# **OOPSLA24** Statically Contextualizing Large Language Models with Typed Holes



### Andrew Blinn · Xiang (Kevin) Li · June Hyung (Jacob) Kim · Cyrus Omar

Future of Programming Lab **University of Michigan** 



## **Problem:**

LLMs often produce broken code, sometimes due to lack of appropriate context. How can we best proactively provide context to inform code completions?



# **Problem:**

LLMs often produce broken code, sometimes due to lack of appropriate context. How can we best proactively provide context to inform code completions?

# **Our Approach:**

A conversation between programmer, language server, and language model, using typed holes to bridge cursor-local and repository-wide semantic information





# **Problem:**

LLMs often produce broken code, sometimes due to lack of appropriate context. How can we best proactively provide context to inform code completions?

# **Our Approach:**

A conversation between programmer, language server, and language model, using typed holes to bridge cursor-local and repository-wide semantic information

# **Our Evaluation:**

We implemented and evaluated our method in Hazel, our lab's typed functional programming language and live program sketching environment

We (partially) reproduced our method and results in TypeScript









# **MOTIVATION Contextualizing Context**





# Complete this program sketch

# Update the EmojiPainter app #
let update: (Model, Action) -> Model =
??
in



# Prompting an LLM is like lightning onboarding

# Update the EmojiPainter app # let update: (Model, Action) -> Model = ?? in

9

- Text window
- Vector Retrieval
- Repository-level

```
(go,idx, (descr: String, status: Bool)) ->
          Dive
            Style()
             Display("flex"),
            Gap("lem"))),
            OnClick(fun () -> go(ToggleTodo(idx)))),
              Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []),
              Div([], (Text(descr)1),
              Text(if status then "Completed" else "Pending")
       let todos_deck: (Action -> Event, (Todo)) -> Node =
        fun go, todos ->
          if (List.is_empty(todos))
          then Text("You're caught up")
          else Div(
            (Create("class", "todos")),
            (Text("todos:"))
           @ List.mapi(fun i,t ->todo_card(go,i,t), todos)) in
       let add_button: (Action -> Event) -> Node =
        fun go -> Div(t
          Style()
           Display("flex"),
            JustifyContent("center"),
           BackgroundColor("#986"),
           BorderRadius("0.3em"),
           Cursor("pointer"))).
          OnClick(fun () -> go(AddTodo))),
          (Text("Add Todo"))) in
       let buffer: (Action -> Event) -> Node =
        fun go -> Div(
           (TextInput((OnInput(fun s -> go(UpdateBuffer(s))))), []))) in
       let bool_eq: (Bool, Bool) -> Bool =
        fun a, b → a && b \/ !a && !b in
       let List.equal: (? -> Bool, (?), (?)) -> Bool =
        fun p, xs, ys →
         case xs, ys
           [], [] => true
            x::xs, y::ys ⇒ p(x, y) && List.equal(p, xs, ys)
           _ => false end in
       let List.cons: (?, (?)) -> (?) =
        fun x, xs -> x::xs in
# Update the EmojiPainter app #
.et update: (Model, Action) -> Model
??
                -> 2 =
       fun f, xs, acc ->
         case xs
          [] => acc
           hd::tl => f(hd, List.fold_right(f, tl, acc)) end in
       let List.append: ((()), ()) -> ()) =
        fun xs, ys -> List.fold_right(List.cons, xs, ys) in
       let List.concat: ((?)) -> (?) =
        fun xss -> List.fold_right(List.append, xss, []) in
       let List.mapi: ((Int, ?) -> ?, (?)) -> (?) =
        fun f, xs ->
         let go: ? -> ? = fun idx, xs ->
          case xs
            | [] => []
             hd::tl => f(idx, hd)::go(idx + 1, tl) end in
          go(0, xs) in
       let List.filteri: ((Int, ?) -> Bool, (?)) -> (?) =
        fun f, xs ->
         List.concat(List.mapi(
           fun i, x -> if f(i, x) then (x) else [], xs)) in
       # A todo has a description and a status #
       type Todo = (String, Bool) in
       # A description input buffer and a todo list #
       type Model = (String, (Todo)) in
       type Action =
        + AddTodo
        + RemoveTodo(Int)
         + ToggleTodo(Int)
         + UpdateBuffer(String) in
         ype Update = (Model, Action) -> Model in
       let Todo.eq: (Todo, Todo) -> Bool =
        fun (d1, s1), (d2, s2) ->
         d1 $== d2 && bool_eq(s1, s2) in
       let Model.eq: (Model, Model) -> Bool =
```

fun (b1, ts1), (b2, ts2) →
 b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr =

+ OnClick() -> Event) + OnMouseDown() -> Event)

+ OnInput(String -> Event)

+ Create(String, String)

+ Style((StyleAttri))



- Text window
- Vector Retrieval
- Repository-level



11

- Text window
- Vector Retrieval
- Repository-level

```
(go,idx, (descr: String, status: Bool)) ->
          Dive
            Style()
            Display("flex"),
            Gap("lem"))),
           OnClick(fun () -> go(ToggleTodo(idx)))),
             Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []),
             Div([], (Text(descr)1),
             Text(if status then "Completed" else "Pending")
       let todos_deck: (Action -> Event, (Todo)) -> Node =
        fun go, todos ->
          if (List.is_empty(todos))
          then Text("You're caught up")
          else Div(
            (Create("class", "todos")),
            (Text("todos:"))
           @ List.mapi(fun i,t ->todo_card(go,i,t), todos)) in
       let add_button: (Action -> Event) -> Node =
        fun go -> Div(t
         Style()
          Display("flex"),
           JustifyContent("center"),
           BackgroundColor("#986"),
           BorderRadius("0.3em"),
          Cursor("pointer"))).
         OnClick(fun () -> go(AddTodo))),
          (Text("Add Todo"))) in
       let buffer: (Action -> Event) -> Node =
        fun go -> Div(
          (TextInput((OnInput(fun s -> go(UpdateBuffer(s))))), []))) in
      let bool_eq: (Bool, Bool) -> Bool =
        fun a, b → a && b \/ !a && !b in
      let List.equal: (? -> Bool, (?), (?)) -> Bool =
       fun p, xs, ys →
         case xs, ys
          [], [] => true
           x::xs, y::ys ⇒ p(x, y) && List.equal(p, xs, ys)
           _ => false end in
       let List.cons: (?, (?)) -> (?) =
        fun x, xs -> x::xs in
Update the EmojiPainter app #
.et update: (Model, Action) -> Model
??
               -> 2 =
       fun f, xs, acc ->
        case xs
          [] => acc
           hd::tl => f(hd, List.fold_right(f, tl, acc)) end in
       let List.append: ((()), ()) -> ()) =
       fun xs, ys -> List.fold_right(List.cons, xs, ys) in
      let List.concat: ((?)) -> (?) =
       fun xss -> List.fold_right(List.append, xss, []) in
      let List.mapi: ((Int, ?) -> ?, (?)) -> (?) =
       fun f, xs ->
        let go: ? -> ? = fun idx, xs ->
          case xs
            [] => []
             hd::tl => f(idx, hd)::go(idx + 1, tl) end in
         go(0, xs) in
      let List.filteri: ((Int, ?) -> Bool, (?)) -> (?) =
        fun f, xs ->
         List.concat(List.mapi(
           fun i, x -> if f(i, x) then (x) else [], xs)) in
      # A todo has a description and a status #
      type Todo = (String, Bool) in
      # A description input buffer and a todo list #
      type Model = (String, (Todo)) in
      type Action =
        + AddTodo
        + RemoveTodo(Int)
        + ToggleTodo(Int)
        + UpdateBuffer(String) in
        ype Update = (Model, Action) -> Model in
      let Todo.eq: (Todo, Todo) -> Bool =
       fun (d1, s1), (d2, s2) ->
        d1 $== d2 && bool_eq(s1, s2) in
       let Model.eq: (Model, Model) -> Bool =
```

