



# Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments

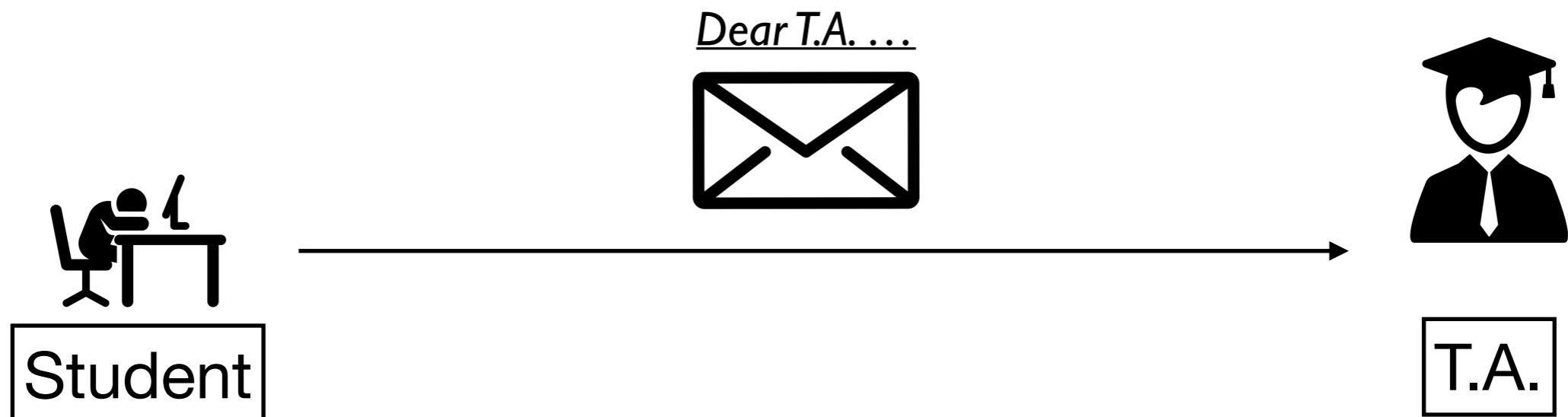
Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh  
Korea University



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
        | (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 generate 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
        | (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
  in

  let rec simplify : aexp -> env -> int -> aexp list
  = fun aexp env flag ->
    match aexp with
    | SUM lst -> SUM(List.map (fun aexp -> doDiff(aexp, x)) lst)
  in

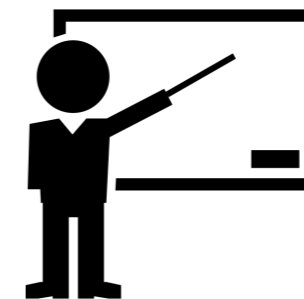
  let rec simplify : aexp -> env -> int -> aexp list
  = fun aexp env flag ->
    match aexp with
    | SUM lst ->
      (
        match lst with
        | (CONST c)::tl -> simplify (SUM tl) (updateEnv ("const", c, 0) env 0) 0
        | (VAR x)::tl -> simplify (SUM tl) (updateEnv (x, 1, 1) env 0) 0
  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

Just Replace "[ ]"  
by "SUM tl"

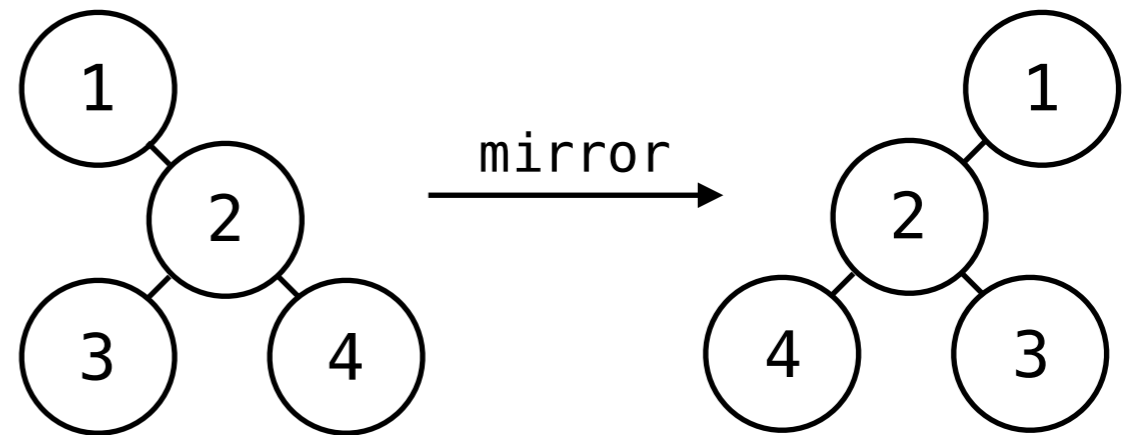


Automated T.A.

