -> ?)
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1' -> n1
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1' -> if ? then ? else ?
```

...

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> SUCC ZERO
  | SUCC n1' -> SUCC (ZERO)
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> SUCC ZERO
  | SUCC n1' -> SUCC (n1')
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> SUCC ZERO
  | SUCC n1' -> SUCC (if ? then ? else ?)
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> SUCC ZERO
  | SUCC n1' -> SUCC (? ?)
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> SUCC ZERO
  | SUCC n1' -> SUCC (true)
```

...

Extremely inefficient!

State-of-the-art: Type-directed Search

- Searching only well-typed program

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> ?
```

Hole type : nat

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> SUCC ?
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> (fun x -> ?)
```

Expression type : t' -> t’

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> n1
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> if ? then ? else ?
```

State-of-the-art: Type-directed Search

- Searching only well-typed program

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> ?
```

Hole type : nat

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> SUCC ?
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> (fun x -> ?)
```

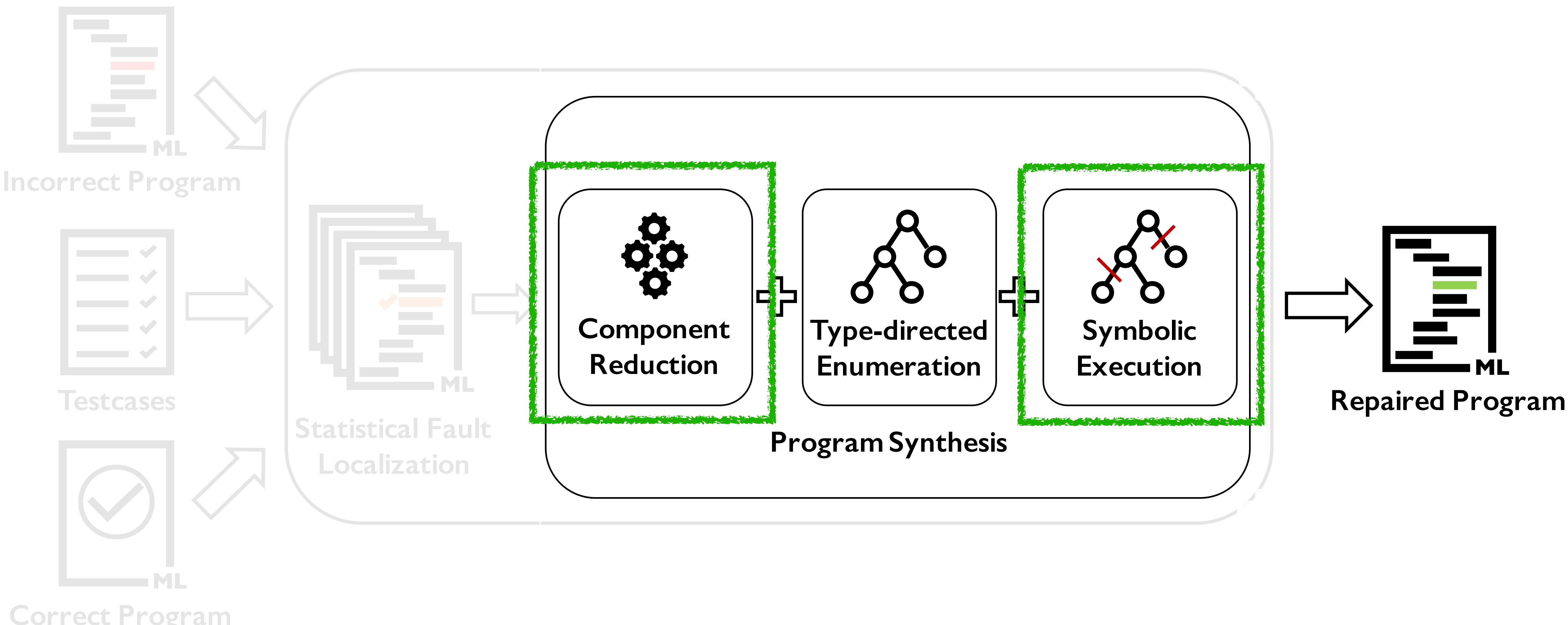
Expression type : t' -> t’

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> n1
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO | SUCC ZERO -> n2
  | SUCC n1’ -> if ? then ? else ?
```

Still inefficient in our cases!

Our Solution



- Component reduction
  - Syntactic component reduction
  - Variable component reduction
- Pruning with symbolic execution

Technique 1: Syntactic Component Reduction

- Enumerating all expressions is very expensive

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

Language:

36 expressions

$$
\begin{array}{lll}
E & ::= & () \mid n \mid x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{str} \mid \lambda \mathsf{x}.\mathsf{E} \mid \mathsf{E}_1 + \mathsf{E}_2 \mid \mathsf{E}_1 - \mathsf{E}_2 \mid \mathsf{E}_1 \times \mathsf{E}_2 \mid \mathsf{E}_1/\mathsf{E}_2 \mid \mathsf{E}_1 \ \mathsf{mod}\ \mathsf{E}_2 \mid -\mathsf{E} \\
& \mid & \mathsf{not}\ E \mid E_1 \mid\mid E_2 \mid E_1 \ \&\& E_2 \mid E_1 < E_2 \mid E_1 > E_2 \mid E_1 \leq E_2 \mid E_1 \geq E_2 \mid E_1 = E_2 \mid E_1 <> E_2 \\
& \mid & E_1\ E_2 \mid E_1 :: E_2 \mid E_1 @ E_2 \mid E_1 \hat{}\ E_2 \mid \mathsf{raise}\ E \mid (E_1, \ldots, E_k) \mid [E_1; \ldots; E_k] \\
& \mid & \mathsf{if}\ E_1\ E_2\ E_3 \mid c(E_1, \ldots, E_k) \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \mid \mathsf{let\ rec}\ f(x) = E_1\ \mathsf{in}\ E_2 \\
& \mid & \mathsf{let}\ x_1 = E_1\ \mathsf{and} \ldots \mathsf{and}\ x_k = E_k\ \mathsf{in}\ E \mid \mathsf{let\ rec}\ f_1(x_1) = E_1\ \mathsf{and} \ldots \mathsf{and}\ f_k(x_k) = E_k\ \mathsf{in}\ E \\
& \mid & \mathsf{match}\ E\ \mathsf{with}\ p_1 \rightarrow E_1 \mid \cdots \mid p_k \rightarrow E_k \\
& \mid & \square
\end{array}
$$

Technique 1: Syntactic Component Reduction

- Enumerating all expressions is very expensive

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

Language:

36 expressions

$$
\begin{array}{lll}
E & ::= & () \mid n \mid x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{str} \mid \lambda x.E \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid E_1 / E_2 \mid E_1 \bmod E_2 \mid -E \\
& \mid & \mathsf{not}\ E \mid E_1 \mid\mid E_2 \mid E_1\ \&\&E_2 \mid E_1 < E_2 \mid E_1 > E_2 \mid E_1 \le E_2 \mid E_1 \ge E_2 \mid E_1 = E_2 \mid E_1 <> E_2 \\
& \mid & E_1\ E_2 \mid E_1 :: E_2 \mid E_1 @ E_2 \mid E_1 \hat{}\ E_2 \mid \mathsf{raise}\ E \mid (E_1, \ldots, E_k) \mid [E_1; \ldots; E_k] \\
& \mid & \mathsf{if}\ E_1\ E_2\ E_3 \mid c(E_1, \ldots, E_k) \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \mid \mathsf{let\ rec}\ f(x) = E_1\ \mathsf{in}\ E_2 \\
& \mid & \mathsf{let}\ x_1 = E_1\ \mathsf{and} \ldots \mathsf{and}\ x_k = E_k\ \mathsf{in}\ E \mid \mathsf{let\ rec}\ f_1(x_1) = E_1\ \mathsf{and} \ldots \mathsf{and}\ f_k(x_k) = E_k\ \mathsf{in}\ E \\
& \mid & \mathsf{match}\ E\ \mathsf{with}\ p_1 \to E_1 \mid \cdots \mid p_k \to E_k \\
& \mid & \square
\end{array}
$$

Solution:

```
let rec natmul n1 n2 =
  match n1 with
  |ZERO -> ZERO
  | SUCC n1' -> natadd n2 (natmul n1' n2)
```

Observation:
Although the implementations are very different, used components are similar.

Technique 1: Syntactic Component Reduction

- Enumerating all expressions is very expensive

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

Language:

4

~~36~~ expressions

$$
\begin{aligned}
E \quad ::= \quad & () \mid n \mid \boxed{x} \mid \text{true} \mid \text{false} \mid \text{str} \mid \lambda x.E \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid E_1/E_2 \mid E_1 \text{ mod } E_2 \mid -E \\
\mid \quad & \text{not } E \mid E_1 \mid\mid E_2 \mid E_1 \&\& E_2 \mid E_1 < E_2 \mid E_1 > E_2 \mid E_1 \le E_2 \mid E_1 \ge E_2 \mid E_1 = E_2 \mid E_1 <> E_2 \\
\mid \quad & \boxed{E_1\ E_2} \mid E_1 :: E_2 \mid E_1 @ E_2 \mid E_1 \hat{\ } E_2 \mid \text{raise } E \mid (E_1, \ldots, E_k) \mid [E_1; \ldots; E_k] \\
\mid \quad & \text{if } E_1\ E_2\ E_3 \mid \boxed{c(E_1, \ldots, E_k)} \mid \text{let } x = E_1 \text{ in } E_2 \mid \text{let rec } f(x) = E_1 \text{ in } E_2 \\
\mid \quad & \text{let } x_1 = E_1 \text{ and} \ldots \text{and } x_k = E_k \text{ in } E \mid \text{let rec } f_1(x_1) = E_1 \text{ and} \ldots \text{and } f_k(x_k) = E_k \text{ in } E \\
\mid \quad & \boxed{\text{match } E \text{ with } p_1 \to E_1 \mid \cdots \mid p_k \to E_k} \\
\mid \quad & \square
\end{aligned}
$$

Solution:

```
let rec natmul n1 n2 =
  match n1 with
  |ZERO -> ZERO
  | SUCC n1' -> natadd n2 (natmul n1' n2)
```

Enumerating expressions only used in solution

Technique 2: Variable Component Reduction

- Enumerating all variables generates redundant programs.

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

Bound Variable: {natmul, n1, n2, n1'}

Technique 2: Variable Component Reduction

- Enumerating all variables generates redundant programs.

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

Bound Variable: {`natmul, n1, n2, n1'`}

Enumeration

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> SUCC n1'
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> n1
```

Technique 2: Variable Component Reduction

- Enumerating all variables generates redundant programs.

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

Bound Variable: {`natmul, n1, n2, n1'`}

```
n1 = SUCC n1'
```

Enumeration

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> SUCC n1'
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> n1
```

Semantically equivalent programs

Technique 2: Variable Component Reduction

- Enumerating all variables generates redundant programs.

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> ?
```

Enumeration

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> SUCC n1'
```

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> n1
```

Bound Variable: {`natmul`, ~~`n1`~~, `n2`, `n1'`}

`n1 = SUCC n1'`

Data-flow analysis:
`n1` can be always expressed with `n1'`

Choosing the minimal set of variables through data-flow analysis

Technique 3: Pruning via symbolic execution

- There are programs eventually inconsistent with the test cases

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1’ -> SUCC ?
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC (ZERO))) ZERO = ZERO
```

Technique 3: Pruning via symbolic execution

- There are programs eventually inconsistent with the test cases

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> SUCC ?
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC (ZERO))) ZERO = ZERO
```

Symbolic execution:

```
natmul (SUCC (SUCC (ZERO))) ZERO => (SUCC ?)
```

Technique 3: Pruning via symbolic execution

- There are programs eventually inconsistent with the test cases

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
  | ZERO -> ZERO
  | SUCC ZERO -> n2
  | SUCC n1' -> SUCC ?
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC (ZERO))) ZERO = ZERO
```

Symbolic execution:

```
natmul (SUCC (SUCC (ZERO))) ZERO => (SUCC ?)
```

SAT (`SUCC ? = ZERO`) => UNSAT

Technique 3: Pruning via symbolic execution

- There are programs eventually inconsistent with the test cases

Partial Program:

```
let rec natmul n1 n2 =
  match n1 with
   | ZERO -> ZERO
   | SUCC ZERO -> n2
   | SUCC n1' -> SUCC ?
```

```
Test cases :
natmul ZERO (SUCC ZERO) = ZERO

natmul (SUCC ZERO) (SUCC ZERO) = (SUCC ZERO)

natmul (SUCC (SUCC (ZERO))) ZERO = ZERO
```

Symbolic execution:

```
natmul (SUCC (SUCC (ZERO))) ZERO => (SUCC ?)
```

SAT (SUCC ? = ZERO) => UNSAT

Safely pruning the partial programs

Evaluation

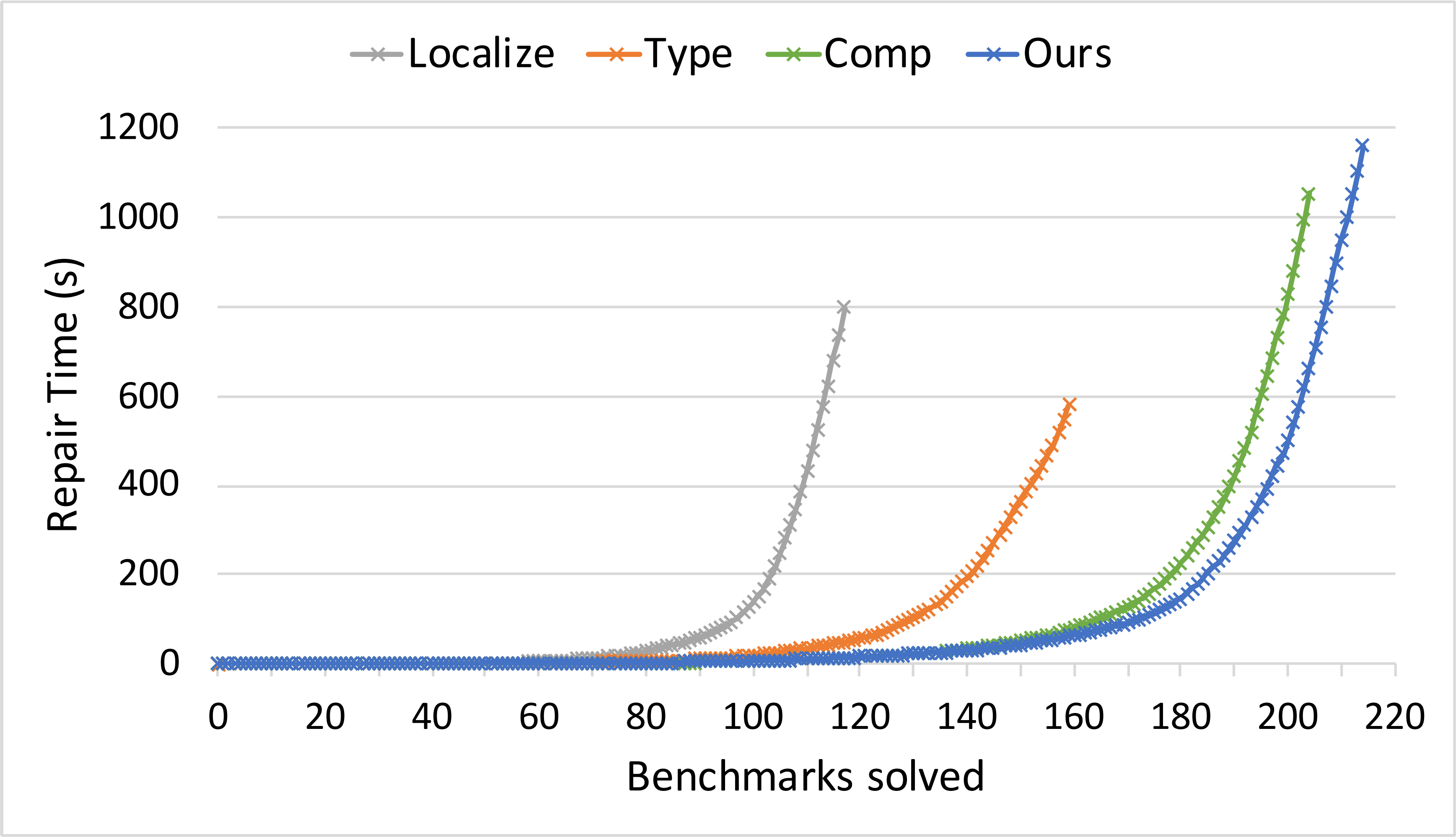
- Evaluated on 497 programs written in OCaml with logical errors from 13 assignments.
- Various task from introductory to advanced (2-154 lines) problems
- Conducted user study with 18 under-graduate students.