let Model.init: Model = ("", []) in
type Event = +Inject(String,(Model,Action)->Model, Action) in

b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

type Event = +inject(String,(Nodel,Action)->Model, Action) in type Attr = + OnClick(() -> Event)

+ OnMouseDown(() -> Event)

fun (b1, ts1), (b2, ts2) ->

+ OnInput(String -> Event)

+ Create(String, String)

+ Style((StyleAttri))



- Text window
- Vector Retrieval
- Repository-level





- Text window
- Vector Retrieval
- Repository-level

Div()
Style()
Display("flex"),
Gap("lem"))),
OnClick(fun () -> go(ToggleTodo(idx)))),
[
Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []

let todos\_deck: (Action -> Event, (Todo)) -> Node =
fun go, todos ->
if (List.is\_empty(todos))
then Text("You're caught up")
else Div(
 (Greate("class", "todos"))

- (Text("todos:"))
- @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in

@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in
let add\_button: (Action -> Event) -> Node =
fun go -> Div((
Style((
Display("flex"),
JustifyContent("center"),
Backgroundcl(""flex"),
Backgroundcl(""

let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div()
Style()

- Display ("flex"),
- Gap("lem"))),
- OnClick(fun () -> go(ToggleTodo(idx)))),

Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []), Div([], (Text(descr))).

Update the EmojiPainter app #

let update: (Model, Action) -> Model
??

### let todos\_deck: (Action -> Event, (Todo)) -> Node = fun go, todos ->

- if (List.is\_empty(todos))
- then Text("You're caught up")
- else Div(
- (Create("class", "todos")), (Text("todos:"))
- @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in

@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in
let add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style((
Display("flex")),

- JustifyContent ("center")
- BackgroundColor ("#986"

let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div()
Style()
Display("flex"),
Gap("lem"))),
OnClick(fun () -> go(ToggleTodo(idx)))),
(
Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))),

let todos\_deck: (Action -> Event, (Todol) -> Node =
fun go, todos ->
if (List.is\_empty(todos))
then Text("You're caught up")
else Div(
 (Create("class", "todos")),
 (Text("todos:"))
 @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in



14

- Text window
- Vector Retrieval
- Repository-level



OnClick(fun () -> go(ToggleTodo(idx)))),

let todos\_deck: (Action -> Event, (Todo)) -> Node =

Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []),

Gap("lem"))),

fun go, todos ->

else Div(

Div([], (Text(descr)

if (List.is\_empty(todos))
then Text("You're caught up")

(Create("class", "todos")), (Text("todos:")) @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in let todos\_deck: (Action -> Event, (Todo)) -> Node = fun go, todos -> if (List.is\_empty(todos)) then Text("You're caught up")

else Div(
 (Create("class", "todos")),
 (Text("todos:"))
 @ List.mapi(fun i,t =>todo\_card(go,i,t), todos)) in

let update: (Model, Action) -> Model =
??
in

Update the EmojiPainter app #



- Text window
- Vector Retrieval
- Repository-level

```
(go,idx, (descr: String, status: Bool)) ->
          Dive
            Style()
             Display("flex"),
            Gap("lem"))),
           OnClick(fun () -> go(ToggleTodo(idx)))),
             Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []),
             Div([], (Text(descr)1),
             Text(if status then "Completed" else "Pending")
       let todos_deck: (Action -> Event, (Todo)) -> Node =
        fun go, todos ->
          if (List.is_empty(todos))
          then Text("You're caught up")
          else Div(
            (Create("class", "todos")),
            (Text("todos:"))
           @ List.mapi(fun i,t ->todo_card(go,i,t), todos)) in
       let add_button: (Action -> Event) -> Node =
        fun go -> Div(t
         Style()
          Display("flex"),
           JustifyContent("center"),
           BackgroundColor("#986"),
           BorderRadius("0.3em"),
          Cursor("pointer"))),
         OnClick(fun () -> go(AddTodo))),
          (Text("Add Todo"))) in
       let buffer: (Action -> Event) -> Node =
        fun go -> Div(
          (TextInput((OnInput(fun s -> go(UpdateBuffer(s))))), []))) in
      let bool_eq: (Bool, Bool) -> Bool =
        fun a, b → a && b \/ !a && !b in
      let List.equal: (? -> Bool, (?), (?)) -> Bool =
       fun p, xs, ys →
         case xs, ys
          [], [] => true
           | x::xs, y::ys => p(x, y) && List.equal(p, xs, ys)
           _ => false end in
       let List.cons: (?, (?)) -> (?) =
        fun x, xs -> x::xs in
Update the EmojiPainter app #
.et update: (Model, Action) -> Model
??
               -> 2 =
       fun f, xs, acc ->
        case xs
          [] => acc
           hd::tl => f(hd, List.fold_right(f, tl, acc)) end in
       let List.append: ((()), ()) -> ()) =
       fun xs, ys -> List.fold_right(List.cons, xs, ys) in
      let List.concat: ((?)) -> (?) =
       fun xss -> List.fold_right(List.append, xss, []) in
      let List.mapi: ((Int, ?) -> ?, (?)) -> (?) =
       fun f, xs ->
        let go: ? -> ? = fun idx, xs ->
          case xs
            | [] => []
             hd::tl => f(idx, hd)::go(idx + 1, tl) end in
         go(0, xs) in
      let List.filteri: ((Int, ?) -> Bool, (?)) -> (?) =
        fun f, xs ->
         List.concat(List.mapi(
           fun i, x -> if f(i, x) then (x) else [], xs)) in
      # A todo has a description and a status #
      type Todo = (String, Bool) in
      # A description input buffer and a todo list #
      type Model = (String, (Todo)) in
      type Action =
        + AddTodo
        + RemoveTodo(Int)
        + ToggleTodo(Int)
        + UpdateBuffer(String) in
        ype Update = (Model, Action) -> Model in
      let Todo.eq: (Todo, Todo) -> Bool =
       fun (d1, s1), (d2, s2) ->
        d1 $== d2 && bool_eq(s1, s2) in
      let Model.eq: (Model, Model) -> Bool =
```

fun (b1, ts1), (b2, ts2) →
 b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr =

+ OnClick() -> Event) + OnMouseDown() -> Event)

+ OnInput(String -> Event)

+ Create(String, String)

+ Style((StyleAttri))



- Text window
- Vector Retrieval
- Repository-level

ananpa #Render main TODO view# #Should be a brown box with rounded corners and nice padding # #Labelled Hazel TODOs. Use comics sans# let view: (Action -> Event, Model) -> Node = fun go, (input: String, todos: (Todo1) -> Divi :Style:: BackgroundColor("#543"), BorderRadius("0.3em"), Color("white"), Display("flex"), FlexDirection ("column") Width("9.2em"), Gap("0.5em"), Padding("0.3em")1)1, Text("Hazel Todos"),

buffer(go),
add\_button(go),

todos\_deck(go, todos)1) in







17

- Text window
- Vector Retrieval
- Repository-level

### let bool\_eq: (Bool, Bool) → Bool = fun a, b → a && b \/ !a && !b in

let List.equal: (? -> Bool, (?), (?)) -> Bool =

un p, xs, ys ->

 $|\Box, \Box \Rightarrow tr$ 

let List.cons:  $(2, (2)) \rightarrow (2)$ 

# Update the EmojiPainter app #

### let update: (Model, Action) -> Model = ??)

in

let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, [) ii

go(0, xs) in

let List.filteri: ((Int, ?) -> Bool, (?)) -> (?) =
fun f, xs ->
List.concat(list.mani())

fun i, x  $\rightarrow$  if f(i, x) then (x) else [], xs)) in

### let todo\_card : (Action -> Event, Int, Todo) -> Node = fun (go,idx, (descr: String, status: Bool)) -> Divit Stylect Display("flex"), Gap("lem"))), OnClick(fun () -> go(ToggleTodo(idx)))), Checkbox((OnClick(fun () -> go(RemoveTodo((idx)))), []), Div([], (Text(descr)1), Text(if status then "Completed" else "Pending") 1) **in** let todos\_deck: (Action -> Event, (Todo)) -> Node = fun go, todos -> if (List.is\_empty(todos)) then Text("You're caught up") else Div( (Create("class", "todos")), (Text("todos:"))

@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in

let add\_button: (Action -> Event) -> Node =
fun go -> Div(t
Style(t
Display("flex"),
JustifyContent("center"),
BackgroundColor("#986"),
BorderRadius("0.3em"),
Cursor("pointer"))),
OnClick(fun () -> go(AddTodo))),
(Text("Add Todo"))) in

let buffer: (Action -> Event) -> Node =
fun go -> Div(

# A todo has a description and a status #
type Todo = (String, Bool) in
# A description input buffer and a todo list #
type Model = (String, (Todo)) in

### type Action =

- + AddTodo + RemoveTodo(Int)
- + ToggleTodo(Int)
- + UpdateBuffer(String) in

type Update = (Nodel, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
fun (d1, s1), (d2, s2) ->
d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool =
fun (b1, ts1), (b2, ts2) ->

tun (b1, ts1), (b2, ts2) ->
b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr =

- + OnClick() -> Event)
- + OnMouseDown(() -> Event)
- + OnInput(String -> Event) + Create(String, String)
- + Style((StyleAttr))





- Text window
- Vector Retrieval
- Repository-level



let todo\_card : (Action -> Event, Int, Todo) -> Node = fun (go,idx, (descr: String, status: Bool)) -> Divit Style() Display("flex"), Gap("lem"))), OnClick(fun () -> go(ToggleTodo(idx)))), Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []), Div([], (Text(descr)1), Text(if status then "Completed" else "Pending") ) in let todos\_deck: (Action -> Event, (Todo1) -> Node = fun go, todos → if (List.is\_empty(todos)) then Text("You're caught up") else Div( (Create("class", "todos")), (Text("todos:")) @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in let add\_button: (Action -> Event) -> Node = fun go -> Divet Stylect Display("flex"), JustifyContent("center"), BackgroundColor("#986"), BorderRadius("0.3em"), Cursor("pointer"))), OnClick(fun () -> go(AddTodo))), (Text("Add Todo"))) in let buffer: (Action -> Event) -> Node = fun go -> Div(

# Update the EmojiPainter app #
let update: (Model, Action) -> Model =
??



# OUR APPROACH Static Contextualization with Typed Holes



**Programmer:** Program sketch

LS: Static context of program hole



LLM: Suggested program hole filling









LS: Static error feedback on filled sketch





21























Language Model







LS: Static error feedback on filled sketch







Language Server

LS: Static error feedback on filled sketch

LLM: Corrected hole filling

LLM: Suggested program hole filling













let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div((
 Style((
 Display("flex"),
 Gap("lem"))),
 OnClick(fun () -> go(ToggleTodo(idx)))),
 ( Checkbox:(OnClick(fun () -> go(RemoveTodo(idx)))), []), Div([], (Text(descr))), Textif status then "Completed" else "Pending") )) in

let todos\_deck: (Action -> Event, (Todo:) -> Node =
fun go, todos ->
if (List.is\_empty(todos))
then Text("You're caught up")
else Div(
 (Create("Class", "todos")),
 (Text("todos"))
 @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in

let add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("center"),
BackgroundColor("#996"),
BorderRadius("0.3em"),
Cursor("pointer"))),
OnClick(fun () -> go(AddTodo))),
(Text("Add Todo"))) in

let buffer: (Action -> Event) -> Node =
fun go -> Div(
[],
 (TextInput((OnInput(fun s -> go(UpdateBuffer(s)))), []))) in

### let bool\_eq: (Bool, Bool) → Bool = fun a, b → a && b \/ !a && !b in

let List.equal: (? → Bool, (?), (?)) → Bool =
fun p, xs, ys →
case xs, ys
[ ], [] ⇒> true
[ x::xs, y::ys ⇒> p(x, y) && List.equal(p, xs, ys)
] \_=> false end in

let List.cons:  $(r, (r)) \rightarrow (r) = fun x, xs \rightarrow x::xs in$ 

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) =
fun f, xs ->
let go: ? -> ? = fun idx, xs ->
case xs
 [] => []
 [ hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

let List.filteri: ((Int, ?) -> Bool, (?)) -> (?) =
fun f, xs ->
List.concat(List.mapi(
fun i, x -> if f(i, x) then (x) else [], xs)) in