# Example I: 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)
```

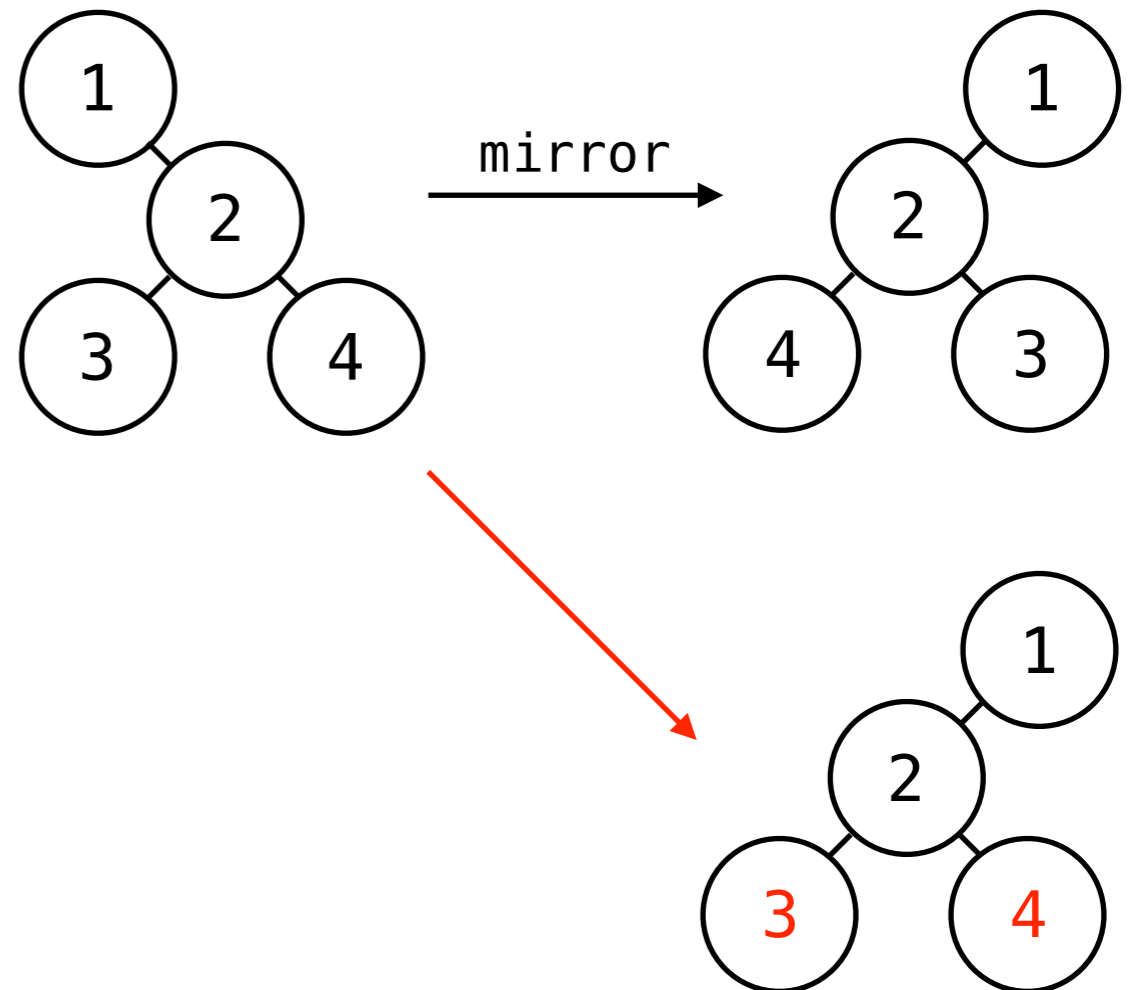


# Example I: Mirroring Tree

- 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 I: Mirroring Tree

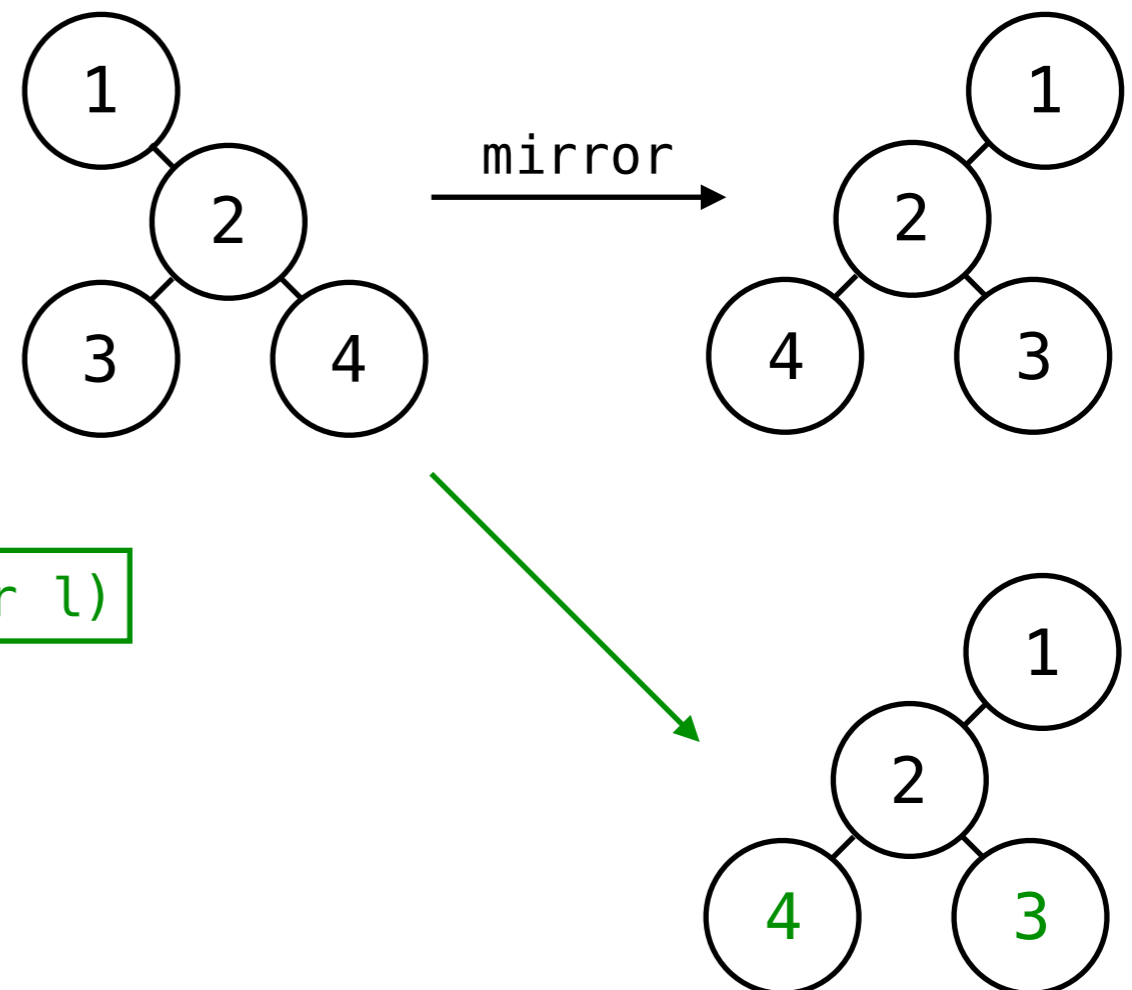
- Warming up!

```
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:

$$2 + (n_1 - 1) \times (n_1 \times (n_2 - 1))$$

# 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:

$$2 + (n_1 - 1) \times (n_1 \times (n_2 - 1))$$

Correct formula:

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

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

Time: 22 sec

# 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]
```

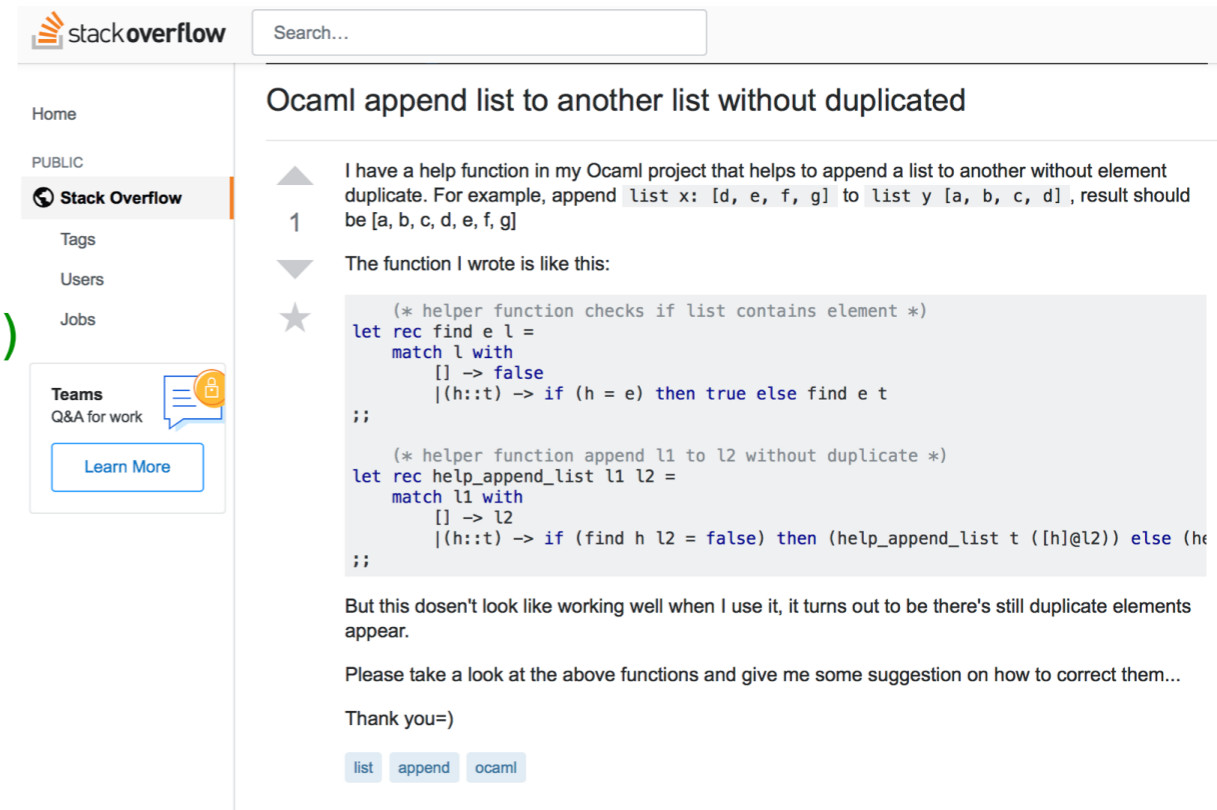
(\* check whether the element e is in list l \*)

```
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
```



The screenshot shows a Stack Overflow page with the following content:

- Stack Overflow logo and search bar at the top.
- Navigation menu on the left: Home, PUBLIC, Stack Overflow, Tags, Users, Jobs, Teams (Q&A for work), and a Learn More button.
- Question title: "Ocaml append list to another list without duplicated".
- Question body:
  - Paragraph 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]"
  - Paragraph 2: "The function I wrote is like this:"
  - Code block showing two OCaml functions:

```
(* 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 (h
```
  - Paragraph 3: "But this doesn't look like working well when I use it, it turns out to be there's still duplicate elements appear."
  - Paragraph 4: "Please take a look at the above functions and give me some suggestion on how to correct them..."
  - Paragraph 5: "Thank you="
- Tags: list, append, ocaml

# 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]
```

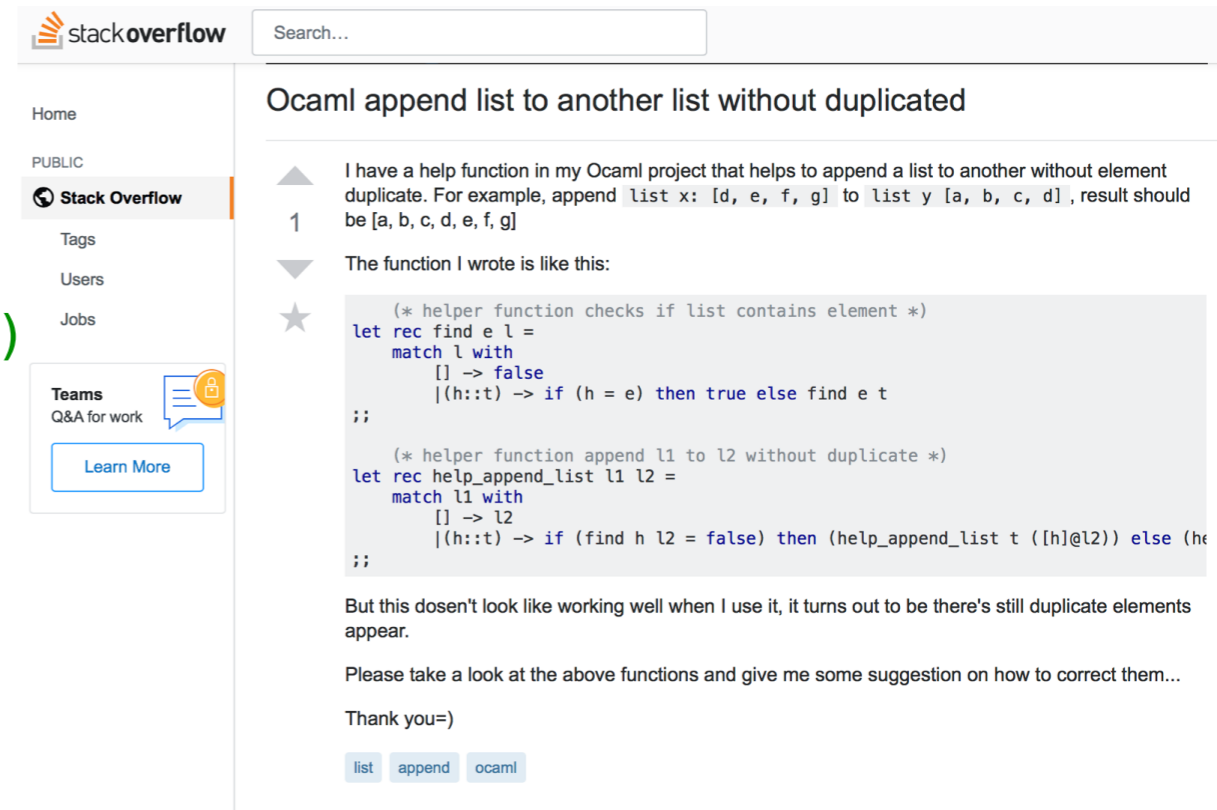
```
(* check whether the element e is in list l *)
```

```
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
```