Effectiveness

No	Problem Description	#P	#T	LOC (min-max)	Time	Fix Rate (#Fix)	
1	Filtering elements satisfying a predicate in a list	3	10	6 (6-7)	13.0	100% (3)	Introductory Fix: 89% Time: 2.5 sec
2	Finding a maximum element in a list	32	10	8 (4-14)	0.2	100% (32)	
3	Mirroring a binary tree	9	10	11 (9-14)	0.1	89% (8)	
4	Checking membership in a binary tree	15	17	11 (9-18)	5.2	80% (12)	
5	Computing $\sum_{i=j}^{k} f(i)$ for j, k, and f	23	11	5 (2-9)	4.2	78% (18)	
6	Adding and multiplying user-defined natural numbers	34	10	20 (13-50)	20.6	59% (20)	Intermediate Fix: 48% Time: 11.6 sec
7	Finding the number of ways of coin-changes	9	10	21 (6-35)	2.6	44% (4)	
8	Composing functions	28	12	7 (3-19)	5.5	43% (12)	
9	Implementing a leftist heap using a priority queue	20	13	43 (33-72)	2.6	40% (8)	
10	Evaluating expressions and propositional formulas	101	17	32 (17-57)	1.2	39% (39)	Advanced Fix: 30% Time: 4.8 sec
11	Adding numbers in user-defined number system	14	10	52 (19-138)	7.0	36% (5)	
12	Deciding lambda terms are well-formed or not	86	11	30 (13-79)	1.3	26% (22)	
13	Differentiating algebraic expressions	123	17	36 (14-154)	11.4	25% (31)	
	Total / Average	497	158	27 (2-154)	5.4	43% (214)	

- Average time: 5.4 sec / Fix rate: 43%
- Generating patches for diverse problems

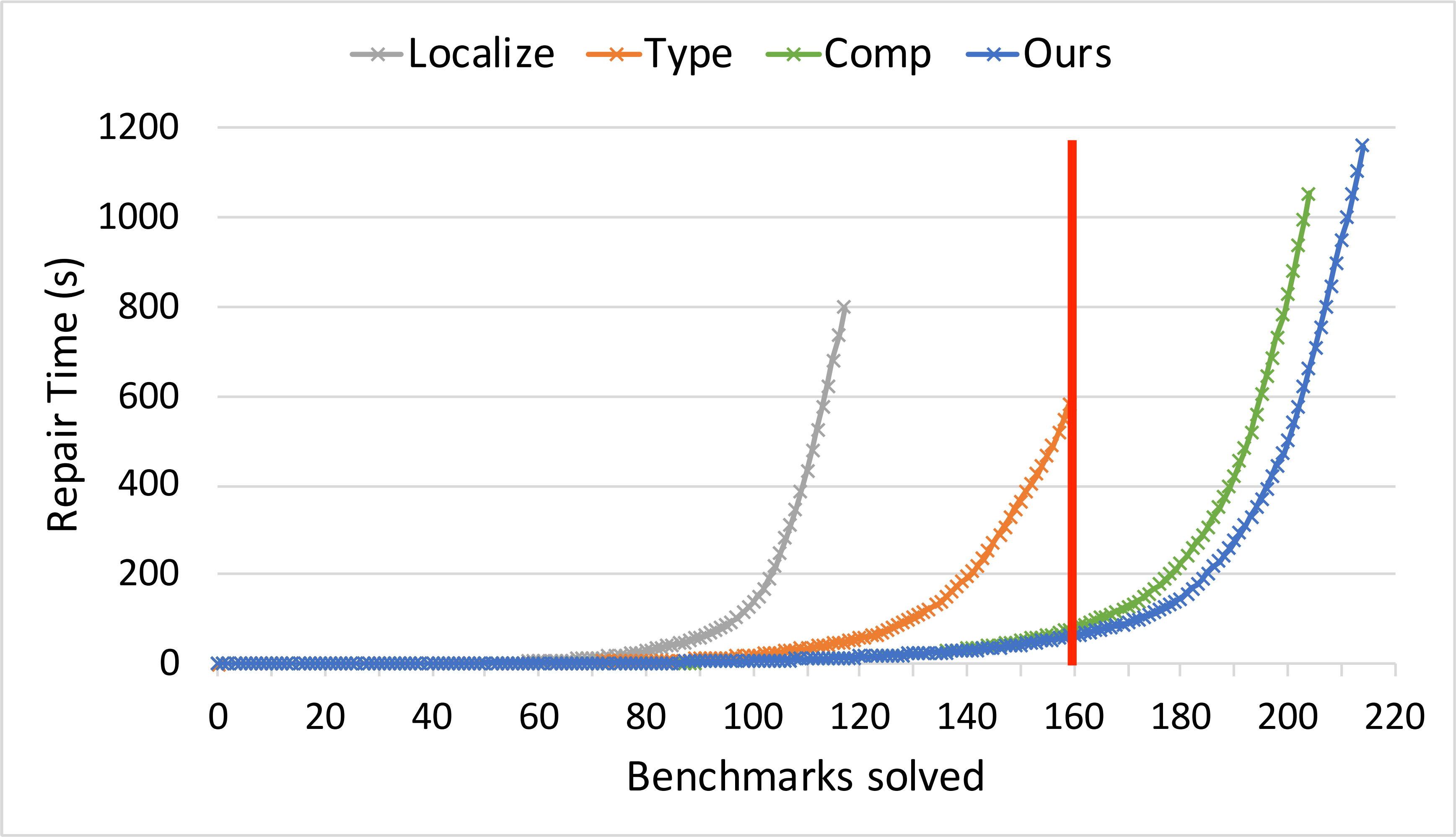
Technique Utility



- Compare to Type : 579sec vs 65sec (x 8.9 faster)

  160 vs 214 (54 submissions more)

Technique Utility



- Compare to Type : 579sec vs 65sec (x 8.9 faster)
  160 vs 214 (54 submissions more)

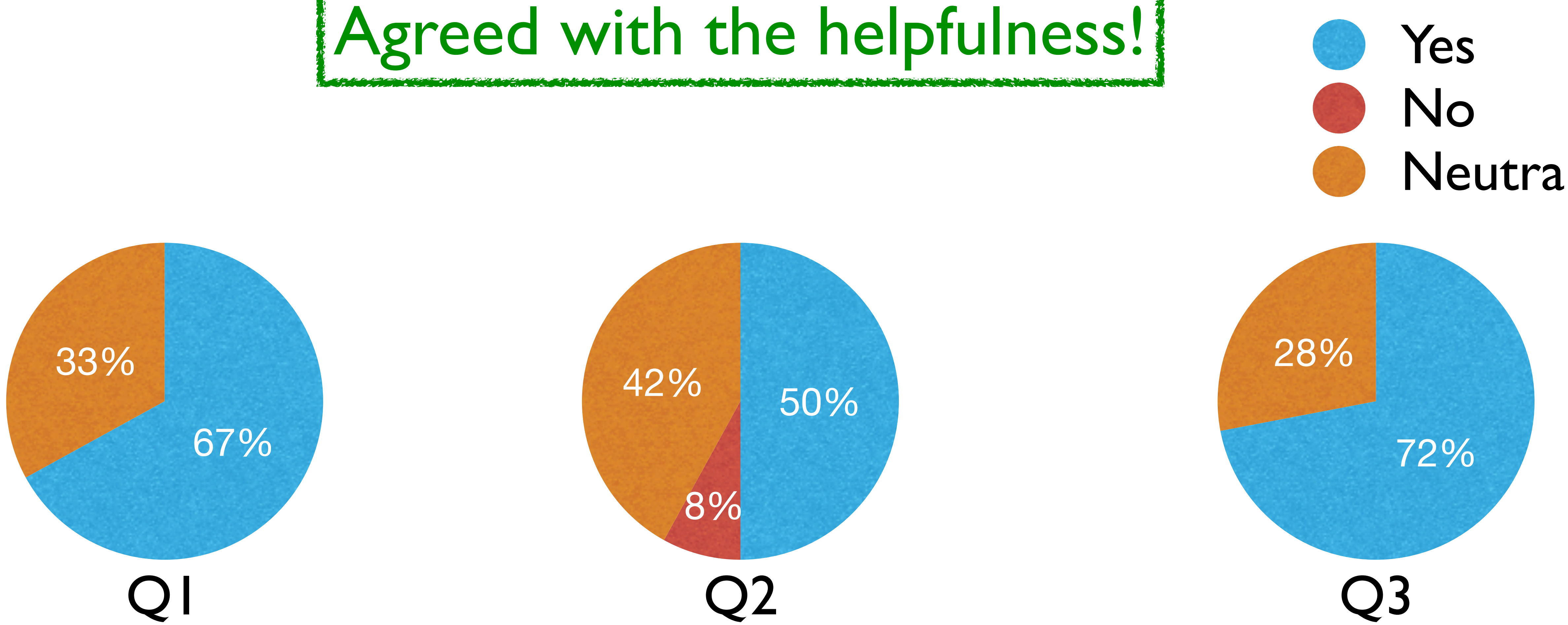
Helpfulness

Q1. Does the tool generate better corrections?

Q2. Does the feedback help to understand your mistakes?

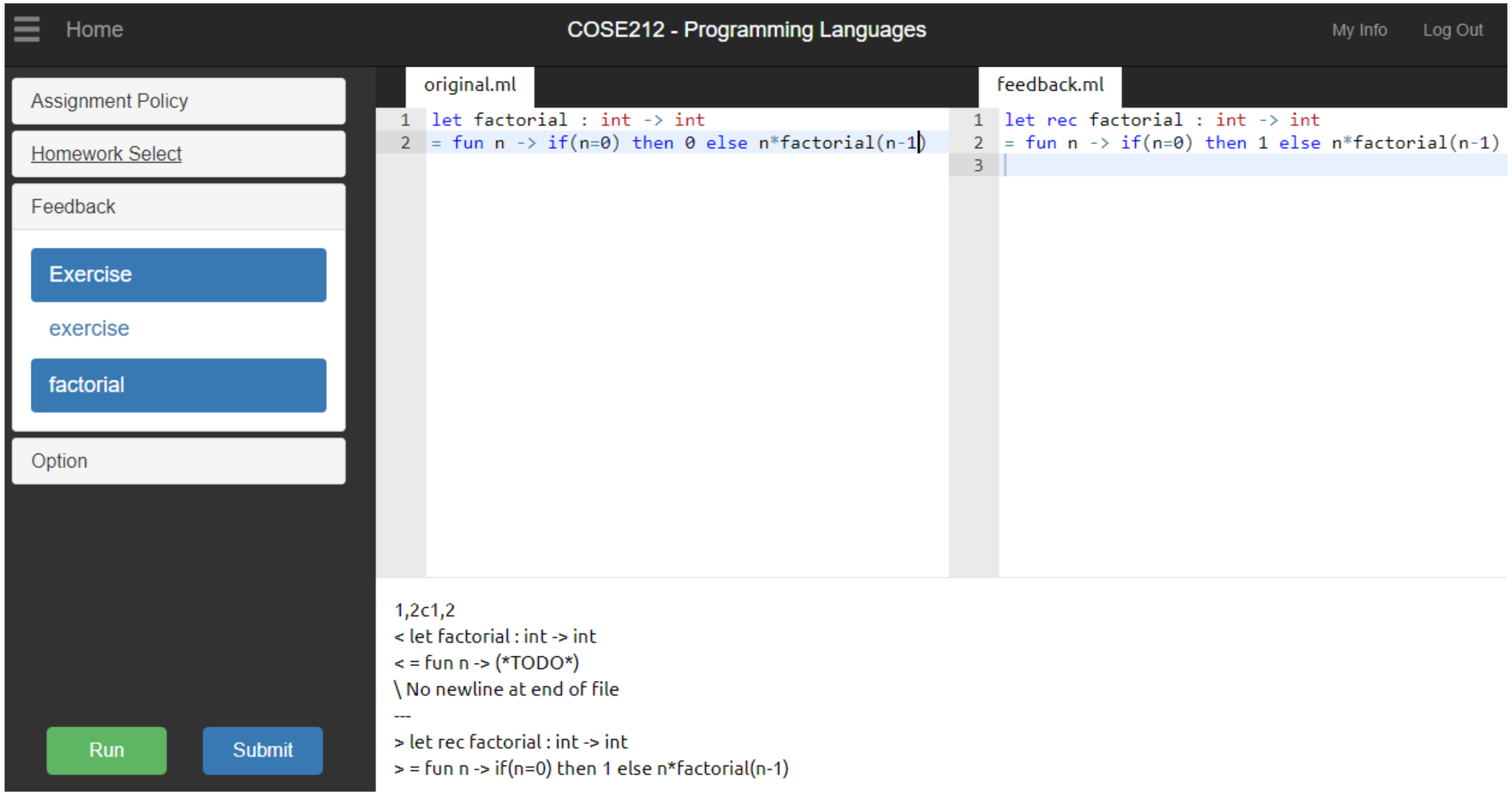
Q3. Is the tool overall useful in learning functional programming?

Agreed with the helpfulness!