### # A todo has a description and a status type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Nodel = (String, (Todo)) in

type Action =
 + AddTodo
 + RemoveTodo(Int)
 + ToggleTodo(Int)
 + UpdateBuffer(String) in type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1, s1), (d2, s2) ->
 d1 \$== d2 && bool\_eq(s1, s2) in

let Hodel.eq: (Hodel, Model) -> Bool =
fun (b1, ts1), (b2, ts2) ->
b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String, Model,Action)->Model, Action) in type Attr = + OnClick(() -> Event) + OnKouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in eventsion

in
type Node =
 Div((Att), (Node))
 Text(String)
 Button((Att), (Node))
 Checkbox((Att), (Node))
 ColorInput((Att), (Node))
 DateInput((Att), (Node))
 NumberInput((Att), (Node))
 Radige((Att), (Node))
 TextInput((Att), (Node))
 timeInput((Att), (Node))
in

in type View = Model → Node in type Render = + Render(String, Model, View, Update) in

let bool\_eq: (Bool, Bool)  $\rightarrow$  Bool = fun a, b  $\rightarrow$  a && b \/ !a && !b in

let List.cons:  $({\mathfrak F}, ({\mathfrak F})) \rightarrow ({\mathfrak F}) =$ fun x, xs  $\rightarrow$  x::xs in let List.is\_empty: () -> Bool =
fun xs ->
 case xs
 [] => true
 [\_::\_ => false end in

# Update the EmojiPainter app #

fun xss -> List.fold\_right(List.append, xss, []) in



Codebase



let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div((
 Style((
 Display("flex"),
 Gap("lem")),
 OnClick(fun () -> go(ToggleTodo(idx)))),
 ( Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []), Div([], (Text(descr))), Text(if status then "Completed" else "Pending") )) in let todos\_deck: (Action -> Event, (Todo:) -> Node =
fun go, todos ->
if (List.is\_empty(todos))
then Text("You're caught up")
else Div(
 (Create("class", "todos")),
 (Text("todos:"))
 @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in

let add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("center"),
BackgroundColor("#986"),
BorderRadius("0.3em"),
Cursor("pointer")),
OnClick(fun () -> go(AddTodo))),
(Text("Add Todo")) in

### let bool\_eq: (Bool, Bool) -> Bool ( fun a, b → a && b \/ !a && !b in

let List.equal: (? → Bool, (?), (?)) → Bool =
fun p, xs, ys →
case xs, ys
| [], [] ⇒ true
| x::xs, y::ys ⇒ p(x, y) && List.equal(p, xs, ys)
| \_ ⇒ false end in

let List.cons:  $(r, (r)) \rightarrow (r) = fun x, xs \rightarrow x::xs in$ 

let List.fold\_right: ((?, ?)-> ?, (?), ?)-> ? =
fun f, xs, acc ->
case xs
 [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end in

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.filteri: ((Int, ?) -> Bool, (?) -> (?) =
fun f, xs ->
List.concat(List.mapi(
fun i, x -> if f(i, x) then (x) else [], xs)) in

### # A todo has a description and a status type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
fun (d1, s1), (d2, s2) ->
d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Nodel, Model) -> Bool = fun (b1, ts1), (b2, ts2) ->
 b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Nodel.init: Model = ("", []) in

type Event = +Inject(String, Model,Action)->Model, Action) in type Attr = + OnClick(() -> Event) + OnHouseOwn(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

in
type Node =
Div((Attr), (Node))
+ Div((Attr), (Node))
+ Gat(String)
+ Button((Attr), (Node))
+ Checkbox((Attr), (Node))
+ DateInput((Attr), (Node))
+ NumberInput((Attr), (Node))
+ Range((Attr), (Node))
+ TextInput((Attr), (Node))
+ TimeInput((Attr), (Node))
in

in
type View = Model -> Node in
type Render =
 + Render(String, Model, View, Upg)

let bool\_eq: (Bool, Bool) -> Bool =
fun a, b -> a && b \/ !a && !b in let List.equal: (? → Bool, (?), (?)) → Bool =
 fun p, xs, ys →
 case xs, ys
 [], [] ⇒ true
 [ x::xs, y::ys ⇒ p(x, y) && List.equal(p, xs
 [ \_ ⇒ false end in

let List.cons:  $(Y, (Y)) \rightarrow (Y) = fun x, xs \rightarrow x::xs in$ 

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, [) in



### **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ٠ in



let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div((
 Style((
 Display("flex"),
 Gap("lem")),
 OnClick(fun () -> go(ToggleTodo(idx)))),
 ( Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []), Div([], (Text(descr))), Text(if status then "Completed" else "Pending") )) in let todos\_deck: (Action -> Event, (Todo:) -> Node =
fun go, todos ->
if (List.is\_empty(todos))
then Text("You're caught up")
else Div(
 (Create("class", "todos")),
 (Text("todos:"))
 @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in

let add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("center"),
BackgroundColor("#986"),
BorderRadius("0.3em"),
Cursor("pointer")),
OnClick(fun () -> go(AddTodo))),
(Text("Add Todo")) in

### let bool\_eq: (Bool, Bool) -> Bool ( fun a, b → a && b \/ !a && !b in

let List.equal: (? → Bool, (?), (?)) → Bool =
fun p, xs, ys →
case xs, ys
| [], [] ⇒ true
| x::xs, y::ys ⇒ p(x, y) && List.equal(p, xs, ys)
| \_ ⇒ false end in

let List.cons:  $(r, (r)) \rightarrow (r) = fun x, xs \rightarrow x::xs in$ 

let List.fold\_right: ((?, ?)-> ?, (?), ?)-> ? =
fun f, xs, acc ->
case xs
 [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end in

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.filteri: ((Int, ?) -> Bool, (?) -> (?) =
fun f, xs ->
List.concat(List.mapi(
fun i, x -> if f(i, x) then (x) else [], xs)) in

### # A todo has a description and a status type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
fun (d1, s1), (d2, s2) ->
d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Nodel, Model) -> Bool = fun (b1, ts1), (b2, ts2) ->
 b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Nodel.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr = + OnClick(() -> Event) + OnMouseDown(() -> Event) + OnInjut(String -> Event) + Create(String, String) + Style((StyleAttr)) in

in
type Node =
Div((Attr), (Node))
+ Div((Attr), (Node))
+ Gat(String)
+ Button((Attr), (Node))
+ Checkbox((Attr), (Node))
+ DateInput((Attr), (Node))
+ NumberInput((Attr), (Node))
+ Range((Attr), (Node))
+ TextInput((Attr), (Node))
+ TimeInput((Attr), (Node))
in

in
type View = Model -> Node in
type Render =
 + Render(String, Model, View, Upg)

let bool\_eq: (Bool, Bool) -> Bool =
fun a, b -> a && b \/ !a && !b in let List.equal: (? → Bool, (?), (?)) → Bool =
 fun p, xs, ys →
 case xs, ys
 [], [] ⇒ true
 [ x::xs, y::ys ⇒ p(x, y) && List.equal(p, xs
 [ \_ ⇒ false end in