The screenshot shows a Stack Overflow question titled "Ocaml append list to another list without duplicated". The user asks for help with a function to append a list to another without duplicates. They provide two OCaml functions: `find` and `help_append_list`. The `find` function checks if an element is in a list. The `help_append_list` function appends elements from `l1` to `l2` only if they are not already in `l2`. The user reports that their implementation still produces duplicate elements and asks for suggestions.

```
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

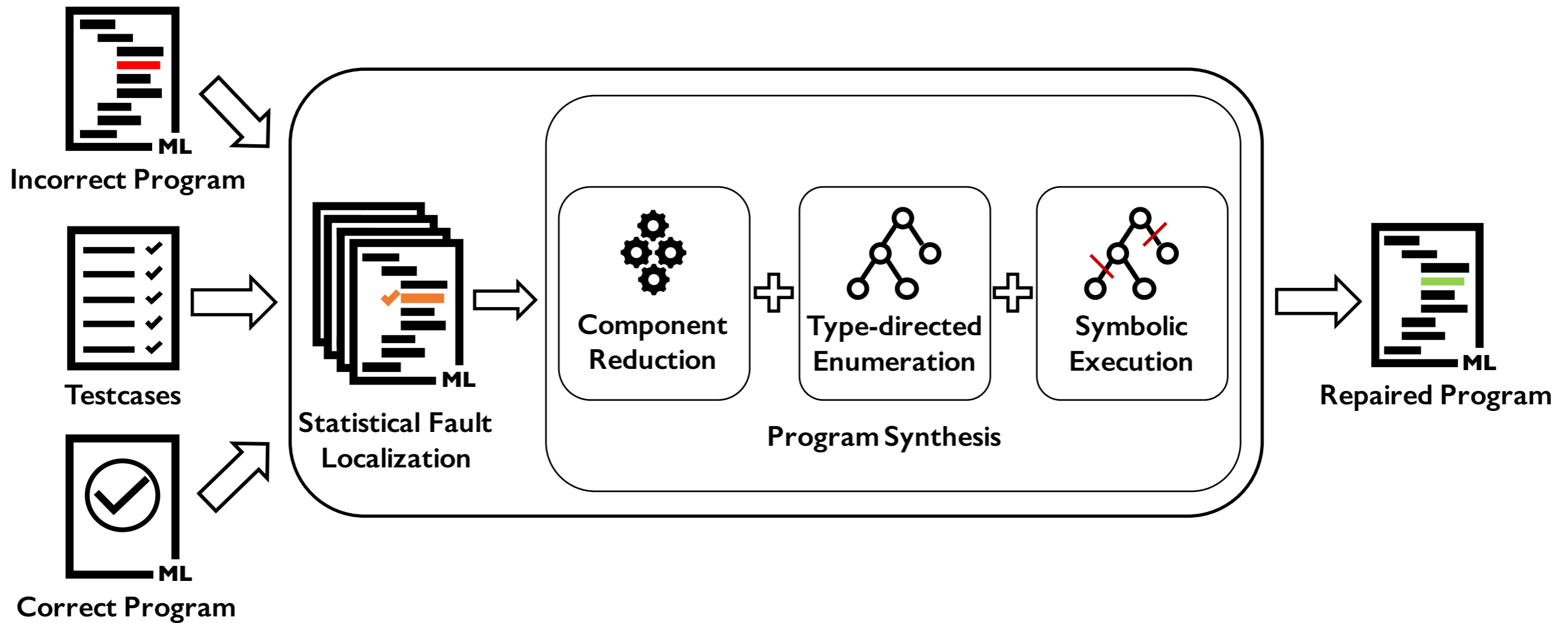
FixML: (helper y [])

Time: 43 sec

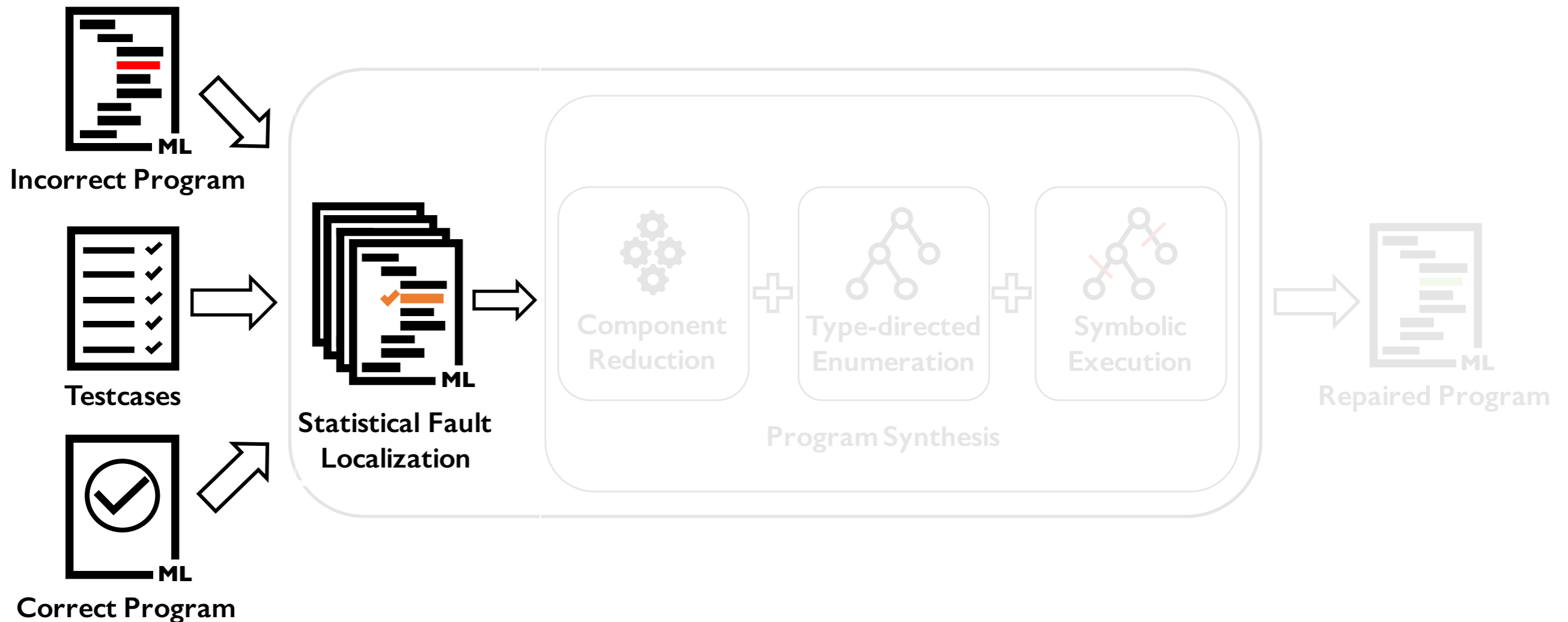
The screenshot shows a Stack Overflow question titled "Ocaml append list to another list without duplicated". The user asks for help with a function to append a list to another without duplicates. They provide two OCaml functions: `find` and `help_append_list`. The `find` function checks if an element is in a list. The `help_append_list` function appends elements from `l1` to `l2` only if they are not already in `l2`. The user reports that their implementation still produces duplicate elements and asks for suggestions.

# 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:

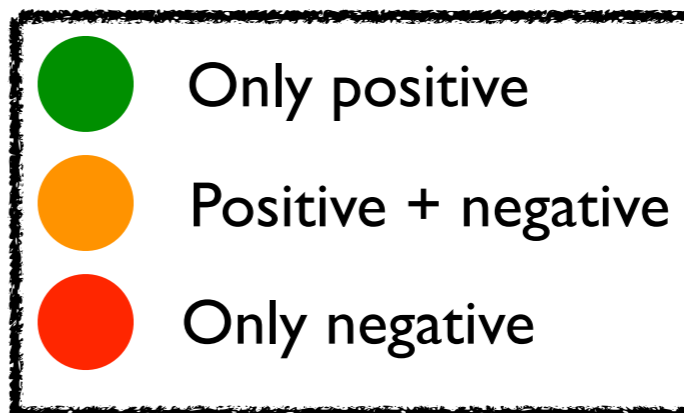
```
let rec natmul n1 n2 =  
  match n1 with  
  | ZERO -> ZERO  
  | SUCC ZERO -> n2  
  | SUCC n1' ->  
    SUCC( match n2 with  
      | ZERO -> ZERO  
      | SUCC ZERO -> SUCC ZERO  
      | SUCC n2' -> SUCC (natmul n1' (natmul n1 n2'))  
    )
```

Test cases :

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

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

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



More **negative**, less **positive** => more 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
```

Return a set of scored partial programs

$P_1$

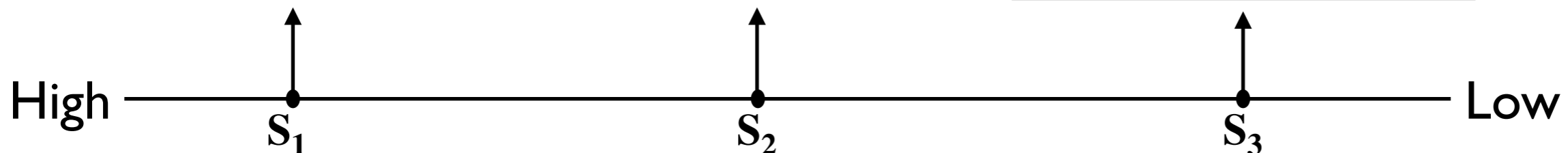
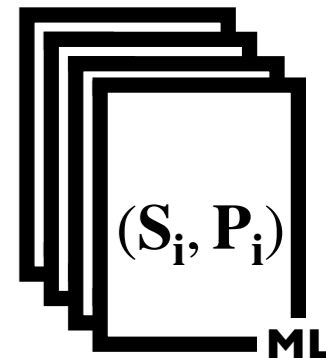
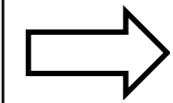
```
let rec natmul n1 n2 =  
  match n1 with  
  | ZERO -> ZERO  
  | SUCC ZERO -> n2  
  | SUCC n1' -> ?
```

$P_2$

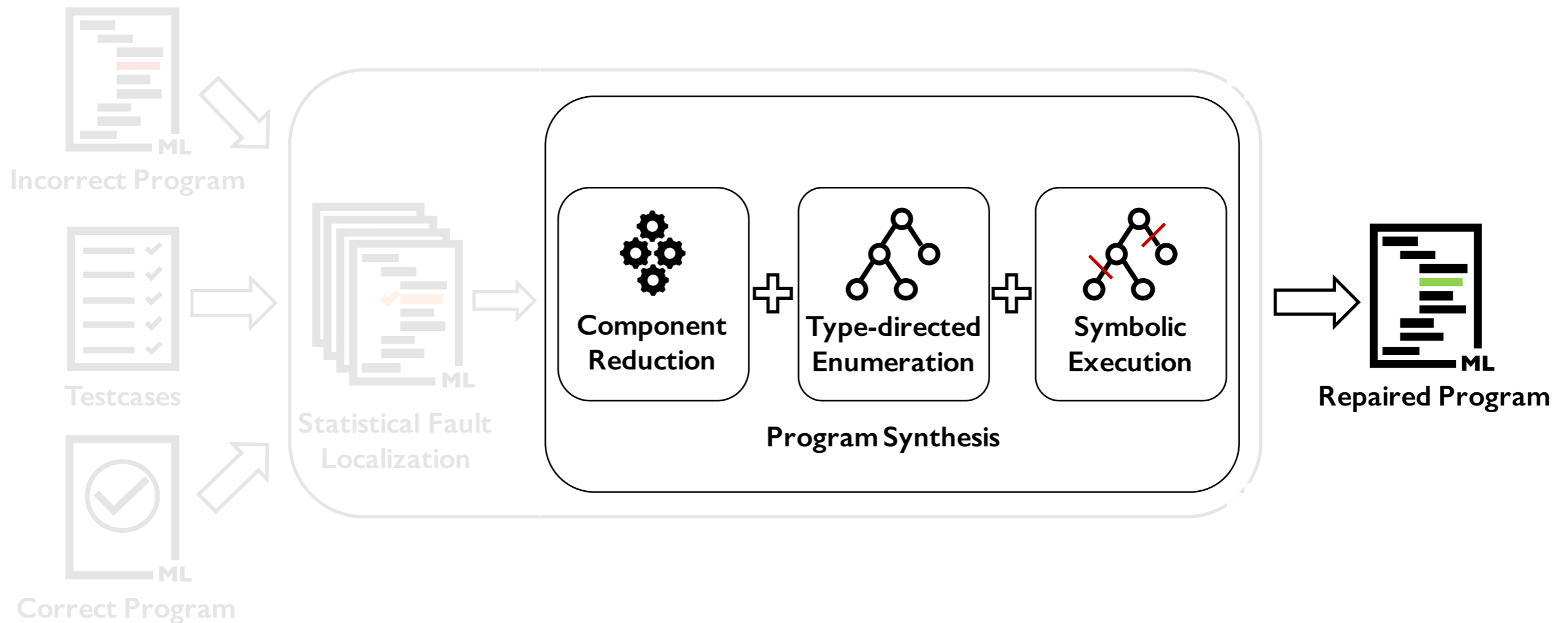
```
let rec natmul n1 n2 =  
  match n1 with  
  | ZERO -> ZERO  
  | SUCC ZERO -> ?  
  | SUCC n1' -> ...
```