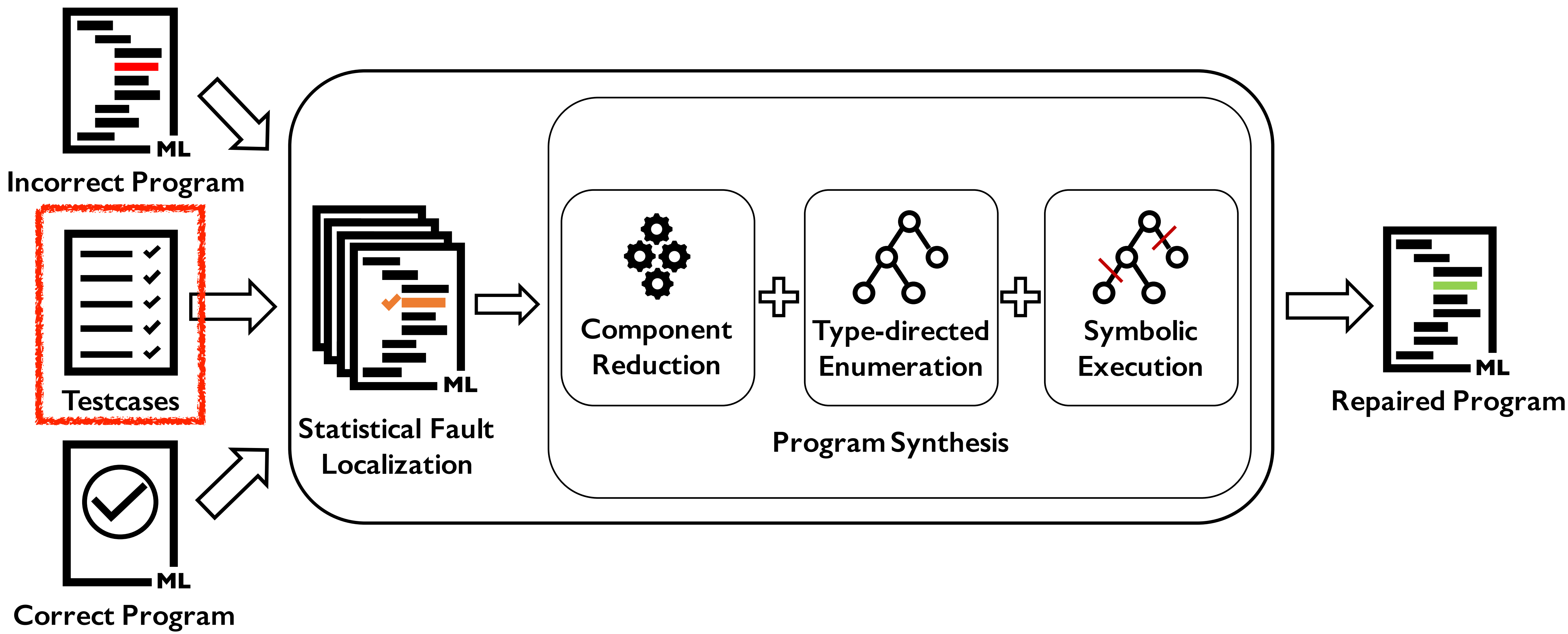


Summary

- The first system to provide personalized feedback of logical errors for functional programming assignments
- Code and our data: https://github.com/kupl/FixML
- Tool usage: https://tryml.korea.ac.kr



Limitation of FixML



- To check the correctness of given programs, FixML still requires test cases that are manually designed.

Automatic and Scalable Detection of Logical Errors in Functional Programming Assignments

Dowon Song, Myungho Lee, and Hakjoo Oh
Korea University

October 2019
OOPSLA`19 @ Athens, Greece

Motivation

- Detecting logical error is challenging and involves a lot of human effort.
  - In a real classroom, there are too many submissions to investigate one by one.
  - Manual test cases sometimes fail to detect corner-case error.

Motivation

- Detecting logical error is challenging and involves a lot of human effort.
  - In a real classroom, there are too many submissions to investigate one by one.
  - Manual test cases sometimes fail to detect corner-case error.
- Prior property-based testing also has limitations.
  - It requires for user to design proper test generator and shrinker manually.
  - Generator basically performs random testing, which makes it hard to detect program-specific errors.

Motivating Example: Composing Function

- Applying a function 'f' to 'x' 'n' times : $\mathtt{iter}(n, f)\ x = (\underbrace{f \circ \cdots \circ f}_{n})(x)$
  - For example, `(iter (5, fun x -> 1 + x) 2)` evaluates to 7.

```
let rec iter : int * (int -> int) -> int -> int
= fun (n, f) x ->
  if (n < 0) then raise (Failure "Invalid Input")
  else if (n = 0) then x
  else f (iter (n-1, f) x)
```

Correct Program

```
let rec iter : int * (int -> int) -> int -> int
= fun (n, f) x ->
  let y = (f x) in
  if (n <= 0) then x else iter (n-1, f) y
```

Buggy Program

Motivating Example: Composing Function

- Applying a function 'f' to 'x' 'n' times : $\mathtt{iter}(n, f)\ x = (\underbrace{f \circ \cdots \circ f}_{n})(x)$
  - For example, `(iter (5, fun x -> 1 + x) 2)` evaluates to 7.

```
let rec iter : int * (int -> int) -> int -> int
= fun (n, f) x ->
  if (n < 0) then raise (Failure “Invalid Input”)
  else if (n = 0) then x
  else f (iter (n-1, f) x)
```

Correct Program

```
let rec iter : int * (int -> int) -> int -> int
= fun (n, f) x ->
  let y = (f x) in
  if (n <= 0) then x else iter (n-1, f) y
```

Buggy Program

Motivating Example: Composing Function

- Applying a function 'f' to 'x' 'n' times : $\mathtt{iter}(n, f)\ x = (\underbrace{f \circ \cdots \circ f}_{n})(x)$
  - For example, `(iter (5, fun x -> 1 + x) 2)` evaluates to 7.
  - Counter-example : `(n, f) = (0, fun x -> 1 mod x)` and `x = 0`

```
let rec iter : int * (int -> int) -> int -> int
= fun (n, f) x ->
  if (n < 0) then raise (Failure "Invalid Input")
  else if (n = 0) then x
  else f (iter (n-1, f) x)
```

Correct Program

```
let rec iter : int * (int -> int) -> int -> int
= fun (n, f) x ->
  let y = (f x) in
  if (n <= 0) then x else iter (n-1, f) y
```

Buggy Program

Motivating Example: Composing Function

- Applying a function ‘f’ to ‘x’ ‘n’ times : $\mathtt{iter}(n, f)\ x = (\underbrace{f \circ \cdots \circ f}_{n})(x)$
  - For example, `(iter (5, fun x -> 1 + x) 2)` evaluates to 7.
  - Counter-example : `(n, f) = (0, fun x -> 1 mod x)` and `x = 0`

```
let rec iter : int * (int -> int) -> int -> int
= fun (n, f) x ->
  if (n < 0) then raise (Failure “Invalid Input”)
  else if (n = 0) then x
  else f (iter (n-1, f) x)
```

Correct Program

Return 0 as an output

```
let rec iter : int * (int -> int) -> int -> int
= fun (n, f) x ->
  let y = (f x) in
  if (n <= 0) then x else iter (n-1, f) y
```

Buggy Program

Division-by-zero

Motivating Example: Composing Function

- Applying a function ‘f’ to ‘x’ ‘n’ times : $\mathtt{iter}(n, f)\ x = (\underbrace{f \circ \cdots \circ f}_{n})(x)$
  - For example, `(iter (5, fun x -> 1 + x) 2)` evaluates to 7.
  - Counter-example : `(n, f) = (0, fun x -> 1 mod x)` and `x = 0`

Automatically generate counter-example for each submission!

```
let r
= fun (n, f) x ->
  if (n < 0) then raise (Failure “Invalid Input”)
  else if (n = 0) then x
  else f (iter (n-1, f) x)
```

Correct Program

Return 0 as an output

```
let rec iter : int * (int -> int) -> int -> int
= fun (n, f) x ->
  let y = (f x) in
  if (n <= 0) then x else iter (n-1, f) y
```

Buggy Program

Division-by-zero

Running Example: List map

- Applying a function to all elements of given integer list

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

Baseline1: Enumerative Search

- Enumerate all possible test cases from the smallest one until we find one causing different outputs.

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

f: `fun x -> ?`

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

Baseline1: Enumerative Search

- Enumerate all possible test cases from the smallest one until we find one causing different outputs.

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

```
f: fun x -> ?
      …
      fun x -> x
      fun x -> ?+?
      fun x -> ?-?
      …
```

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

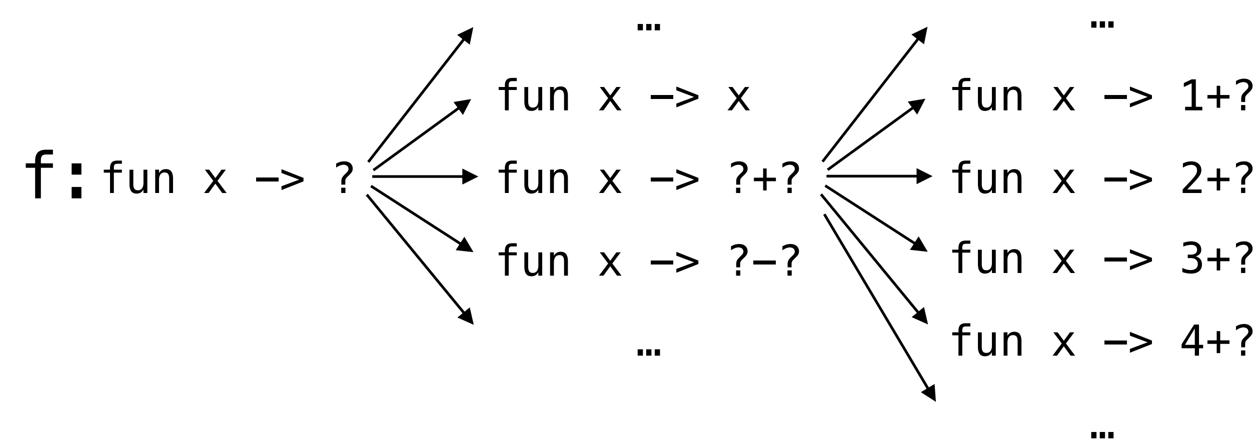
Buggy Program

Baseline1: Enumerative Search

- Enumerate all possible test cases from the smallest one until we find one causing different outputs.

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program




```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

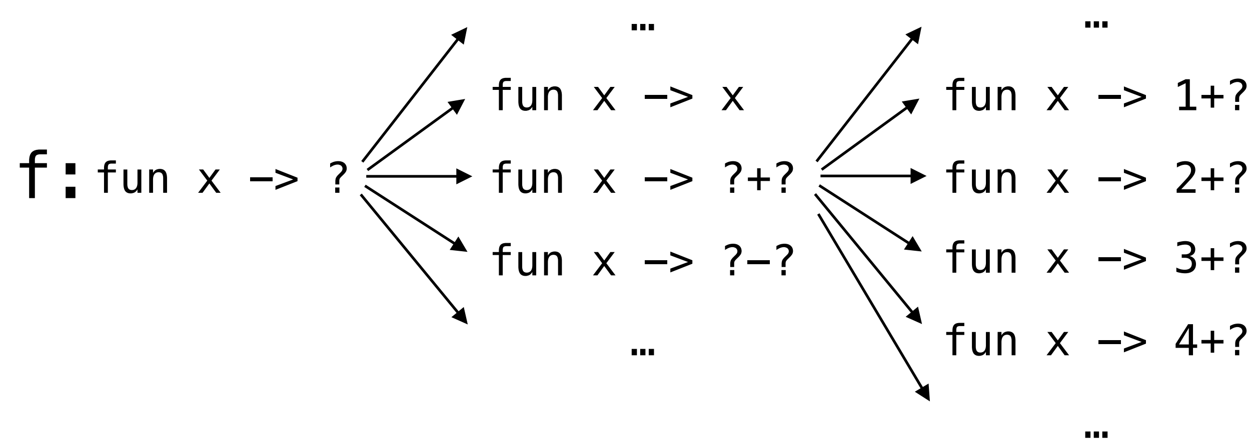
Buggy Program

Baseline1: Enumerative Search

- Enumerate all possible test cases from the smallest one until we find one causing different outputs.

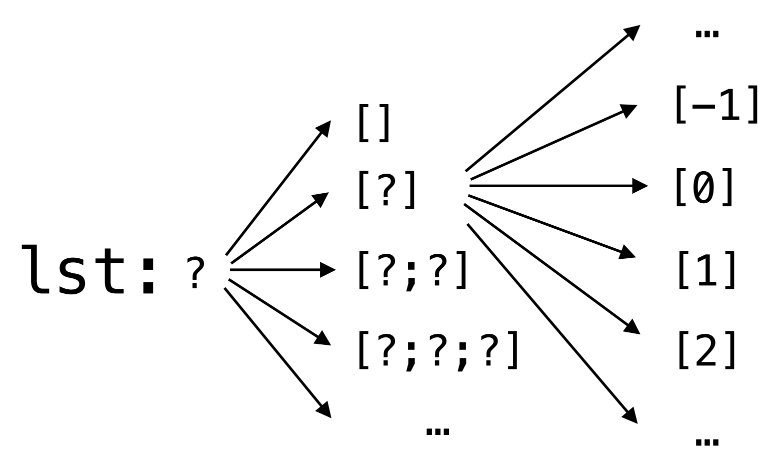
```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program




```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

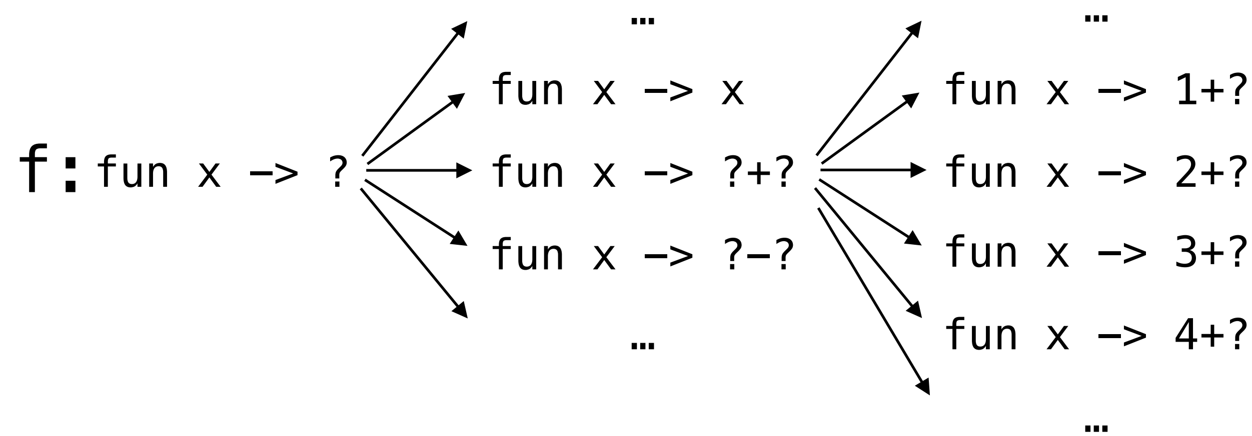


Baseline1: Enumerative Search

- Enumerate all possible test cases from the smallest one until we find one causing different outputs.

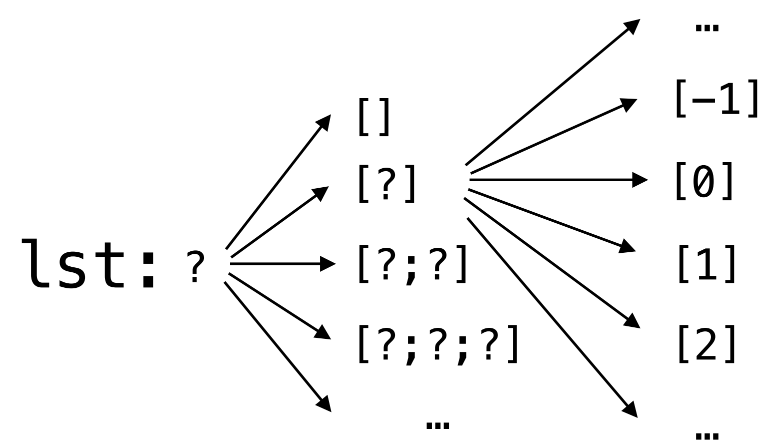
```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program




```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program



Inefficient to search infinite values!

Baseline2: Symbolic Execution

- Systematically compare two programs by executing them symbolically.

$\texttt{f}=\alpha_f \qquad \texttt{lst}=\alpha_{lst}$

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

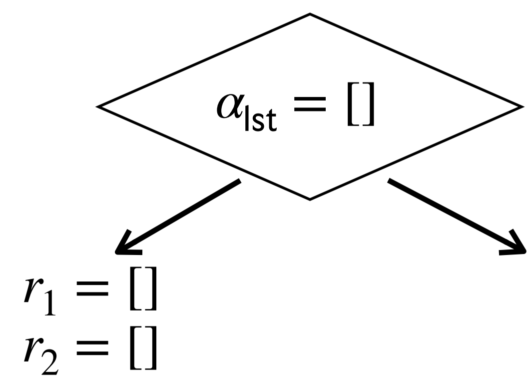
Baseline2: Symbolic Execution

- Systematically compare two programs by executing them symbolically.

$\texttt{f}=\alpha_{\mathsf{f}}$ $\texttt{lst}=\alpha_{\mathsf{lst}}$

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program