let List.cons:  $(Y, (Y)) \rightarrow (Y) = fun x, xs \rightarrow x::xs in$ 

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, [) in



### **Program Sketch:**

# Update the EmojiPainter app #

### let update: (Model, Action) -> Model = ?? • in



31

let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div((
 Style((
 Display("flex"),
 Gap("lem")),
 OnClick(fun () -> go(ToggleTodo(idx)))),
 ( Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), []), Div([], (Text(descr))), Text(if status then "Completed" else "Pending") )) in let todos\_deck: (Action -> Event, (Todo:) -> Node =
fun go, todos ->
if (List.is\_empty(todos))
then Text("You're caught up")
else Div(
 (Create("class", "todos")),
 (Text("todos:"))
 @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in

let add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("center"),
BackgroundColor("#986"),
BorderRadius("0.3em"),
Cursor("pointer")),
OnClick(fun () -> go(AddTodo))),
(Text("Add Todo")) in

let bool\_eq: (Bool, Bool) -> Bool = fun a, b -> a && b \/ !a && !b in

let List.equal: (? → Bool, (?), (?)) → Bool =
fun p, xs, ys →
case xs, ys
| [], [] ⇒ true
| x::xs, y::ys ⇒ p(x, y) && List.equal(p, xs, ys)
| \_ ⇒ false end in

let List.cons:  $(r, (r)) \rightarrow (r) = fun x, xs \rightarrow x::xs in$ 

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.filteri: ((Int, ?) -> Bool, (?) -> (?) =
fun f, xs ->
List.concat(List.mapi(
fun i, x -> if f(i, x) then (x) else [], xs)) in

# A todo has a description and a status
type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
fun (d1, s1), (d2, s2) ->
d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Nodel, Model) -> Bool = fun (b1, ts1), (b2, ts2) ->
 b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String, Model,Action)->Model, Action) in type Attr = + OnClick(() -> Event) + OnMouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

in type Node = Div(Attr), (Node)) + Text(String) + Button(Attr), (Node)) + Checkbox((Attr), (Node)) + Checkbox((Attr), (Node)) + DateInput((Attr), (Node)) + Range((Attr), (Node)) + Range((Attr), (Node)) + TextInput((Attr), (Node)) in

in
type View = Model -> Node in
type Render =



let List.append: (((), ()) -> ()) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

### **Program Sketch:**

# Update the EmojiPainter app #

### let update: (Model, Action) -> Model = 22 ٦n





Div([], (Text(descr))), Text(if status then "Completed" else "Pending") odos\_deck: (Action -> Event, (Todo)) -> Nod fun go, todos ->
 if (List.is\_empty(todos))
 then Text("You're caught up") else Div( (Create("class", "todos")), (Text"todos:")) @ List.mapi(fun i,t →todo\_card(go,i,t), todos)) in et add\_button: (Action -> Event) -> Node :
fun go -> Div((
Style(:
Display("flex"),
JustifyContent("center"),
BackgroundColor("4986"),
BorderRadius("0.3em"),
Cursor("moniter"))). Cursor("pointer")1), OnClick(fun () -> go(AddTodo)) (Text("Add Todo")1) in et buffer: (Action -> Event) -> Node

todo\_card : (Action -> Event, Int, Todo) -> Node =
in (go,idx, (descr: String, status: Bool)) ->
Div()

Checkbox((OnClick(fun () -> go(RemoveTodo((idx)))), []),

Gap("lem"))),
OnClick(fun () -> go(ToggleTodo(idx)))),

Style() Display("flex")

fun go -> Div C (TextInput((OnInput(fun s -> go(UpdateBuffer(s)))), [])))

### let bool\_eq: (Bool, Bool) -> Bool fun a, b → a && b \/ !a && !b in

let List.equal: (? -> Bool, (?), (?)) -> Bool 

let List.cons: (?, (?)) -> (?) ->
fun x, xs -> x::xs in 

let List.fold\_right: (( $\gamma, \gamma$ )  $\rightarrow \gamma, (\gamma), \gamma$ )  $\rightarrow \gamma = fun f, xs, acc \rightarrow$ 

case xs
 [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end in

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons,

tet List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, [))

et List.mapi: ((Int, ?) -> ?, (?)) -> (?) = fun f, xs ->

fun f, xs ->
 let go: y -> y = fun idx, xs ->
 case xs
 [] => []
 [hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

let List.filteri: ((Int, ?) → Bool, (?)) → (?) =
fun f, xs →
List.concat(List.mapi(
fun i, x → if f(i, x) then (x) else [], xs)) in

### # A todo has a description and a type Todo = (String, Bool) in

# A description input buffer and a todo list
type Model = (String, (Todo)) in

type Action =
+ AddTodo
+ RemoveTodo(Int)
+ ToggleTodo(Int)
+ UpdateBuffer(String) in

sype Update = (Model, Action) -> Model i let Todo.eq: (Todo, Todo) -> Bool :
 fun (d1 e1) (d2 e2) =>

d1 \$== d2 && bool\_eq(s1, s2) et Model.eq: (Model, Mode)

b1 \$== b2 && List.equal(Todo.eq, ts1, t

et Model.init: Model = ("", []) in type Event = +Inject(String.(Model.Action

type Event = +Inject(String, type Attr = + OnClick(() -> Event) + OnMouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) - Create(String, String)

Style((StyleAttr))



et List.append: (((?), (?)) -> (?)) = fun xs, ys -> List.fold\_right(List.cons, xs, ys) in t List.concat: (())) -> ()) = fun xss -> List.fold\_right(List.append, xss, []) in



Fillable by any expression of type (Model, Action) -> Model

> SelectEmoji : Emoji -> Actio model\_init : Model updateGrid : (Grid, Row, Col clearGrid : Grid -> Grid fillRowInGrid : (Grid, Row,

**EXP** ? Empty expression hole



update: (Model, Action) -> Model =



let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div(;
Style((
type Row = Int in
), pe Col = Int in OnClick(fun () -> go(ToggleTodo(idx)))) Checkbox(:OnClick(fun () → go(RemoveTodo(idx)))), []), Div([], :Text(descr)), Text(if status then "Completed" else "Pending") )) in type Grid = ((Emoji)) in fun go, todos -> if (List.is.empty(todos)) dear Total type Emoji = String in (Create("class", "todos") (Text("todos:"))
@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in let add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("center"),
BackgroundColor("#996"),
BorderRadius("0.3em"),
Cursor("pointer")),
OnClick(fun () -> go(AddTodo)),
(Text("Add Todo")) in

(TextInput(OnInput(fun s -> go(UpdateBuffer(s)))), []))) in

### type Model = ( Grid, Emoji, (Emoji) ) in

let List.equal: (? -> Bool, (?), (?)) -> Bool =
fun p, xs, ys ->
case xs, ys
 [], [] => true
 [ x:txs, y::ys => p(x, y) && List.equal(p, xs, ys)
 [ \_=> false end in

let List.cons: (?, (?)) -> (? type Action = + SelectEmoji(Emoji) + StampEmoji(Row, Col) + ClearGrid( + FillRow(Row) in

let List.append: (([]), (]) -> (]) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys)