$P_3$

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

```
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)
```

...

# 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

Still inefficient in our cases!

```
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 ?
```

# Our Solution

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

# Technique I: 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

```
 $E ::= () \mid n \mid x \mid \text{true} \mid \text{false} \mid \text{str} \mid \lambda x.E \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid E_1 / E_2 \mid E_1 \bmod E_2 \mid -E$   
| not  $E \mid E_1 \parallel E_2 \mid E_1 \&\&E_2 \mid E_1 < E_2 \mid E_1 > E_2 \mid E_1 \leq E_2 \mid E_1 \geq E_2 \mid E_1 = E_2 \mid E_1 <> E_2$   
|  $E_1 E_2 \mid E_1 :: E_2 \mid E_1 @ E_2 \mid E_1 \wedge E_2 \mid \text{raise } E \mid (E_1, \dots, E_k) \mid [E_1; \dots; E_k]$   
| if  $E_1 E_2 E_3 \mid c(E_1, \dots, E_k) \mid \text{let } x = E_1 \text{ in } E_2 \mid \text{let rec } f(x) = E_1 \text{ in } E_2$   
| let  $x_1 = E_1 \text{ and } \dots \text{ and } x_k = E_k \text{ in } E \mid \text{let rec } f_1(x_1) = E_1 \text{ and } \dots \text{ and } f_k(x_k) = E_k \text{ in } E$   
| match  $E \text{ with } p_1 \rightarrow E_1 \mid \dots \mid p_k \rightarrow E_k$   
|  $\square$ 
```

Solution:

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

# Technique I: 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

```
 $E ::= () \mid n \mid x \mid \text{true} \mid \text{false} \mid \text{str} \mid \lambda x.E \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid E_1 / E_2 \mid E_1 \bmod E_2 \mid -E$   
| not  $E \mid E_1 \parallel E_2 \mid E_1 \&\&E_2 \mid E_1 < E_2 \mid E_1 > E_2 \mid E_1 \leq E_2 \mid E_1 \geq E_2 \mid E_1 = E_2 \mid E_1 <> E_2$   
|  $E_1 E_2 \mid E_1 :: E_2 \mid E_1 @ E_2 \mid E_1 \wedge E_2 \mid \text{raise } E \mid (E_1, \dots, E_k) \mid [E_1; \dots; E_k]$   
| if  $E_1 E_2 E_3 \mid c(E_1, \dots, E_k) \mid \text{let } x = E_1 \text{ in } E_2 \mid \text{let rec } f(x) = E_1 \text{ in } E_2$   
| let  $x_1 = E_1 \text{ and } \dots \text{ and } x_k = E_k \text{ in } E \mid \text{let rec } f_1(x_1) = E_1 \text{ and } \dots \text{ and } f_k(x_k) = E_k \text{ in } E$   
| match  $E \text{ with } p_1 \rightarrow E_1 \mid \dots \mid p_k \rightarrow E_k$   
|  $\square$ 
```

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 I: 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:

<sup>4</sup>  
~~36 expressions~~

```
 $E ::= () \mid n \mid x \mid \text{true} \mid \text{false} \mid \text{str} \mid \lambda x.E \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid E_1 / E_2 \mid E_1 \bmod E_2 \mid -E$   
| not  $E$  |  $E_1 \parallel E_2$  |  $E_1 \&\&E_2$  |  $E_1 < E_2$  |  $E_1 > E_2$  |  $E_1 \leq E_2$  |  $E_1 \geq E_2$  |  $E_1 = E_2$  |  $E_1 <> E_2$   
|  $E_1 E_2$  |  $E_1 :: E_2$  |  $E_1 @ E_2$  |  $E_1 \wedge E_2$  | raise  $E$  |  $(E_1, \dots, E_k)$  |  $[E_1; \dots; E_k]$   
| if  $E_1 E_2 E_3$  |  $c(E_1, \dots, E_k)$  | let  $x = E_1$  in  $E_2$  | let rec  $f(x) = E_1$  in  $E_2$   
| let  $x_1 = E_1$  and ... and  $x_k = E_k$  in  $E$  | let rec  $f_1(x_1) = E_1$  and ... and  $f_k(x_k) = E_k$  in  $E$   
| match  $E$  with  $p_1 \rightarrow E_1 \mid \dots \mid p_k \rightarrow E_k$   
|  $\square$ 
```