```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

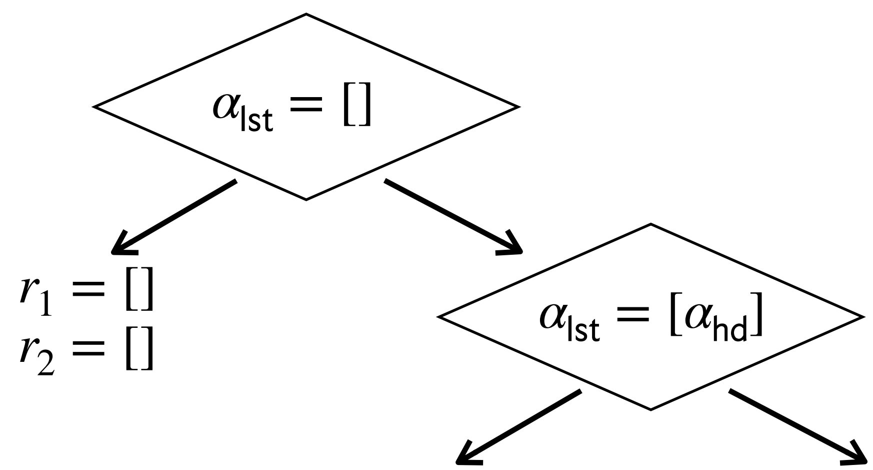
Baseline2: Symbolic Execution

- Systematically compare two programs by executing them symbolically.

`f=`α_f `lst=`α_{lst}

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program




```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

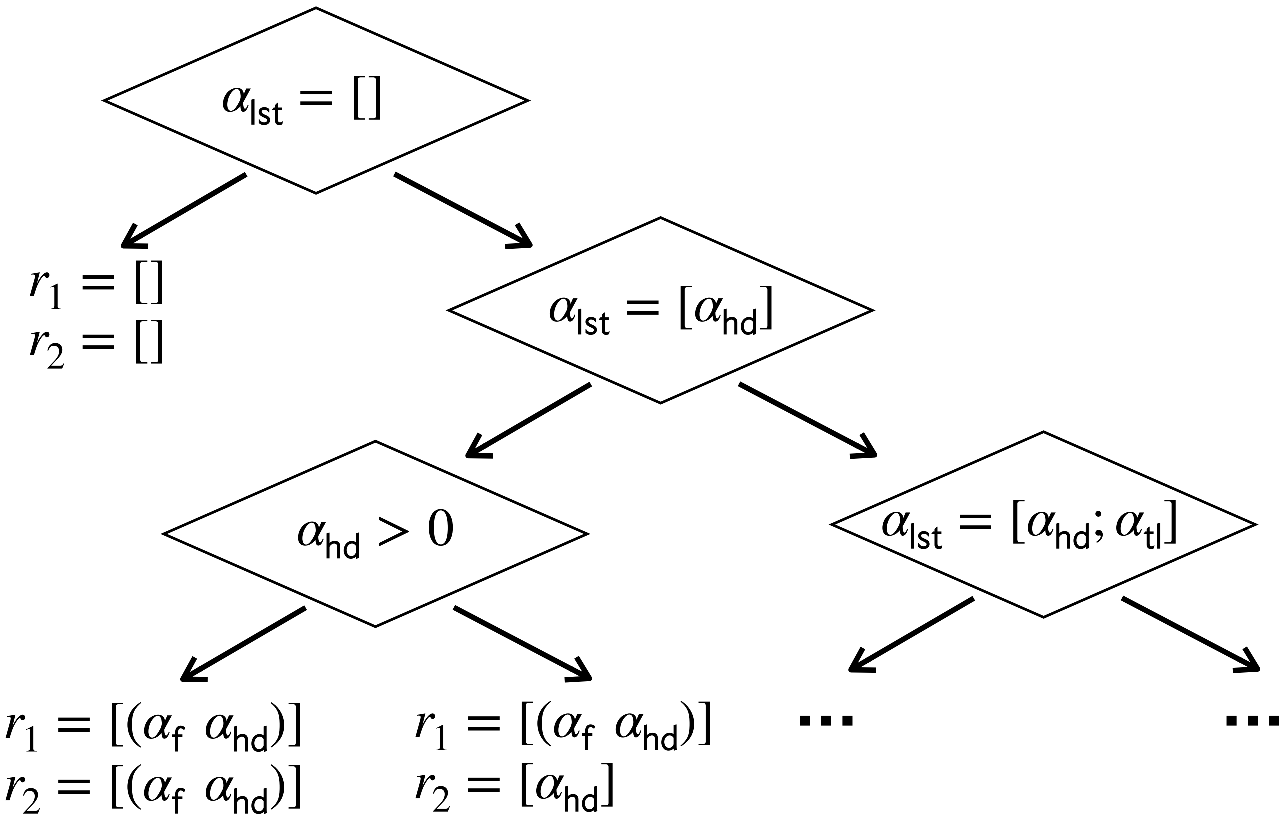
Baseline2: Symbolic Execution

- Systematically compare two programs by executing them symbolically.

$\texttt{f}=\alpha_{\mathsf{f}}$ $\texttt{lst}=\alpha_{\mathsf{lst}}$

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program




```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

1. Path explosion

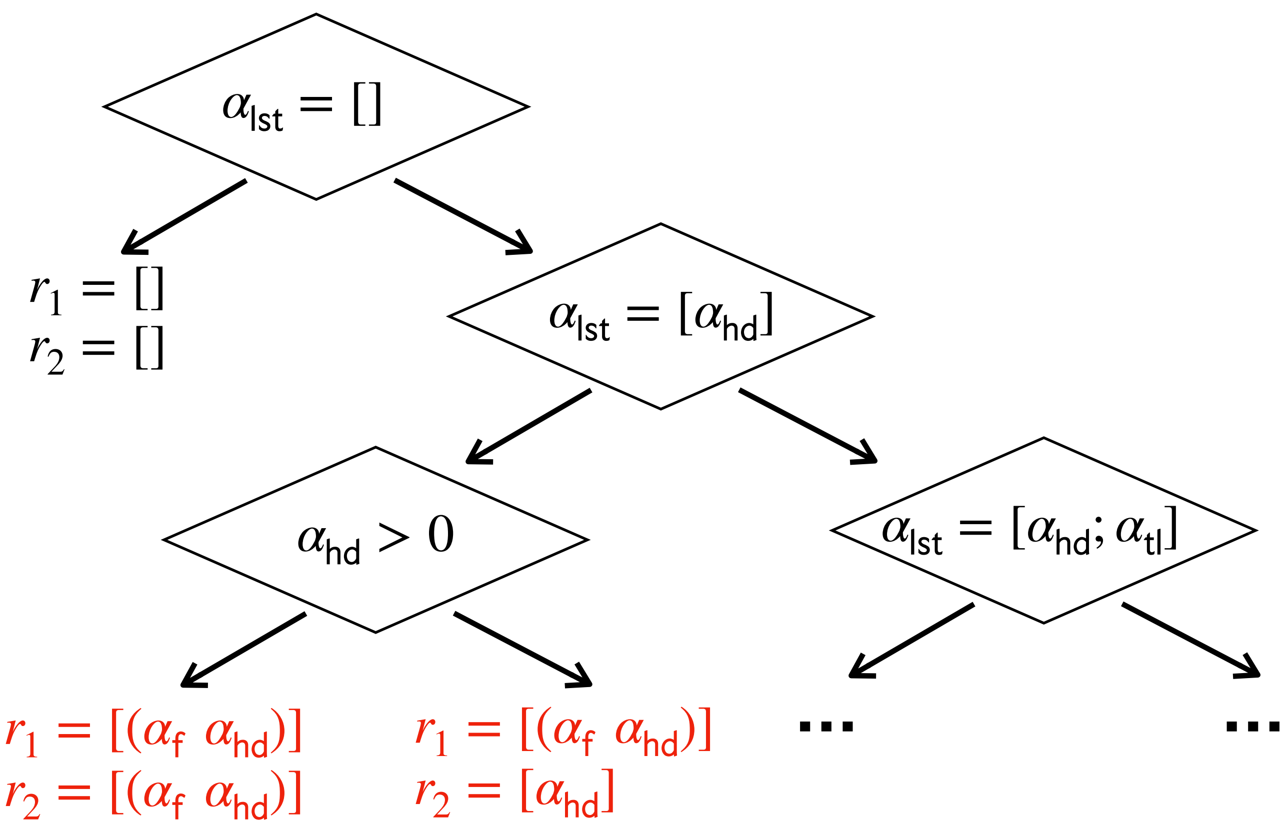
Baseline2: Symbolic Execution

- Systematically compare two programs by executing them symbolically.