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.mapi: ((Int,  $\gamma$ )  $\rightarrow \gamma$ , ( $\gamma$ ))  $\rightarrow$  ( $\gamma$ ) = tex list.map1: (int, i -> i, (i) -> i; =
fun f, xs ->
 let go: ? -> ? = fun idx, xs ->
 case xs
 [ [] => []
 [ hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

 $\begin{array}{l} \texttt{let List,filteri: ((Int, ?) \rightarrow Bool, (?)) \rightarrow (?) = \\ \texttt{fun } \texttt{f}, \texttt{xs} \rightarrow \\ \texttt{List,concat(List,mapi()) \\ \texttt{fun } \texttt{i}, \texttt{x} \rightarrow \texttt{if } \texttt{f}(\texttt{i}, \texttt{x}) \texttt{ then } (\texttt{x}) \texttt{ else } [], \texttt{xs})) \texttt{ in } \end{array}$ 

### # A todo has a description and a sta type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Nodel = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Nodel, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1, s1), (d2, s2) ->
 d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool bl \$== b2 && List.equal(Todo.eq, tsl, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr = + OnCLick(() -> Event) + OnMouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

ype Node = type Node = Div(iAttr), (Node)) Faxt(String) Button(iAttr),(Node)) + Checkbox(iAttr),(Node)) + ColorInput(iAttr),(Node)) + DateInput(iAttr),(Node)) + RumberInput(iAttr),(Node)) + TextInput(iAttr),(Node)) + TimEInput(iAttr),(Node)) n

type View = Model -> Node in
type Render =



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

### **Relevant Types:**

**Program Sketch:** 

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ? ?

### Fillable by any expression of type (Model, Action) -> Model



SelectEmoji : Emoji -> Actio model\_init : Model updateGrid : (Grid, Row, Co clearGrid : Grid -> Grid fillRowInGrid : (Grid, Row,



let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div(;
Style((
type Row = Int in
), vpe Col = Int in
OnClick(fun () -> go(ToggleTodo(idx)))) Checkbox(:OnClick(fun () -> go(RemoveTodo(idx))); []), Div([], (Text(descr))), Text(if status then "Completed" else "Pending") ) in type Grid = ((Emoji)) in fun go, todos -> if (List.is.empty(todos)) dear Total type Emoji = String in (Create("class", "todos") (Text("todos:"))
@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in let add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("center"),
BackgroundColor("#996"),
BorderRadius("0.3em"),
Cursor("pointer")),
OnClick(fun () -> go(AddTodo)),
(Text("Add Todo")) in

(TextInput(OnInput(fun s -> go(UpdateBuffer(s)))), []))) in

### type Model = ( Grid, Emoji, (Emoji) ) in

let List.equal: (? -> Bool, (?), (?)) -> Bool =
fun p, xs, ys ->
case xs, ys
 [], [] => true
 [ x:txs, y::ys => p(x, y) && List.equal(p, xs, ys)
 [ \_=> false end in

let List.cons: (?, (?)) -> (? type Action = + SelectEmoji(Emoji) + StampEmoji(Row, Col) + ClearGrid( + FillRow(Row) in

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) f

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) = tex list.map::(int, p -> p, (p) -> (p) =
fun f, xs ->
 let go: p -> p = fun idx, xs ->
 case xs
 [ [] => []
 [ hd::tl => f(idx, hd:::go(idx + 1, tl) end in
 go(0, xs) in

let List.filteri: ((Int, ?) → Bool, (?) → (?) =
fun f, xs →
List.concat(List.mapi(
fun i, x → if f(i, x) then (x) else [], xs)) in

### # A todo has a description and a stat type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Nodel = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
fun (d1, s1), (d2, s2) ->
d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool + bl \$== b2 && List.equal(Todo.eq, tsl, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr = + OnCLick(() -> Event) + OnMouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

ype Node = type Node =
 type Node =
 tork(Attri, (Node))
 text(String)
 Button((Attri, (Node))
 torktok(Attri, (Node))
 torktok(Attri, (Node))
 torktok(Attri, (Node))
 NumberInput((Attri, (Node))
 Range((Attri, (Node))
 torktnput((Attri, (Node))
 timeInput((Attri, (Node))
 torktnput((Attri, (Node))

cype View = Model -> Node in cype Render =



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

### **Relevant Types:**

**Program Sketch:** 

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ? ?

### Fillable by any expression of type (Model, Action) -> Model



SelectEmoji : Emoji -> Actio model\_init : Model updateGrid : (Grid, Row, Co clearGrid : Grid -> Grid fillRowInGrid : (Grid, Row,



let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div(;
Style((
type Row = Int in
), pe Col = Int in OnClick(fun () -> go(ToggleTodo(idx)))) Checkbox((OnClick(fun () → go(RemoveTodo(idx)))), []), Div([], (Text(descr))), Text(if status then "Completed" else "Pending") )) in type Grid = ((Emoji)) in fun go, todos -> if (List.is.empty(todos)) chem for (Dist.is.empty(todos)) type Emoji = String in (Create("class", "todos") (Text("todos:"))
@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in let add\_button: (Action -> Event) -> Node =
fun go -> Div(:
Style: :
Display("flex"),
JustifyContent:"conter"),
outifyContent:"conter";

JustityContent("Center"), BackgroundColor("#986"), BorderRadius("0.3em"), Cursor("pointer"))), OnClick(fun () -> go(AddTodo))) (Text("Add Todo"))) in 

(TextInput(OnInput(fun s -> go(UpdateBuffer(s)))), []))) in

let bool\_eq: (Bool, Bool) -> Bool fun a, b -> a && b \/ !a && !b in let List.equal: (? → Bool, (?), (?)) → Bool =
fun p, xs, ys →
case xs, ys
 [], [] ⇒ true
 [ x::xs, y::ys ⇒> p(x, y) && List.equal(p, xs, ys)
 [ \_= ⇒ false end in

let List.cons: (?, (?)) -> (? type Action = + SelectEmoji(Emoji) + StampEmoji(Row, Col) + ClearGrid( + FillRow(Row) in

case xs
[ ] => acc
[ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end in

let List.append: (([]), (]) -> (]) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys)

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) = tex list.map1: (int, i -> i, (i) -> i; =
fun f, xs ->
 let go: ? -> ? = fun idx, xs ->
 case xs
 [ [] => []
 [ hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

 $\begin{array}{l} \mbox{let List.filteri: ((Int, \gamma) \rightarrow Bool, (\gamma)) \rightarrow (\gamma) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List.concat(List.mapi() \\ \mbox{fun } i, x \rightarrow \mbox{if } f(i, x) \mbox{then } (x) \mbox{else } [], xs)) \mbox{in } \end{array}$ 