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'}

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'
```

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

**n1 = SUCC n1'**

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

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

Choosing the minimal set of variables through data-flow analysis

# Technique 3: Symbolic Execution

- 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: Symbolic Execution

- 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: Symbolic Execution

- 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: Symbolic Execution

- 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

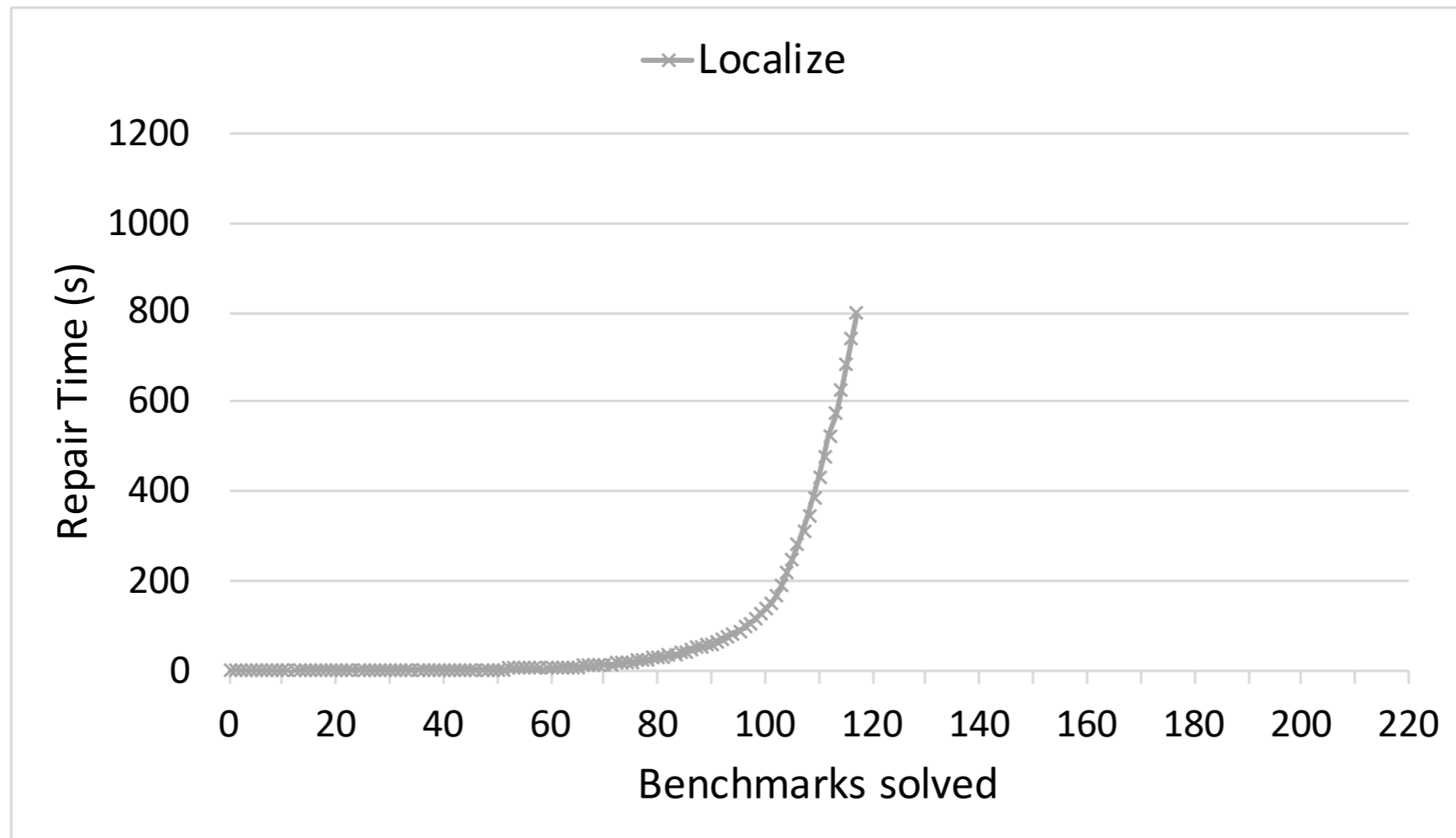
- Evaluated on **497 programs written in OCaml** with logical errors from 13 assignments
- Various task **from introductory to advanced (2-154 lines)** problems
- Conducted **user study** with 18 students.

# 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)	<b>Introductory</b> <b>Fix: 89%</b> <b>Time: 2.5 sec</b>
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)	<b>Intermediate</b> <b>Fix: 48%</b> <b>Time: 11.6 sec</b>
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)	<b>Advanced</b> <b>Fix: 30%</b> <b>Time: 4.8 sec</b>
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

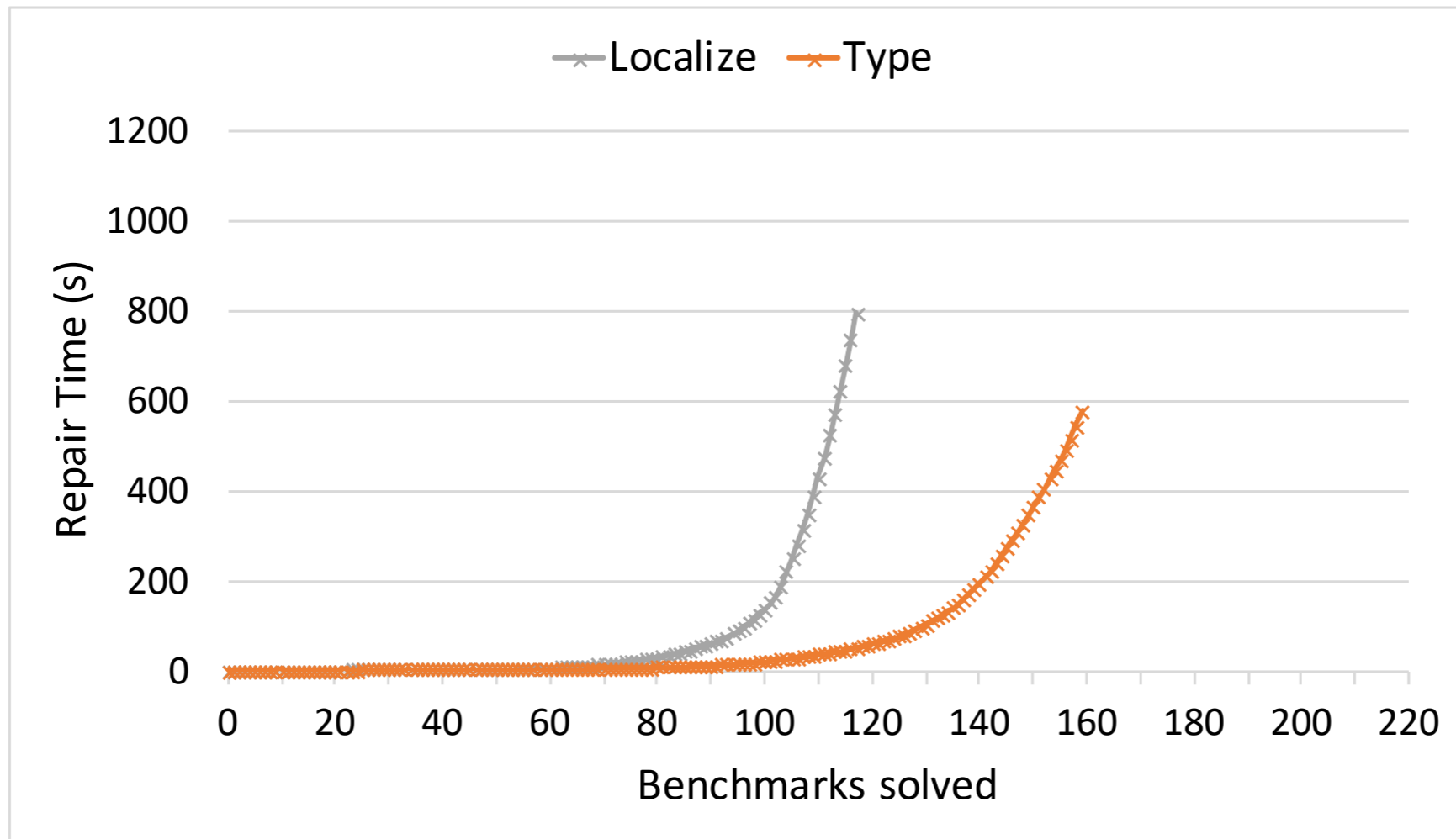
# Technique Utility



- Only statistical fault localization with enumerative search