`f=`α_f `lst=`α_{lst}

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program




```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

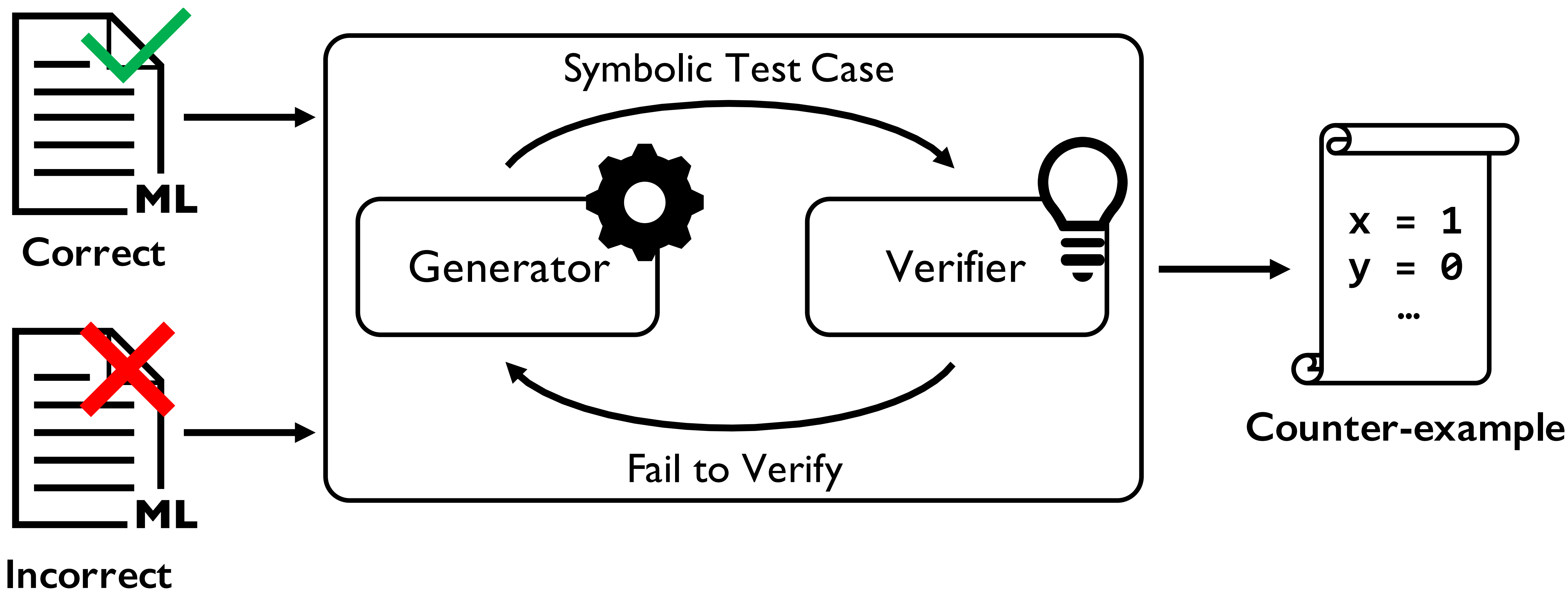
1. Path explosion
2. Hard to handle non-primitive symbols

Key Idea

- Combine enumerative search and symbolic execution to overcome the key limitations of each other.
  - Enumerative search
    - Effectively generate small code snippet such as non-primitive values (e.g. function type value)
    - Hard to enumerate infinite number of primitive values
  - Symbolic execution
    - Easy to deduce specific primitive values using constraint solving
    - Heavy to apply to non-primitive values

Our Approach

- Given a reference program and a buggy program, generate a counter-example without any human effort.

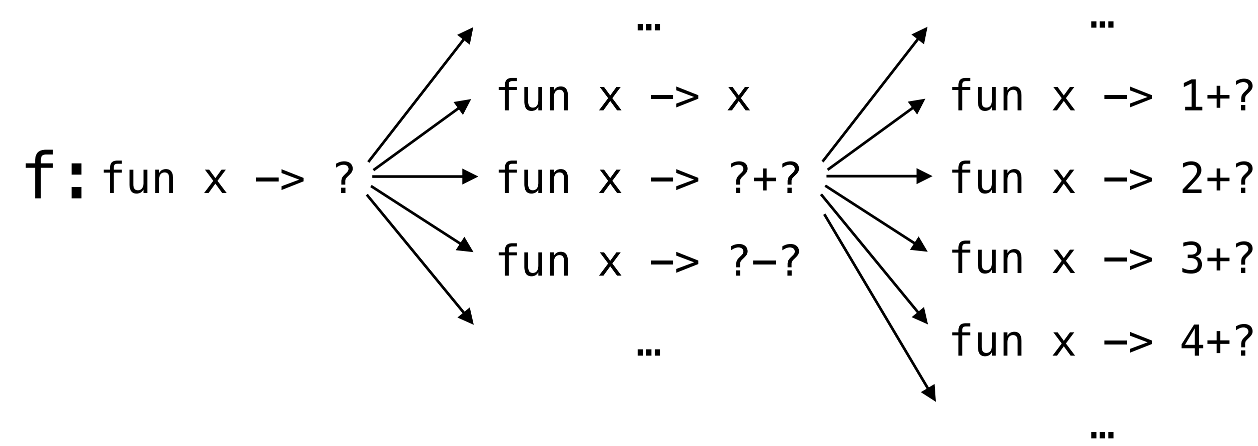


Symbolic Test Case Generation

- Instead of generating concrete ones, synthesizing symbolic test cases by representing primitive values as symbols.

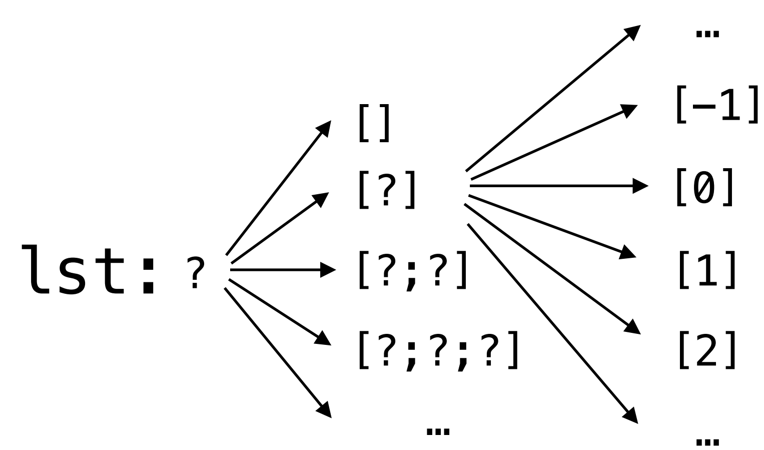
```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program




```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

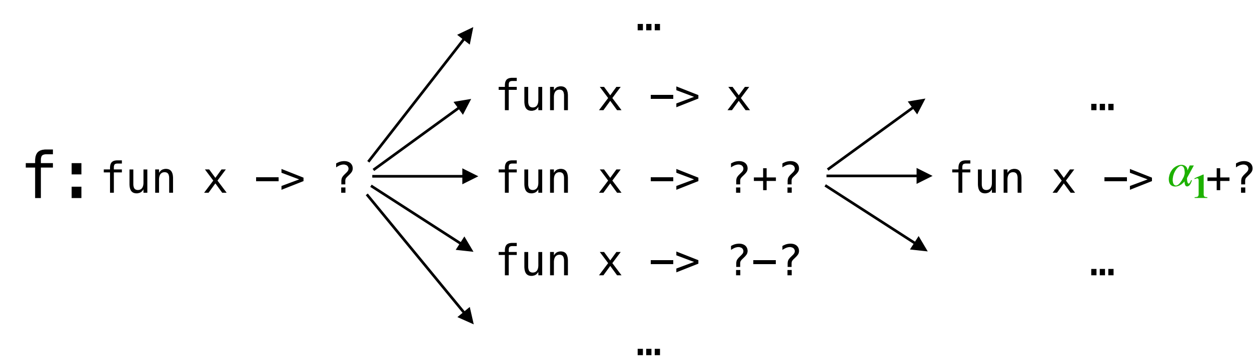


Symbolic Test Case Generation

- Instead of generating concrete ones, synthesizing symbolic test cases by representing primitive values as symbols.

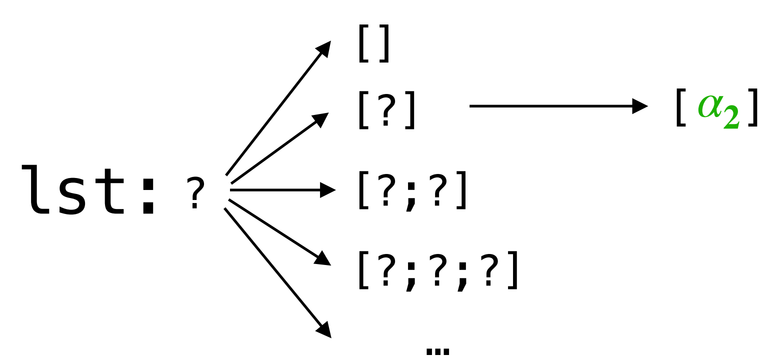
```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program




```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program



Reduce the search space

Bounded Symbolic Execution

- Compute a set of all possible outputs and paths by running two programs with symbolic test cases.

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

Symbolic test cases:

- f = $(\text{fun x} -> x + \alpha_1)$
- lst = $[\alpha_2]$

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

Bounded Symbolic Execution

- Compute a set of all possible outputs and paths by running two programs with symbolic test cases.

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

Symbolic test cases:

- f = $(\text{fun x} -> \text{x} + \alpha_1)$
- lst = $[\alpha_2]$

Symbolic execution result:

- Correct : $\Phi_c = \{(\text{true}, [\alpha_2 + \alpha_1])\}$
- Buggy : $\Phi_b = \{(\alpha_2 > 0, [\alpha_2 + \alpha_1]), (\alpha_2 \leq 0, [\alpha_2])\}$

Validation

- Automatically infer specific values by solving the resulting verification condition.

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

Symbolic test cases:

- $\mathsf{f} = (\mathsf{fun}\ \mathsf{x} -> \mathsf{x} + \alpha_1)$
- $\mathsf{lst} = [\alpha_2]$

Symbolic execution result:

- Correct : $\Phi_c = \{(\text{true}, [\alpha_2 + \alpha_1])\}$
- Buggy : $\Phi_b = \{(\alpha_2 > 0, [\alpha_2 + \alpha_1]), (\alpha_2 \leq 0, [\alpha_2])\}$

Verification condition:

$$\bigwedge_{(\pi_c, v_c) \in \Phi_c} \pi_c \implies \bigvee_{(\pi_b, v_b) \in \Phi_b} \pi_b \wedge v_c = v_b$$

Validation

- Automatically infer specific values by solving the resulting verification condition.

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

Symbolic test cases:

- f = (fun x $-$ > x + α_1)
- lst = $[\alpha_2]$

Symbolic execution result:

- Correct : $\Phi_c = \{(\text{true}, [\alpha_2 + \alpha_1])\}$
- Buggy : $\Phi_b = \{(\alpha_2 > 0, [\alpha_2 + \alpha_1]), (\alpha_2 \leq 0, [\alpha_2])\}$

Verification condition:

$$(\text{true} \implies (\alpha_2 > 0 \wedge [\alpha_2 + \alpha_1] = [\alpha_2 + \alpha_1]) \vee (\alpha_2 \leq 0 \wedge [\alpha_2 + \alpha_1] = [\alpha_2]))$$

Validation

- Automatically infer specific values by solving the resulting verification condition.

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

Symbolic test cases:

- f = (fun x − > x + α_1)
- lst = $[\alpha_2]$

Symbolic execution result:

- Correct : $\Phi_c = \{(\text{true}, [\alpha_2 + \alpha_1])\}$
- Buggy : $\Phi_b = \{(\alpha_2 > 0, [\alpha_2 + \alpha_1]), (\alpha_2 \leq 0, [\alpha_2])\}$

Verification condition:

$$(\text{true} \implies (\alpha_2 > 0 \wedge [\alpha_2 + \alpha_1] = [\alpha_2 + \alpha_1]) \vee (\alpha_2 \leq 0 \wedge [\alpha_2 + \alpha_1] = [\alpha_2]))$$

Validation

- Automatically infer specific values by solving the resulting verification condition.

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

Correct Program

```
let rec map : (int -> int) -> int list -> int list
= fun f lst ->
  match lst with
  | [] -> []
  | hd::tl -> if hd > 0 then (f hd)::(map f tl)
              else hd::(map f tl)
```

Buggy Program

Symbolic test cases:

- $\text{f} = (\text{fun x} -> \text{x} + \alpha_1)$
- $\text{lst} = [\alpha_2]$

Symbolic execution result:

- Correct : $\Phi_c = \{(\text{true}, [\alpha_2 + \alpha_1])\}$
- Buggy : $\Phi_b = \{(\alpha_2 > 0, [\alpha_2 + \alpha_1]), (\alpha_2 \leq 0, [\alpha_2])\}$

Verification condition:

$$(\text{true} \implies (\alpha_2 > 0 \wedge [\alpha_2 + \alpha_1] = [\alpha_2 + \alpha_1]) \vee (\alpha_2 \leq 0 \wedge [\alpha_2 + \alpha_1] = [\alpha_2]))$$

When $\alpha_1 = 1 \wedge \alpha_2 = 0$, the VC is false.

Counter Example :

- $\text{f} = (\text{fun x} -> \text{x} + 1)$
- $\text{lst} = [0]$

Evaluation

- Implemented our approach in a tool, TestML.
- Evaluated it on 4,060 submissions from 10 problems used in our functional programming course.
- Research questions:
    - How effectively does TestML detect erroneous submissions than manual test cases?
    - Is TestML more effective than property-based testing?
    - Can TestML enhance automatic program repair system?

Effectiveness

No	Problem Description	# Error Programs			
		TestML ✓ Manual ✓	TestML ✓ Manual ✗	TestML ✗ Manual ✓	Total
1	Finding a maximum element in a list	35	10	0	45
2	Filtering a list	5	4	0	9
3	Mirroring a binary tree	9	0	0	9
4	Checking membership in a binary tree	19	0	0	19
5	Computing $\sum_{i=j}^{k} f(i)$ for j, k, and f	32	0	0	32
6	Composing functions	46	3	0	49
7	Adding numbers in user-defined number system	14	4	0	18
8	Evaluating expressions and propositional formulas	105	7	0	112
9	Deciding lambda terms are well-formed or not	116	25	0	141
10	Differentiating algebraic expressions	162	35	0	197
	Total	543	88	0	631

- For comparison, we used 10 manual test cases which have been continually refined.
- TestML found 88 more errors than human-provided test cases.

Comparison with property-based testing

No	Problem Description	QCheck1		QCheck2		TestML	
		#E	Time	# E	Time	#E	Time
1	Finding a maximum element in a list	45	86.0	38	72.6	45	0.5
3	Mirroring a binary tree	9	0.0	9	0.0	9	0.3
4	Checking membership in a binary tree	19	0.0	19	0.0	19	0.5
7	Adding numbers in user-defined number system	18	0.8	18	0.8	18	0.3
8	Evaluating expressions and propositional formulas	112	3.7	112	10.5	112	6.5
9	Deciding lambda terms are well-formed or not	139	110.4	130	555.8	141	10.4
10	Differentiating algebraic expressions	186	390.1	182	318.6	197	86.6
	Total	528	592.0	508	958.4	541	105.1

- Used QCheck, a property-based testing tool for OCaml.
- Manually designed well-tuned test generator and shrinker for QCheck.
- TestML outperforms QCheck without any human effort.

Test-suite Overfitted Patch

- Test-case-based program repair sometimes produces test-suite overfitted patches which satisfy only given test cases.

```
let rec eval_exp e =
  match e with
  | Num n -> n
  | Add (e1, e2) -> (eval_exp e1) + (eval_exp e2)
  | Sub (e1, e2) -> (eval_exp e1) + (eval_exp e2)

let rec eval f =
  match f with
  | True -> true
  | False -> false
  | Not f -> not (eval f)
  | AndAlso (f1, f2) -> (eval f1) && (eval f2)
  | OrElse (f1, f2) -> (eval f1) || (eval f2)
  | Imply (f1, f2) -> not (eval f1) || (eval f2)
  | Less (e1, e2) -> (eval_exp e1) < (eval_exp e2)
```

Buggy Program

Input	Output
Less (Num 1, Num 2)	true
Less (Sub (Num 1, Num 2), Num 4)	true
Less (Add (Num 1, Num 3), Sub (Num 2, Num 3))	false

Test Cases

Test-suite Overfitted Patch

- Test-case-based program repair sometimes produces test-suite overfitted patches which satisfy only given test cases.

```
let rec eval_exp e =
  match e with
  | Num n -> n
  | Add (e1, e2) -> (eval_exp e1) + (eval_exp e2)
  | Sub (e1, e2) -> (eval_exp e1) + (eval_exp e2)

let rec eval f =
  match f with
  | True -> true
  | False -> false
  | Not f -> not (eval f)
  | AndAlso (f1, f2) -> (eval f1) && (eval f2)
  | OrElse (f1, f2) -> (eval f1) || (eval f2)
  | Imply (f1, f2) -> not (eval f1) || (eval f2)
  | Less (e1, e2) -> (eval_exp e1) < (eval_exp e2)
```

Buggy Program

Input	Output
`Less (Num 1, Num 2)`	`true`
`Less (Sub (Num 1, Num 2), Num 4)`	`true`
`Less (Add (Num 1, Num 3), Sub (Num 2, Num 3))`	`false`

`1+3 < 2+3 => true`

Test Cases

Test-suite Overfitted Patch

- Test-case-based program repair sometimes produces test-suite overfitted patches which satisfy only given test cases.

```
let rec eval_exp e =
  match e with
  | Num n -> n
  | Add (e1, e2) -> (eval_exp e1) + (eval_exp e2)
  | Sub (e1, e2) -> (eval_exp e1) + (eval_exp e2)

let rec eval f =
  match f with
  | True -> true
  | False -> false
  | Not f -> not (eval f)
  | AndAlso (f1, f2) -> (eval f1) && (eval f2)
  | OrElse (f1, f2) -> (eval f1) || (eval f2)
  | Imply (f1, f2) -> not (eval f1) || (eval f2)
  | Less (e1, e2) -> (eval_exp e1) < (eval_exp e2)
```

Buggy Program

```
let rec eval_exp e =
  match e with
  | Num n -> n
  | Add (e1, e2) -> (eval_exp e2) + (eval_exp e2)
  | Sub (e1, e2) -> (eval_exp e1) + (eval_exp e2)

let rec eval f =
  match f with
  | True -> true
  | False -> false
  | Not f -> not (eval f)
  | AndAlso (f1, f2) -> (eval f1) && (eval f2)
  | OrElse (f1, f2) -> (eval f1) || (eval f2)
  | Imply (f1, f2) -> not (eval f1) || (eval f2)
  | Less (e1, e2) -> (eval_exp e1) < (eval_exp e2)
```

Overfitted Patch

Input	Output
`Less (Num 1, Num 2)`	`true`
`Less (Sub (Num 1, Num 2), Num 4)`	`true`
`Less (Add (Num 1, Num 3), Sub (Num 2, Num 3))`	`false`

Test Cases

Test-suite Overfitted Patch

- Test-case-based program repair sometimes produces test-suite overfitted patches which satisfy only given test cases.