# A todo has a description and a sta type Todo = (String, Bool) in

# A description input buffer and a todo list :
type Model = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in

type Update = (Nodel, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1, s1), (d2, s2) ->
 d1 \$== d2 && bool\_eq(s1, s2) in

let Nodel.eq: (Nodel, Model) -> Bool b1 \$== b2 && List.equal(Todo.eq, ts1, ts2)

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr = + OnCLick(() -> Event) + OnMouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

ype Node =

ype View = Model -> Node in



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**

### type Model = ( Grid, Emoji, [Emoji] ) in

## **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = 23

### Fillable by any expression of type (Model, Action) -> Model



SelectEmoji : Emoji -> Actio model\_init : Model updateGrid : (Grid, Row, Col clearGrid : Grid -> Grid fillRowInGrid : (Grid, Row,


let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div(:
Style((
type Row = Int in
, OnClick(fun () -> go(ToggleTodo(idx))) Checkbox(:OnClick(fun () → go(RemoveTodo(idx)))), []), Div([], [Text(descr))), Text(if status then "Completed" else "Pending") ) in type Grid = ((Emoji)) in fun go, todos -> if (List.is.empty(todos)) chem for function type Emoji = String in (Create("class", "todos") (Text("todos:"))
@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in let add\_button: (Action -> Event) -> Node =
fun go -> Div(:
Style: :
Display("flex"),
JustifyContent:"conter"),
outifyContent:"conter"; JustityContent("Center"), BackgroundColor("#986"), BorderRadius("0.48em"), Cursor("pointer"))), OnClick(fun () → go(AddTodo)); (Text("Add Todo"))) in

(TextInput(OnInput(fun s -> go(UpdateBuffer(s)))), []))) in

let bool\_eq: (Bool, Bool) -> Bool fun a, b -> a && b \/ !a && !b in

let List.equal: (? → Bool, (?), (?)) → Bool =
fun p, xs, ys →
case xs, ys
 [], [] ⇒ true
 [ x::xs, y::ys ⇒> p(x, y) && List.equal(p, xs, ys)
 [ \_= ⇒ false end in

letList.cons: (?, (?)) -> (? type Action = + SelectEmoji(Emoji) + StampEmoji(Row, Col) + ClearGrid( + FillRow(Row) in

case xs
[ ] => acc
[ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end in

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys)

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) if

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) = tex list.map1: (int, i -> i, (i) -> i; =
fun f, xs ->
 let go: ? -> ? = fun idx, xs ->
 case xs
 [ [] => []
 [ hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

 $\begin{array}{l} \mbox{let List.filteri: ((Int, \gamma) \rightarrow Bool, (\gamma)) \rightarrow (\gamma) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List.concat(List.mapi() \\ \mbox{fun } i, x \rightarrow \mbox{if } f(i, x) \mbox{then } (x) \mbox{else } [], xs)) \mbox{in } \end{array}$ 

# A todo has a description and a sta type Todo = (String, Bool) in

# A description input buffer and a todo list :
type Model = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in

type Update = (Nodel, Action) -> Model in let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1, s1), (d2, s2) ->
 d1 \$== d2 && bool\_eq(s1, s2) in

let Nodel.eq: (Nodel, Model) -> Bool

b1 \$== b2 && List.equal(Todo.eq, ts1, ts2)

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr = + OnCLick(() -> Event) + OnMouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

ype Node =

sype Node = + Div(1Attr), (Node)) + Eut(String) = Button(1Attr), (Node)) + Checkbox(1Attr), (Node)) + Checkbox(1Attr), (Node)) + BateInput(1Attr), (Node)) + NumbeInput(1Attr), (Node)) + Range(1Attr), (Node)) + TextInput(1Attr), (Node)) = TimeInvi(1Attr), (Node))

TimeInput((Attr),(Node))

pe View = Model -> Node in



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**

#### type Model = ( Grid, Emoji, [Emoji] ) in

## **Program Sketch:**

# Update the EmojiPainter app #









Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), [] Div([], (Text(descr))), Text(if status then "Completed" else "Pending") type Grid = ((Emoji)) in
fungo, todos -> if (List.is\_empty(todos)) type Emoji = String in (Create:"class", "todos" @ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in et add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("conter"),
austifyContent("conter") JustifyContent("Center"), BackgroundColor("#986"), BorderRadius("0.3em"), Cursor("pointer")), OnClick(fun () -> go(AddTodo))) (Text("Add Todo"))) in let buffer: (Action -> Event) -> Node
fun go -> Div(

let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div()
Style()
Style()
type Row = Int in
,

OnClick(fun () -> go(ToggleTodo(idx))

(TextInput((OnInput(fun s -> go(UpdateBuffer(s)))), []))) in

let bool\_eq: (Bool, Bool) -> Bool fun a, b → a && b \/ !a && !b in

let List.equal: (? → Bool, (?), (?)) → Bool =
fun p, xs, ys →
case xs, ys
 [], [] ⇒ true
 | xi:xs, y::ys ⇒> p(x, y) && List.equal(p, xs, ys)
 | \_=> false end in let List.cons: (?, (?)) -> (?)
fun x, xs -> x::xs in

let List.fold\_right: (( $\gamma, \gamma$ )  $\rightarrow \gamma, (\gamma), \gamma$ )  $\rightarrow \gamma = fun f, xs, acc \rightarrow$ 

case xs
 [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end f

let List.append: (((), ()) -> ()) =
fun xs, ys -> List.fold\_right(List.cons,

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, [ let List.mapi: ((Int, ?) -> ?, (?)) -> (?) =

fun f, xs ->
let go: } -> ? = fun idx, xs ->
case xs
 [] => []
 [hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in fun f, xs ->

 $\begin{array}{l} \mbox{let List,filteri: ((Int, ?) \rightarrow Bool, (?)) \rightarrow (?) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List,concat(List,mapi() \\ \mbox{fun } i, x \rightarrow if \mbox{f(i, x) then } (x) \mbox{else [], xs)) in } \end{array}$ 

#### # A todo has a description and a : type Todo = (String, Bool) in

# A description input buffer and a todo list
type Model = (String, (Todol) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in

type Update = (Nodel, Action) -> Model i

let Todo.eq: (Todo, Todo) -> Bool :
 fun (d1, s1), (d2, s2) -> dl \$== d2 && bool\_eq(s1, s2)

let Nodel.eq: (Nodel, Model) -> Boo b1 \$== b2 && List.equal(Todo.eq, ts1, ts2

let Model.init: Model = ("", []) in sype Event = +Inject(String,(Nodel,Action)->Model, Action)

type Event = +Inject(String, type Attr = + OnClick(() -> Event) + OnNouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) - Create(String, String)

Style((StyleAttri))

NumberInput((Attr),(Note))
Radio((Attr),(Node))
Range((Attr),(Node))

TimeInput((Attr),(Node

View = Model -> Node in



let List.append: (([?]), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: (())) -> () = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**