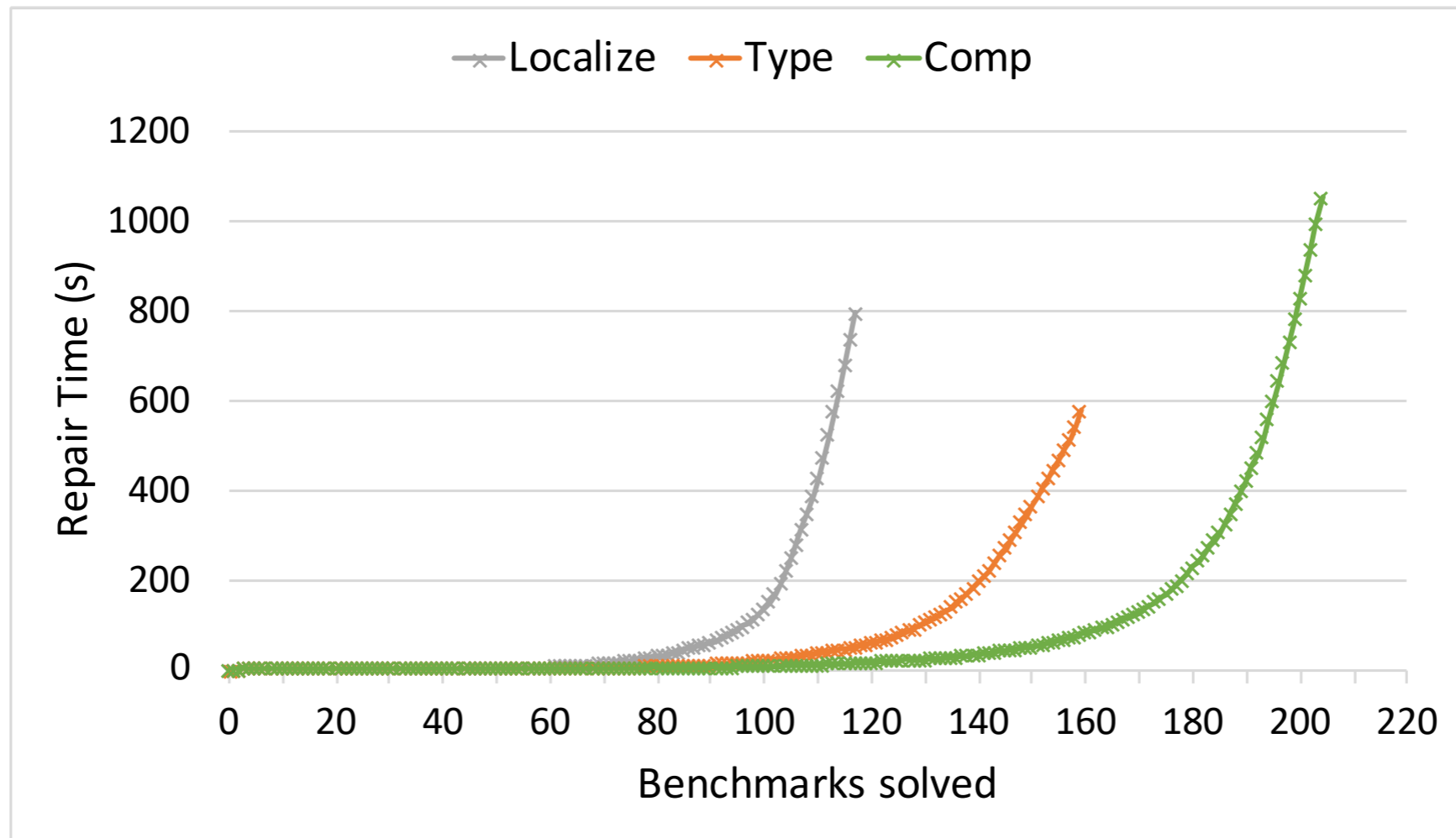


# Technique Utility



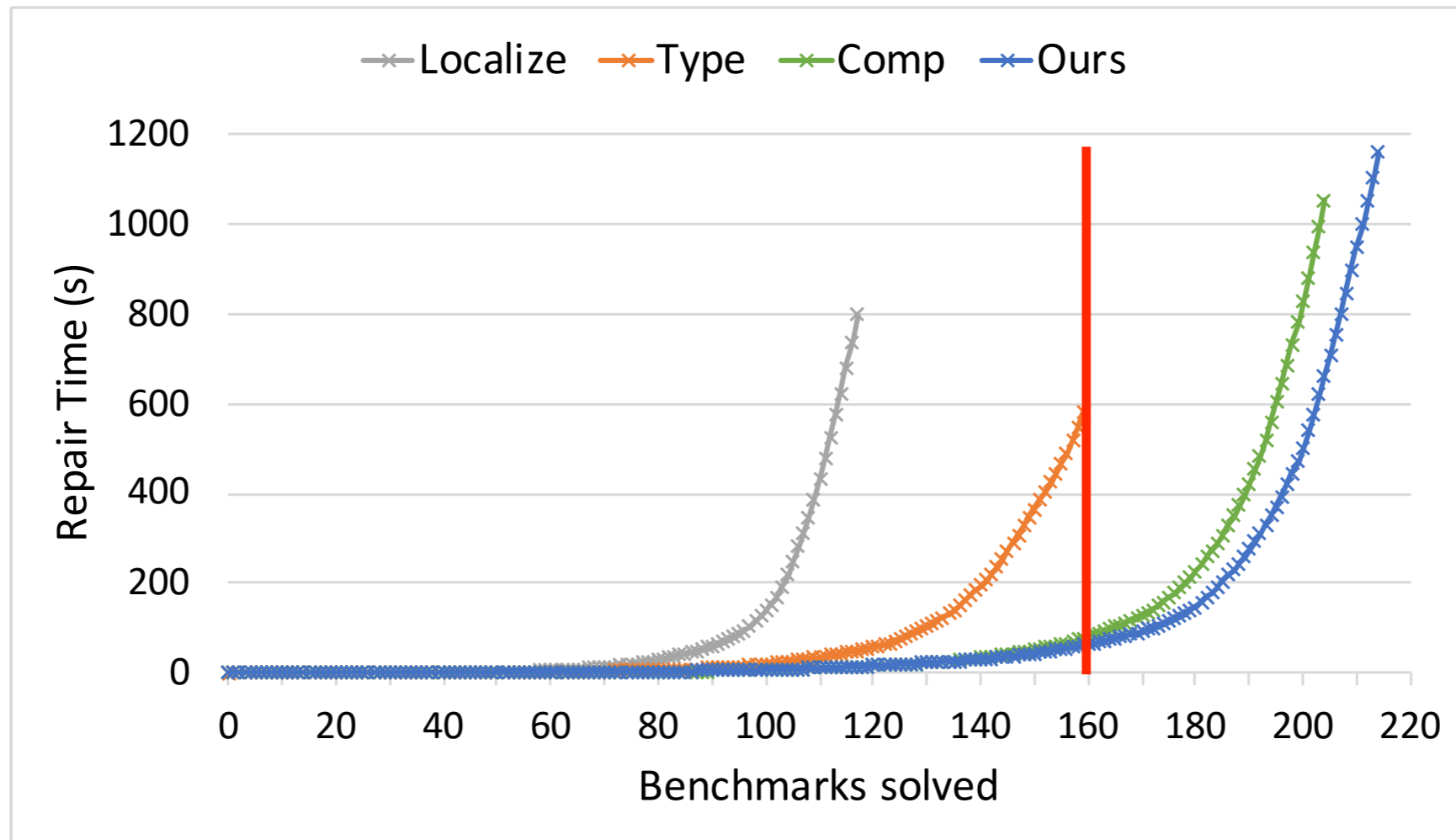
- Statistical fault localization + type-directed search

# Technique Utility



- Localization + type-directed search + component reduction

# Technique Utility



- Localization + type + component + symbolic execution
- Compare to Type : 579sec vs 65sec (x 8.9 faster)  
160 vs 214 (54 submissions more)

# User Study

- Conducted user study with 18 undergraduate students.
- Requested to solve three problems.
- Provide feedback and survey it.

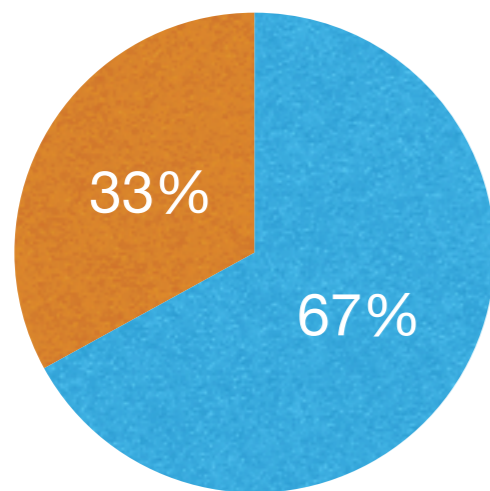
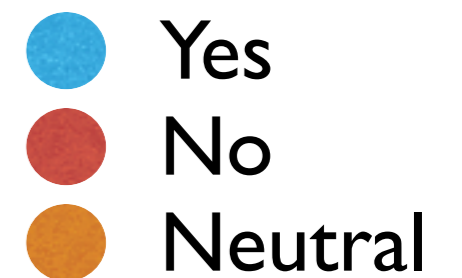
# Helpfulness

Q1. Does the tool generate better corrections?

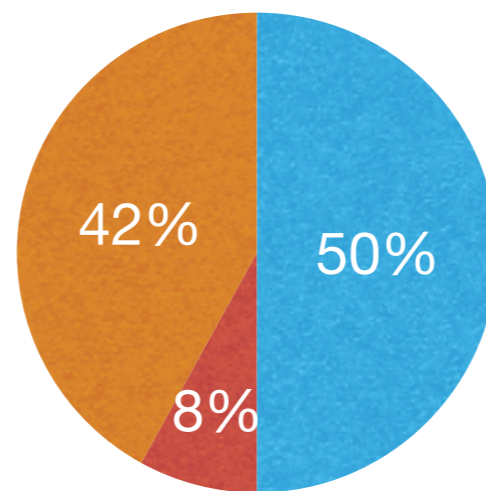
Q2. Does the feedback help to understand your mistakes?

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

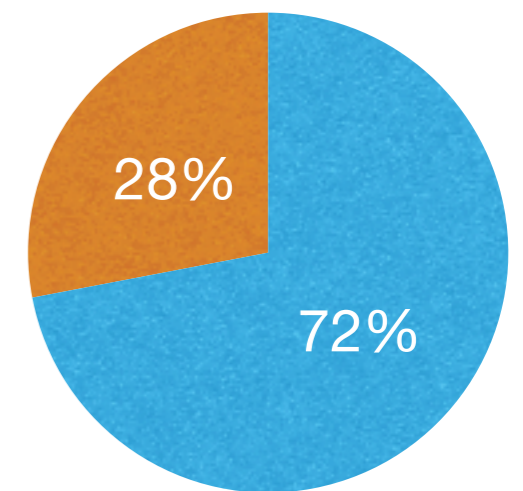
Agreed with the helpfulness!



Q1



Q2



Q3

# Helpfulness

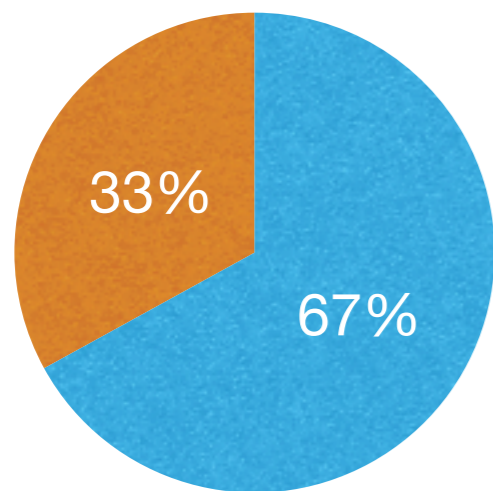
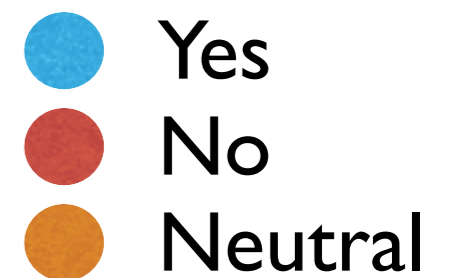
Q1. Does the tool generate better corrections?

Q2. Does the feedback help to understand your mistakes?

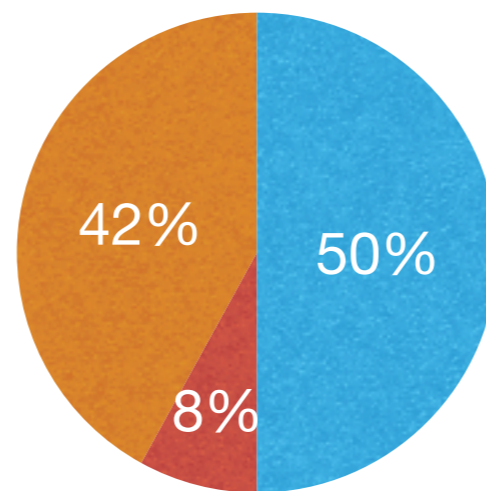
Q3. Is the tool overall useful in learning functional programming?



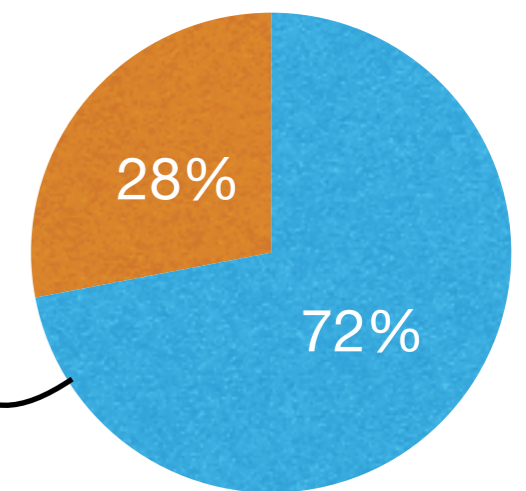
“Since the tool generates the **optimal patch**, we can **learn the programming skills** from it”



Q1



Q2



Q3

# Limitations

- Multiple Errors

Buggy implementation:

```
let rec eval f =  
  match f with  
  | True -> true | False -> false  
  | Not e -> if e = True then false else true  
  | AndAlso (e1, e2) -> if e1 = True && e2 = True then true else false  
  | OrElse (e1, e2) -> if e1 = False && e2 = False then false else true  
  | Imply (e1, e2) -> if e1 = True && e2 = False then false else true |
```

- Dependence on test cases

Buggy implementation:

```
let rec sigma f a b =  
  if f a != f b then f b + sigma f a (b-1) else f b
```

Feedback: replace `(f a != f b)` by `(a != b)`

Test cases :

```
(fun x-> x * x) 1 3 => 14
```

```
(fun x-> x + x) 1 3 => 12
```

```
(fun x-> (x*x)+x) 3 6 => 104
```

```
(fun x -> x mod 3) 1 5 => 6
```

# 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.kroea.ac.kr>

The screenshot displays the COSE212 - Programming Languages web interface. On the left, a navigation menu includes 'Home', 'Assignment Policy', 'Homework Select', 'Feedback', 'Exercise', 'exercise', 'factorial', and 'Option'. The main area is split into two panels: 'original.ml' and 'feedback.ml'. The 'original.ml' panel shows the following code:

```
1 let factorial : int -> int
2 = fun n -> if(n=0) then 0 else n*factorial(n-1)
```

The 'feedback.ml' panel shows the corrected code:

```
1 let rec factorial : int -> int
2 = fun n -> if(n=0) then 1 else n*factorial(n-1)
3
```

At the bottom, there are 'Run' and 'Submit' buttons. Below the code panels, a feedback message is displayed:

```
1,2c1,2
< let factorial : int -> int
< = fun n -> (*TODO*)
\ No newline at end of file
---
> let rec factorial : int -> int
> = fun n -> if(n=0) then 1 else n*factorial(n-1)
```



# Summary

- The first system providing personalized feedback of logical errors for functional programs
- Code and our data: <https://github.com/kupl/FixML>
- Tool usage: <https://tryml.kroea.ac.kr>

The screenshot shows the FixML tool interface. At the top, there's a navigation bar with 'Home', 'COSE212 - Programming Languages', 'My Info', and 'Log Out'. On the left, there's a sidebar with 'Assignment Policy', 'Homework Select', 'Feedback', 'Exercise', 'exercise', 'factorial', and 'Option'. The main area is split into two panels: 'original.ml' and 'feedback.ml'. The 'original.ml' panel shows the following code:

```
1 let factorial : int -> int
2 = fun n -> if(n=0) then 0 else n*factorial(n-1)
```

The 'feedback.ml' panel shows the corrected code:

```
1 let rec factorial : int -> int
2 = fun n -> if(n=0) then 1 else n*factorial(n-1)
3
```

At the bottom, there's a 'Run' button and a 'Submit' button. Below the code, there's a feedback message: '1,2c1,2 < let factorial : int -> int <= fun n -> (\*TODO\*) \ No newline at end of file -> let rec factorial : int -> int >= fun n -> if(n=0) then 1 else n\*factorial(n-1)'. A large black box with white text 'Thank you for listening!' is overlaid on the bottom right of the screenshot.