```
let rec eval_exp e =
  match e with
  | Num n -> n
  | Add (e1, e2) -> (eval_exp e1) + (eval_exp e2)
  | Sub (e1, e2) -> (eval_exp e1) + (eval_exp e2)

let rec eval f =
  match f with
  | True -> true
  | False -> false
  | Not f -> not (eval f)
  | AndAlso (f1, f2) -> (eval f1) && (eval f2)
  | OrElse (f1, f2) -> (eval f1) || (eval f2)
  | Imply (f1, f2) -> not (eval f1) || (eval f2)
  | Less (e1, e2) -> (eval_exp e1) < (eval_exp e2)
```

Buggy Program

```
let rec eval_exp e =
  match e with
  | Num n -> n
  | Add (e1, e2) -> (eval_exp e2) + (eval_exp e2)
  | Sub (e1, e2) -> (eval_exp e1) + (eval_exp e2)

let rec eval f =
  match f with
  | True -> true
  | False -> false
  | Not f -> not (eval f)
  | AndAlso (f1, f2) -> (eval f1) && (eval f2)
  | OrElse (f1, f2) -> (eval f1) || (eval f2)
  | Imply (f1, f2) -> not (eval f1) || (eval f2)
  | Less (e1, e2) -> (eval_exp e1) < (eval_exp e2)
```

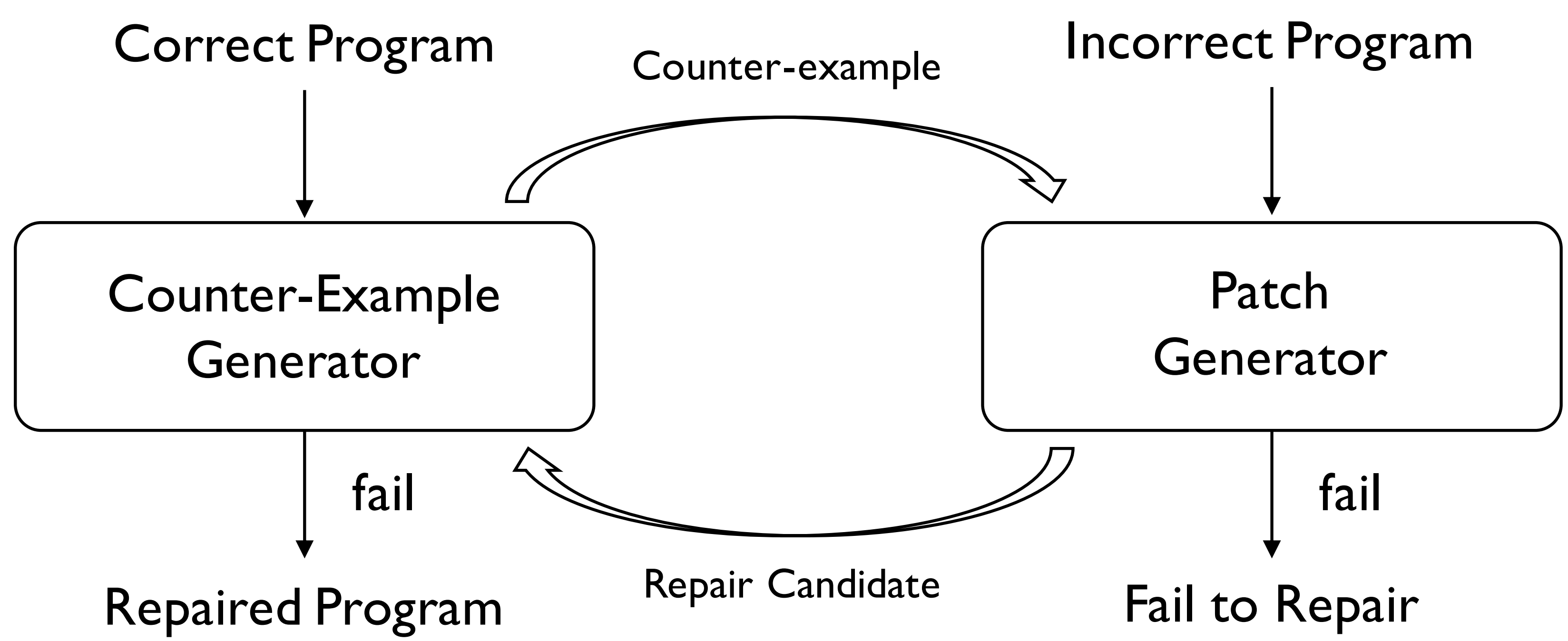
Overfitted Patch

Input	Output
Less (Num 1, Num 2)	true
Less (Sub (Num 1, Num 2), Num 4)	true
Less (Add (Num 1, Num 3), Sub (Num 2, Num 3))	false

Test Cases

3+3 < 2+3 => false

Counter-example Guided Repair



- Verify the correctness of generated patch by generating counter-example.

- Supplement the given test suite with newly found counter examples, and try to fix the error again.

Usefulness in Automatic Program Repair

No	Problem Description	Manual Test Suite				Our Technique			
		#E	#P	#O	Rate	#E	#P	#O	Rate
1	Finding a maximum element in a list	35	32	0	90%	45	42	0	93%
2	Filtering a list	5	3	0	60%	9	6	0	67%
3	Mirroring a binary tree	9	7	1	78%	9	8	0	89%
4	Checking membership in a binary tree	19	11	1	58%	19	12	0	63%
5	Computing $\sum_{i=j}^{k} f(i)$ for j, k, and f	32	11	6	34%	32	16	1	50%
6	Composing functions	46	17	0	37%	49	20	0	41%
7	Adding numbers in user-defined number system	14	4	2	29%	18	9	0	50%
8	Evaluating expressions and propositional formulas	105	29	12	28%	112	45	0	40%
9	Deciding lambda terms are well-formed or not	116	16	29	14%	141	33	0	23%
10	Differentiating algebraic expressions	162	26	7	16%	197	46	0	23%
	Total/Average	543	156	58	29%	631	237	1	38%

- Applied our counter-example generation algorithm to FixML.
- Significantly reduce the number of test-suite overfitted patches (58 to 1).
- The patch rate eventually increased (from 29% to 38%).

Summary

- We proposed a novel technique for detecting logical errors in functional programming assignments without any human effort.
  - Combining enumerative search and symbolic execution in a synergistic way
- The evaluation results show that our technique is useful for error detection and program repair.
- Code and our data: https://github.com/kupl/TestML

Summary

- We proposed a novel technique for detecting logical errors in functional programming assignments without any human effort.
  - Combining enumerative search and symbolic execution in a synergistic way
- The evaluation results show that our technique is useful for error detection and program repair.

Thank you for listening!

- Code and our data: https://github.com/kupl/TestML

Supplementary

Supplementary

Supplementary

Supplementary

Supplementary

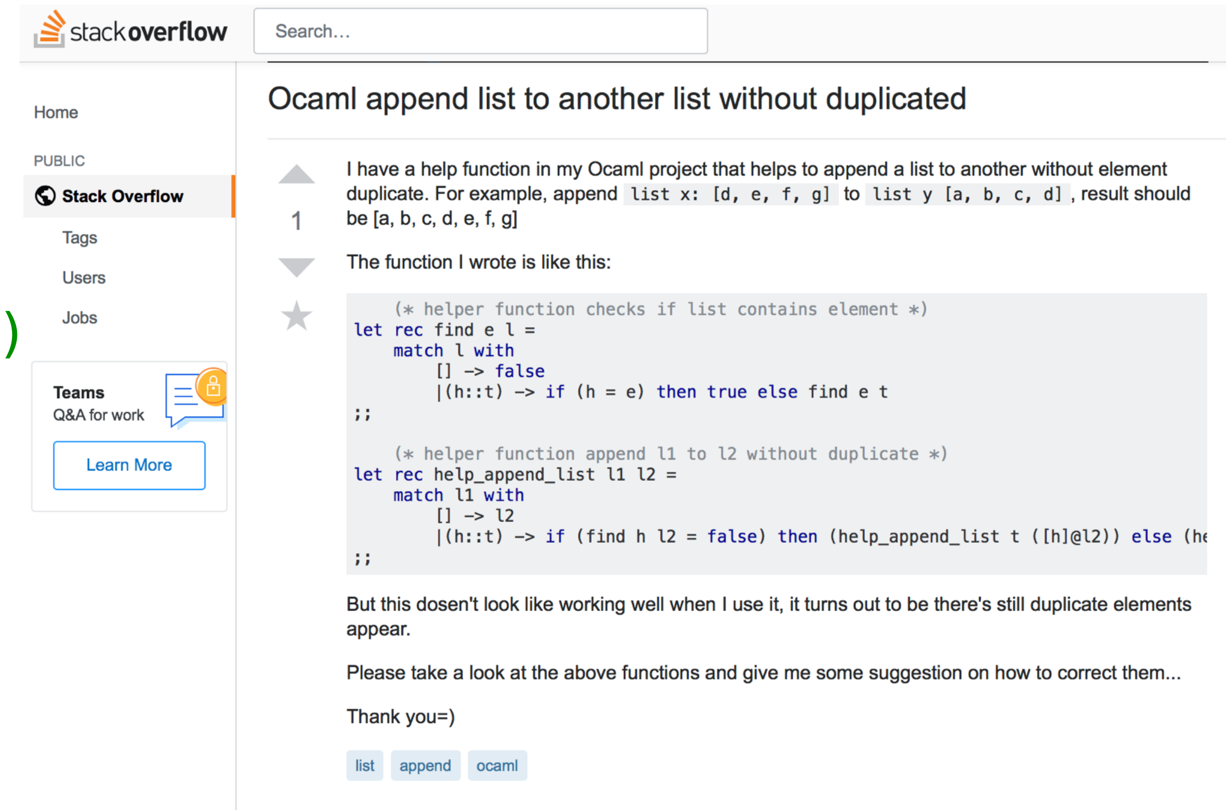
Supplementary

Example3: Append Lists

- Stackoverflow example

```
Test cases :
append_list [1;3] [3;4;5] = [3;4;5;1]
append_list [1] [3;3;4] = [3;4;1]
```

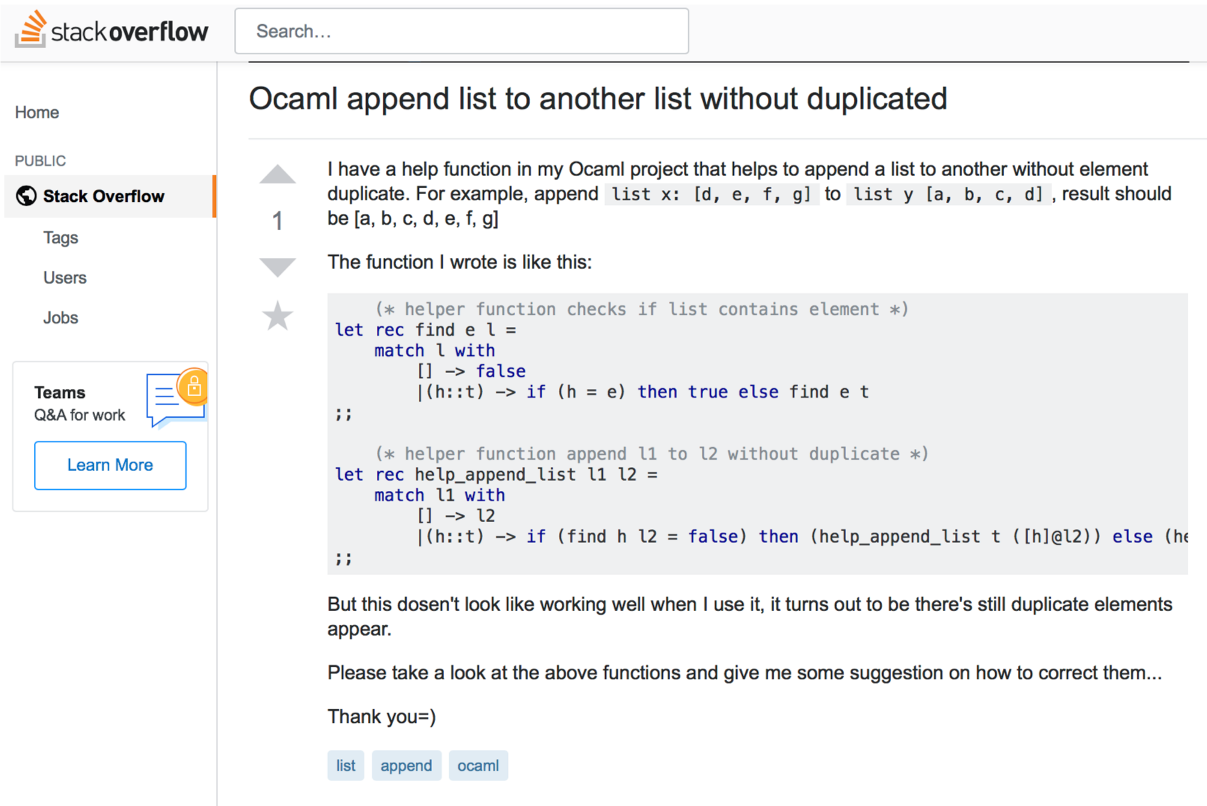
```
(* check whether the element e is in list l *)
let rec find e l =
  match l with
  | [] -> false
  | h::t -> if h = e then true else find e t

(* append l1's elements not in l2 *)
let rec helper l1 l2 =
  match l1 with
  | [] -> l2
  | h::t ->
    if find h l2 = false then helper t (l2@[h])
    else helper t l2

let append_list x y = helper x y
```



Example3: Append Lists

- Stackoverflow example

```
Test cases :
append_list [1;3] [3;4;5] = [3;4;5;1]
append_list [1] [3;3;4] = [3;4;1]
```


stackoverflow
Search...
Home
PUBLIC
Stack Overflow
Tags
Users
Jobs
Teams
Q&A for work
Learn More

Ocaml append list to another list without duplicated

1

I have a help function in my Ocaml project that helps to append a list to another without element duplicate. For example, append `list x: [d, e, f, g]` to `list y [a, b, c, d]`, result should be [a, b, c, d, e, f, g]

The function I wrote is like this:

```
    (* helper function checks if list contains element *)
let rec find e l =
    match l with
        [] -> false
        |(h::t) -> if (h = e) then true else find e t
;;

    (* helper function append l1 to l2 without duplicate *)
let rec help_append_list l1 l2 =
    match l1 with
        [] -> l2
        |(h::t) -> if (find h l2 = false) then (help_append_list t ([h]@l2)) else (he
;;
```

But this dosen't look like working well when I use it, it turns out to be there's still duplicate elements appear.

Please take a look at the above functions and give me some suggestion on how to correct them...

Thank you=)

list append ocaml

```
(* check whether the element e is in list l *)
let rec find e l =
  match l with
  | [] -> false
  | h::t -> if h = e then true else find e t

(* append l1's elements not in l2 *)
let rec helper l1 l2 =
  match l1 with
  | [] -> l2
  | h::t ->
    if find h l2 = false then helper t (l2@[h])
    else helper t l2

let append_list x y = helper x y
```

append_list [1] [3;3;4] = [3;3;4;1]

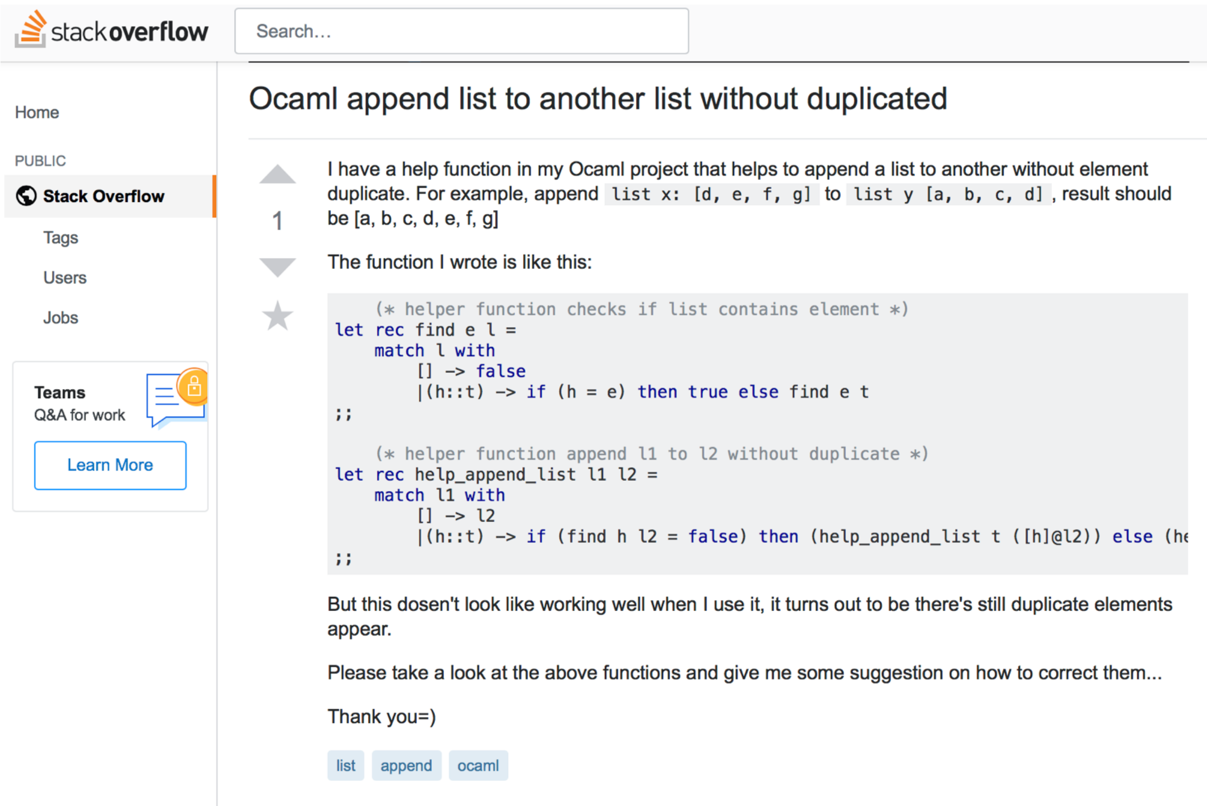
Do not check the duplication in list y

Example3: Append Lists

- Stackoverflow example

```
Test cases :
append_list [1;3] [3;4;5] = [3;4;5;1]
append_list [1] [3;3;4] = [3;4;1]
```

```
(* check whether the element e is in list l *)
let rec find e l =
  match l with
  | [] -> false
  | h::t -> if h = e then true else find e t

(* append l1's elements not in l2 *)
let rec helper l1 l2 =
  match l1 with
  | [] -> l2
  | h::t ->
    if find h l2 = false then helper t (l2@[h])
    else helper t l2

let append_list x y = helper x y
```

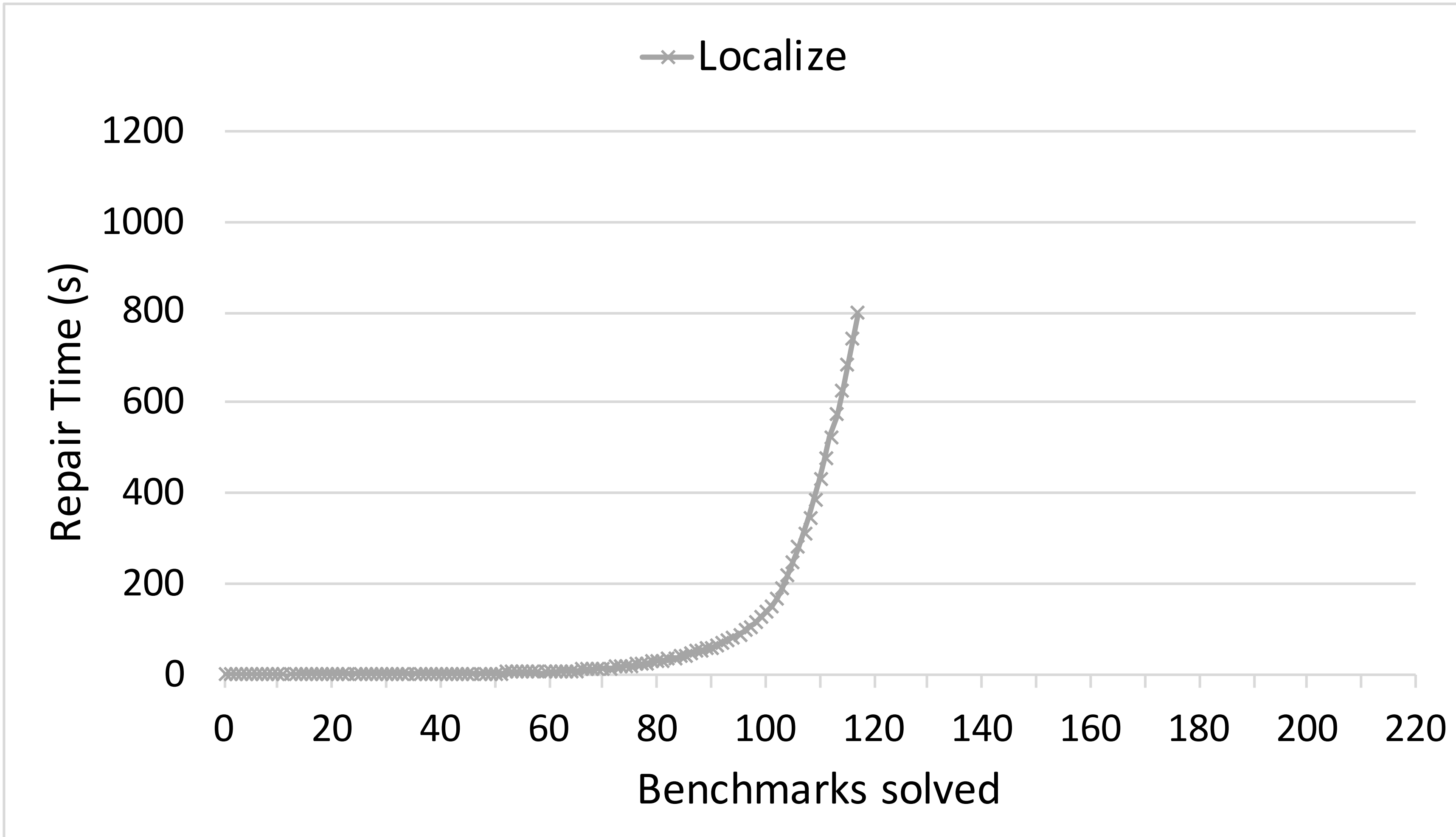


Do not check the duplication in list y

FixML: (helper y [])

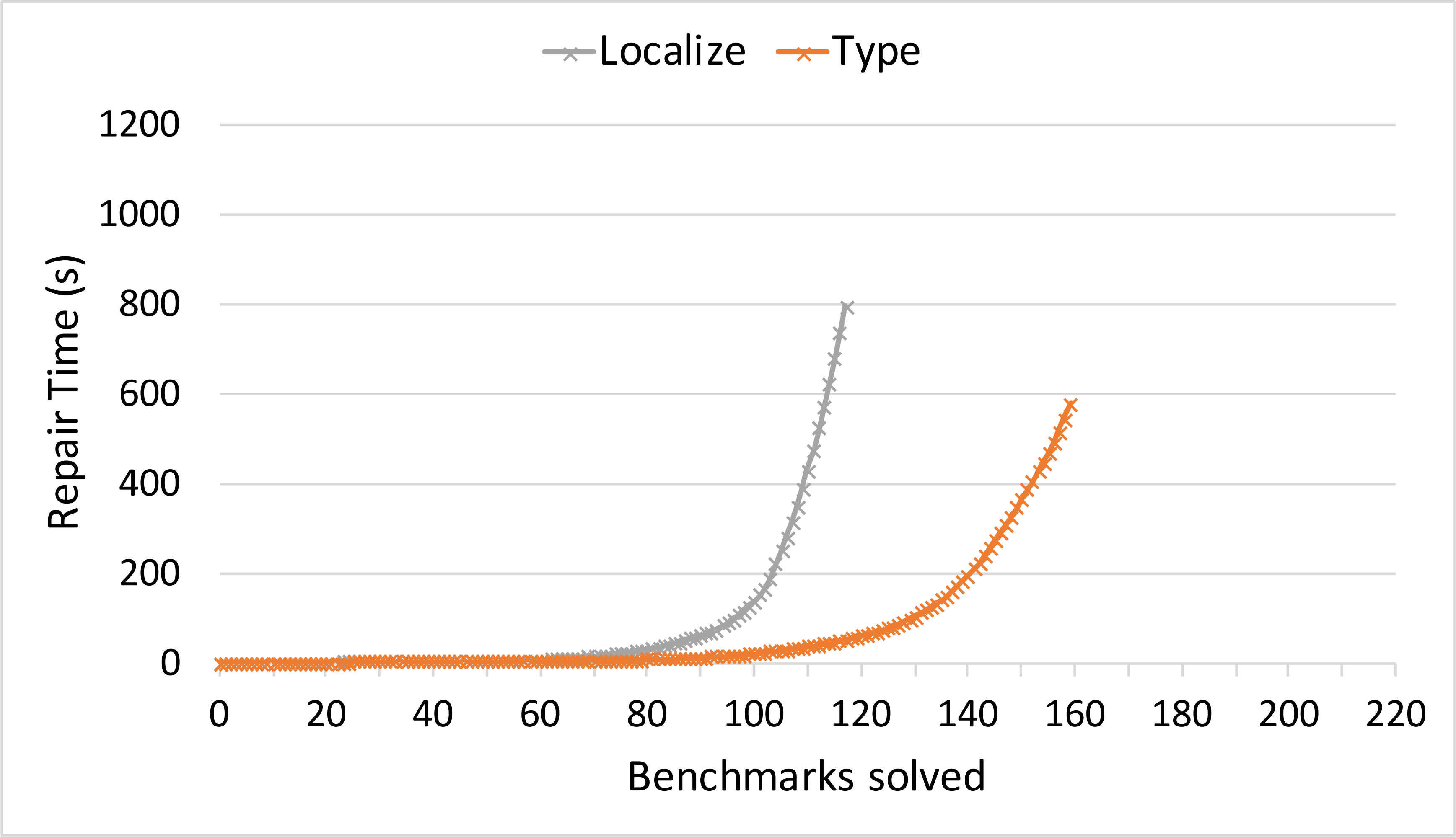
Time: 43 sec

Technique Utility



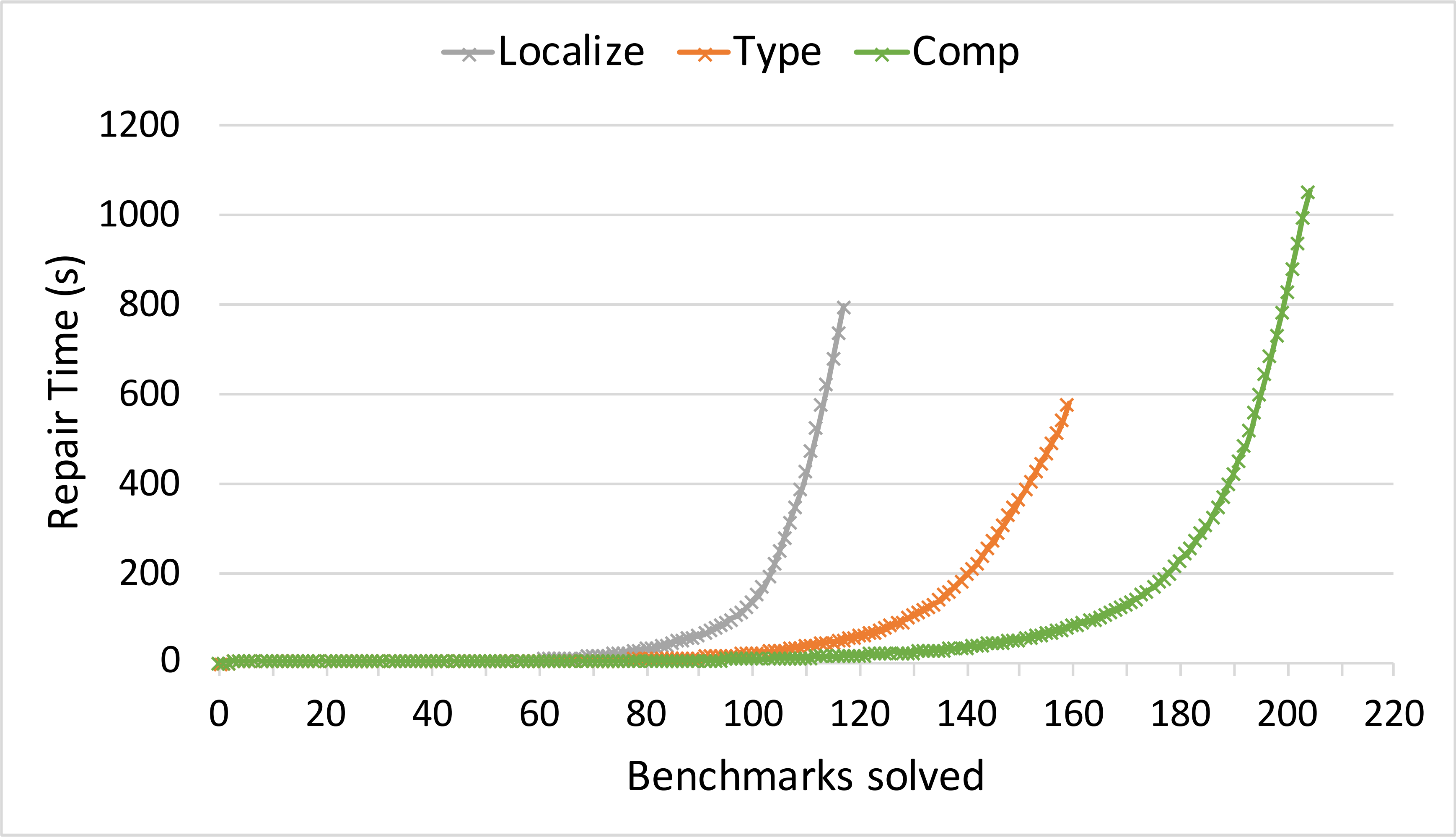
- Only statistical fault localization with enumerative search

Technique Utility



- Statistical fault localization + type-directed search

Technique Utility



- Localization + type-directed search + component reduction

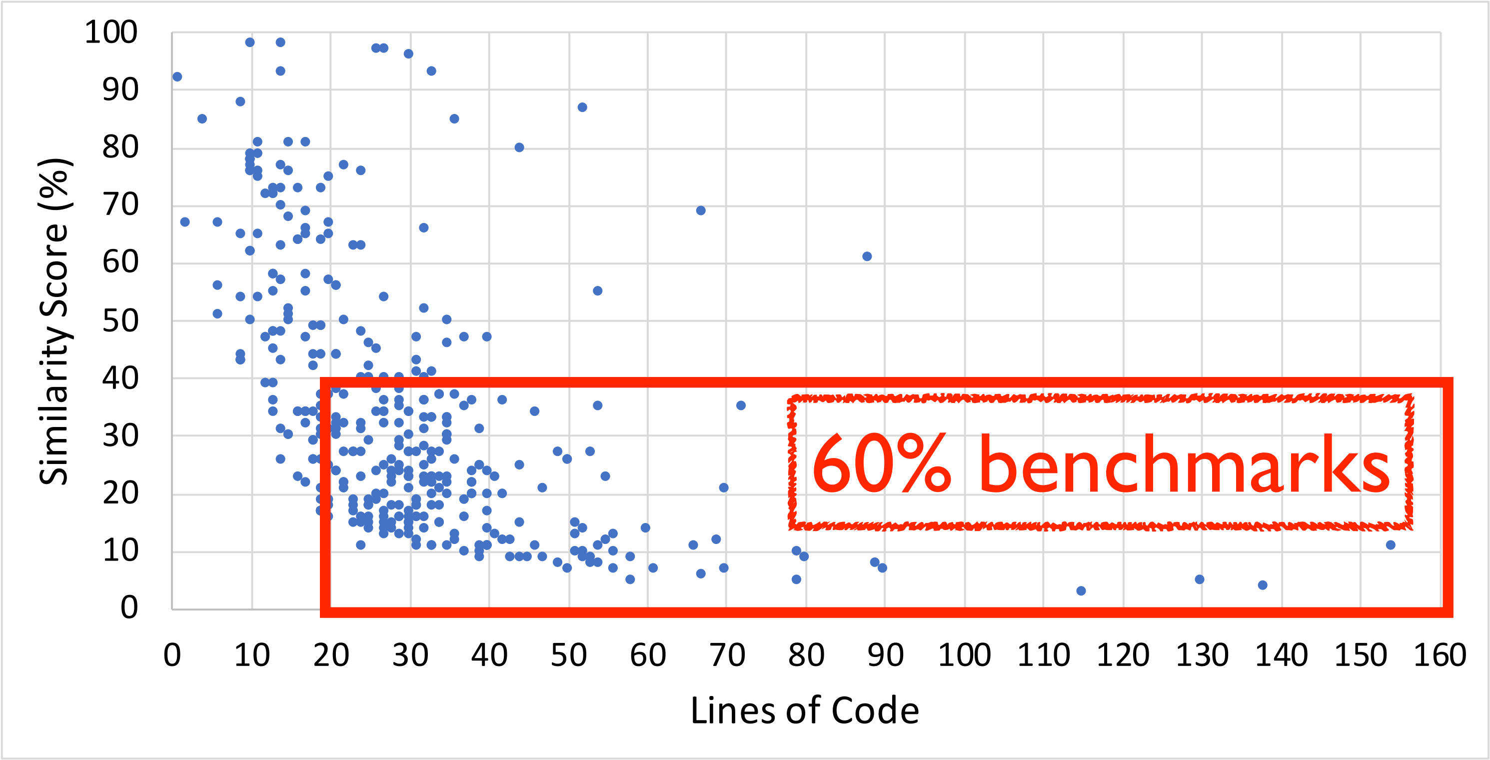
Failure reasons

1. Multiple error

2. Scalability issues

3. Cannot fix by replacing expressions

Results: Similarity



- Calculate the top-1 similarity among the correct programs.

=> Providing feedback by detecting most similar solution is not much helpful.

Motivation

- Evaluation of programming assignment heavily relies on given test cases.
- To properly evaluate students' submissions, instructors manually design these test cases.
- However manually designed test cases sometimes miss some incorrect submissions.

Motivation

- Evaluation of programming assignment heavily relies on given test cases.
- To properly evaluate students' submissions, instructors manually design these test cases.
- However manually designed test cases sometimes miss some incorrect submissions.

Solve this problem by
generating counter-example automatically

Example1: Lambda Calculus

- Check all variables in given lambda calculus is bounded

```
type var = string
type lambda =
  | V of var
  | P of var * lambda
  | C of lambda * lambda

let rec remove (var,p) =
  match p with
  | V x -> if x = var then V “f” else V x
  | P (x,p) -> P (x,remove (var,p))
  | C (p1,p2) -> C (remove (var,p1),remove (var,p2))

let rec check p =
  match p with
  | V x -> if x = “f” then true else false
  | P (x,p) -> check (remove (x,p))
  | C (p1,p2) -> (check p1) && (check p2)
```

Test cases :

check(x) $=$ false
check($\lambda x\,.\,y$) $=$ false
check($\lambda x\,.\,((\lambda y\,.\,y)\ \ x)$) $=$ true

Example1: Lambda Calculus

- Check all variables in given lambda calculus is bounded

```
type var = string
type lambda =
  | V of var
  | P of var * lambda
  | C of lambda * lambda
```

Test cases :

check(x) = false
check($\lambda x . y$) = false
check($\lambda x . ((\lambda y . y) \; x)$) = true

```
let rec remove (var,p) =
  match p with
  | V x -> if x = var then V "f" else V x
  | P (x,p) -> P (x,remove (var,p))
  | C (p1,p2) -> C (remove (var,p1),remove (var,p2))
```

$\boxed{\lambda x . x \to \lambda x . f}$

```
let rec check p =
  match p with
  | V x -> if x = "f" then true else false
  | P (x,p) -> check (remove (x,p))
  | C (p1,p2) -> (check p1) && (check p2)
```

Example 1: Lambda Calculus

- Check all variables in given lambda calculus is bounded

```
type var = string
type lambda =
  | V of var
  | P of var * lambda
  | C of lambda * lambda
```

```
Test cases :
```

check(x) = false
check($\lambda x \,.\, y$) = false
check($\lambda x \,.\, ((\lambda y \,.\, y)\ \ x)$) = true

```
let rec remove (var,p) =
  match p with
  | V x -> if x = var then V “f” else V x
  | P (x,p) -> P (x,remove (var,p))
  | C (p1,p2) -> C (remove (var,p1),remove (var,p2))
```

$\lambda x \,.\, x \rightarrow \lambda x \,.\, f$

```
let rec check p =
  match p with
  | V x -> if x = “f” then true else false
  | P (x,p) -> check (remove (x,p))
  | C (p1,p2) -> (check p1) && (check p2)
```

check(f) = true

Example1: Lambda Calculus

- Check all variables in given lambda calculus is bounded

```
type var = string
type lambda =
  | V of var
  | P of var * lambda
  | C of lambda * lambda
```

Test cases :

$\text{check}(x) = \text{false}$

$\text{check}(\lambda x \,.\, y) = \text{false}$

$\text{check}(\lambda x \,.\, ((\lambda y \,.\, y) \;\; x)) = \text{true}$

```
let rec remove (var,p) =
  match p with
  | V x -> if x = var then V "f" else V x
  | P (x,p) -> P (x,remove (var,p))
  | C (p1,p2) -> C (remove (var,p1),remove (var,p2))
```

$\lambda x \,.\, x \rightarrow \lambda x \,.\, f$

```
let rec check p =
  match p with
  | V x -> if x = "f" then true else false
  | P (x,p) -> check (remove (x,p))
  | C (p1,p2) -> (check p1) && (check p2)
```

$\text{check}(f) = \text{true}$

Counter Example : V "f" = false

Example2: Differentiation

- Generate more complicated input

```
type aexp =
  | Const of int
  | Var of string
  | Power of string * int
  | Times of aexp list
  | Sum of aexp list
```

```
Test cases :
  diff (Const 1,“x”) = Const 0
  diff (Var “x”,“x”) = Const 1
  diff (Power (“x”, 3),“x”) = Times[Const 3; Power (“x”, 2)]
```

```
let rec diff (exp, var) =
  match exp with
  | Const n -> Const 0
  | Var str -> if str = var then Const 1 else Const 0
  | Power (str, n) -> if str = var && n > 0 then Times [Const n; Power (str, n-1)] else Const 0
  | Times lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [Times (diff (hd, var)::tl); Times [hd; diff (Times tl, var)]])
  | Sum lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [diff (hd, var); diff (Times tl, var)])
```

Example2: Differentiation

- Generate more complicated input

```
type aexp =
  | Const of int
  | Var of string
  | Power of string * int
  | Times of aexp list
  | Sum of aexp list

let rec diff (exp, var) =
 match exp with
  | Const n -> Const 0
  | Var str -> if str = var then Const 1 else Const 0
  | Power (str, n) -> if str = var && n > 0 then Times [Const n; Power (str, n-1)] else Const 0
  | Times lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [Times (diff (hd, var)::tl); Times [hd; diff (Times tl, var)]])
  | Sum lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [diff (hd, var); diff (Times tl, var)])
```

```
Test cases :
 diff (Const 1,“x”) = Const 0
 diff (Var “x”,“x”) = Const 1
 diff (Power (“x”, 3),“x”) = Times[Const 3; Power (“x”, 2)]
```

$$\boxed{(f(x) + h(x) + g(x))' = f'(x) + (g(x)h(x))'}$$

Example2: Differentiation

- Generate more complicated input

```
type aexp =
  | Const of int
  | Var of string
  | Power of string * int
  | Times of aexp list
  | Sum of aexp list
```

```
Test cases :
  diff (Const 1,“x”) = Const 0
  diff (Var “x”,“x”) = Const 1
  diff (Power (“x”, 3),“x”) = Times[Const 3; Power (“x”, 2)]
```

```
let rec diff (exp, var) =
  match exp with
  | Const n -> Const 0
  | Var str -> if str = var then Const 1 else Const 0
  | Power (str, n) -> if str = var && n > 0 then Times [Const n; Power (str, n-1)] else Const 0
  | Times lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [Times (diff (hd, var)::tl); Times [hd; diff (Times tl, var)]])
  | Sum lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [diff (hd, var); diff (Times tl, var)])
```

$$\boxed{(f(x)+h(x)+g(x))' = f'(x) + (g(x)h(x))'}$$

Counter Example :
Sum[Var “x”;Var “x”;Const -1] => 2

Example1: Lambda Calculus

- It is impossible for instructors to inspect every corner-cases for evaluation.

```
type var = string
type lambda =
  | V of var
  | P of var * lambda
  | C of lambda * lambda

let rec remove (var,p) =
  match p with
  | V x -> if x = var then V “f” else V x
  | P (x,p) -> P (x,remove (var,p))
  | C (p1,p2) -> C (remove (var,p1),remove (var,p2))

let rec check p =
  match p with
  | V x -> if x = “f” then true else false
  | P (x,p) -> check (remove (x,p))
  | C (p1,p2) -> (check p1) && (check p2)
```

Example1: Lambda Calculus

- It is impossible for instructors to inspect every corner-cases for evaluation.

```
type var = string
type lambda =
  | V of var
  | P of var * lambda
  | C of lambda * lambda

let rec remove (var,p) =
  match p with
  | V x -> if x = var then V "f" else V x
  | P (x,p) -> P (x,remove (var,p))
  | C (p1,p2) -> C (remove (var,p1),remove (var,p2))

let rec check p =
  match p with
  | V x -> if x = "f" then true else false
  | P (x,p) -> check (remove (x,p))
  | C (p1,p2) -> (check p1) && (check p2)
```

Example1: Lambda Calculus

- It is impossible for instructors to inspect every corner-cases for evaluation.

```
type var = string
type lambda =
  | V of var
  | P of var * lambda
  | C of lambda * lambda

let rec remove (var,p) =
  match p with
  | V x -> if x = var then V "f" else V x
  | P (x,p) -> P (x,remove (var,p))
  | C (p1,p2) -> C (remove (var,p1),remove (var,p2))

let rec check p =
  match p with
  | V x -> if x = "f" then true else false
  | P (x,p) -> check (remove (x,p))
  | C (p1,p2) -> (check p1) && (check p2)
```

Counter Example : V "f" => false

Example2: Differentiation

- It is hard to identify error in complicated programs and generating error-triggering input is also nontrivial.

```
type aexp =
  | Const of int
  | Var of string
  | Power of string * int
  | Times of aexp list
  | Sum of aexp list

let rec diff (exp, var) =
  match exp with
  | Const n -> Const 0
  | Var str -> if str = var then Const 1 else Const 0
  | Power (str, n) -> if str = var && n > 0 then Times [Const n; Power (str, n-1)] else Const 0
  | Times lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [Times (diff (hd, var)::tl); Times [hd; diff (Times tl, var)]])
  | Sum lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [diff (hd, var); diff (Times tl, var)])
```

Example2: Differentiation

- It is hard to identify error in complicated programs and generating error-triggering input is also nontrivial.

```
type aexp =
  | Const of int
  | Var of string
  | Power of string * int
  | Times of aexp list
  | Sum of aexp list

let rec diff (exp, var) =
  match exp with
  | Const n -> Const 0
  | Var str -> if str = var then Const 1 else Const 0
  | Power (str, n) -> if str = var && n > 0 then Times [Const n; Power (str, n-1)] else Const 0
  | Times lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [Times (diff (hd, var)::tl); Times [hd; diff (Times tl, var)]])
  | Sum lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [diff (hd, var); diff (Times tl, var)])
```

$$\boxed{(f(x) + h(x) + g(x))' = f'(x) + (g(x)h(x))'}$$

Example2: Differentiation

- It is hard to identify error in complicated programs and generating error-triggering input is also nontrivial.

```
type aexp =
  | Const of int
  | Var of string
  | Power of string * int
  | Times of aexp list
  | Sum of aexp list

let rec diff (exp, var) =
 match exp with
  | Const n -> Const 0
  | Var str -> if str = var then Const 1 else Const 0
  | Power (str, n) -> if str = var && n > 0 then Times [Const n; Power (str, n-1)] else Const 0
  | Times lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [Times (diff (hd, var)::tl); Times [hd; diff (Times tl, var)]])
  | Sum lst ->
    (match lst with
    | [] -> Const 0
    | [hd] -> diff (hd, var)
    | hd::tl -> Sum [diff (hd, var); diff (Times tl, var)])
```

$$\boxed{(f(x) + h(x) + g(x))' = f'(x) + (g(x)h(x))'}$$

Counter Example :
Sum[Var “x”;Var “x”;Const -1] => 2