#### type Action =

- + SelectEmoji(Emoji)
- + StampEmoji(Row, Col)
- + ClearCell(Row, Col)
- + ClearGrid
- + FillRow(Row) in

**type** Model = ( Grid, Emoji, [Emoji] ) **in** 

## **Program Sketch:**

# Update the EmojiPainter app #









let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div((
Style()
Style()
type Row = Int in
), OnClick(fun () -> go(ToggleTodo(idx)) Checkbox((OnClick(fun () -> go(RemoveTodo(idx)))), [] Div([], (Text(descr))), Text(if status then "Completed" else "Pending") type Grid = ((Emoji)) in
fungo, todos -> if (List.is\_empty(todos)) type Emoji = String in (Create:"class", "todos" (Text("todos:"))
@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in et add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("conter"),
austifyContent("conter") JustityContent ("Center"), BackgroundColor ("#986"), BorderRadius ("0.486"), Cursor ("pointer")11), OnClick (fun () -> go(AddTodo))) (Text("Add Todo")1) in let buffer: (Action -> Event) -> Node
fun go -> Div( (TextInput((OnInput(fun s -> go(UpdateBuffer(s)))), []))) in

let bool\_eq: (Bool, Bool) -> Bool fun a, b → a && b \/ !a && !b in

let List.equal: (? → Bool, (?), (?)) → Bool =
fun p, xs, ys →
case xs, ys
 [], [] ⇒ true
 | xi:xs, y::ys ⇒> p(x, y) && List.equal(p, xs, ys)
 | \_=> false end in let List.cons: (?, (?)) -> (?) =
fun x, xs -> x::xs in

let List.fold\_right: (( $\gamma, \gamma$ )  $\rightarrow \gamma, (\gamma), \gamma$ )  $\rightarrow \gamma = fun f, xs, acc \rightarrow$ 

case xs
 [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end f

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, >)

let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, [] let List.mapi: ((Int, ?) -> ?, (?)) -> (?) =

fun f, xs ->
 let go: ¿ -> ? = fun idx, xs ->
 case xs
 [] => []
 [hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in fun f, xs ->

 $\begin{array}{l} \mbox{let List,filteri: ((Int, ?) \rightarrow Bool, (?)) \rightarrow (?) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List,concat(List,mapi() \\ \mbox{fun } i, x \rightarrow if \mbox{f(i, x) then } (x) \mbox{else [], xs)) in } \end{array}$ 

#### # A todo has a description and a s type Todo = (String, Bool) in

# A description input buffer and a todo list
type Model = (String, (Todol) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Nodel, Action) -> Model i

let Todo.eq: (Todo, Todo) -> Bool :
 fun (d1, s1), (d2, s2) -> dl \$== d2 && bool\_eq(s1, s2)

let Nodel.eq: (Nodel, Model) -> Boo b1 \$== b2 && List.equal(Todo.eq, ts1, ts2

let Model.init: Model = ("", []) in sype Event = +Inject(String,(Nodel,Action)->Model, Action)

type Event = +Inject(String,(
type Attr =
 OnClick(() -> Event)
 OnClick(() -> Event)
 OnInput(String -> Event)
 Create(String, String)
 OnInput(String)

+ Style((StyleAttri)

NumberInput((Attr),(Node))
Radio((Attr),(Node))
Range((Attr),(Node))

TimeInput((Attr),(Node

View = Model -> Node in



let List.append: (([?]), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: (())) -> () = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**

#### type Action =

- + SelectEmoji(Emoji)
- + StampEmoji(Row, Col)
- + ClearCell(Row, Col)
- + ClearGrid
- + FillRow(Row) in

type Model =

Grid

Emoji, [Emoji] ) in

## **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ? ?

#### Fillable by any expression of type (Model, Action) -> Model







let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div()
Style()
Style()
type Row = Int in
, OnClick(fun () -> go(ToggleTodo(idx))) Checkbox((OnClick(fun () -> go(RemoveTodo((dx)))), []), Div([], (Text(descr))), Text(if status then "Completed" else "Pending") et todos\_deck: (Action  $\rightarrow$  Event, (Todo))  $\rightarrow$  Node fun go, todos ->
 if (List.is\_empty(todos)) type Emoji = String in Create("class", "todos" (Text("todos:"))
@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in et add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("conter"),
austifyContent("conter") JustifyContent("Center"), BackgroundColor("#986"), BorderRadius("0.48m"), Cursor("pointer"))), OnClick(fun () -> go(AddTodo))) (Text("Add Todo"))) in let buffer: (Action -> Event) -> Node
fun go -> Div( (TextInput((OnInput(fun s -> go(UpdateBuffer(s)))), []))) in

#### let bool\_eq: (Bool, Bool) -> Bool fun a, b → a && b \/ !a && !b in

let List.equal: () → Bool, (), () → Bool =
fun p, xs, ys →
case xs, ys
 [], [] ⇒ true
 [ x:txs, y::ys ⇒> p(x, y) && List.equal(p, xs, ys)
 [ \_=> false end in let List.cons: (?, (?)) -> (?)
fun x, xs -> x::xs in

let List.fold\_right: (( $\gamma, \gamma$ )  $\rightarrow \gamma, (\gamma), \gamma$ )  $\rightarrow \gamma = fun f, xs, acc \rightarrow$ case xs
 [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end f

let List.append: (((), ()) -> ()) =
fun xs, ys -> List.fold\_right(List.cons,

let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) = fun f, xs ->

fun f, xs ->
 let go: ¿ -> ? = fun idx, xs ->
 case xs
 [] => []
 [hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

 $\begin{array}{l} \mbox{let List,filteri: ((Int, ?) \rightarrow Bool, (?)) \rightarrow (?) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List,concat(List,mapi() \\ \mbox{fun } i, x \rightarrow if \mbox{f(i, x) then } (x) \mbox{else [], xs)) in } \end{array}$ 

# A todo has a description and a s
type Todo = (String, Bool) in

# A description input buffer and a todo list
type Model = (String, (Todol) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in

type Update = (Nodel, Action) -> Model i

let Todo.eq: (Todo, Todo) -> Bool :
 fun (d1, s1), (d2, s2) -> dl \$== d2 && bool\_eq(s1, s2) i

let Nodel.eq: (Nodel, Model) -> Boo b1 \$== b2 && List.equal(Todo.eq, ts1, ts2

let Model.init: Model = ("", []) in sype Event = +Inject(String,(Nodel,Action)->Model, Action)

type Event = +Inject(String,(
type Attr =
 OnClick(() -> Event)
 OnClick(() -> Event)
 OnInput(String -> Event)
 Create(String, String)
 OnInput(String)

+ Style((StyleAttri)

NumberInput((Attr),(Node))
Radio((Attr),(Node))
Range((Attr),(Node))

TimeInput((Attr),(Node

View = Model -> Node in



let List.append: (([?]), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: (())) -> () = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**

#### type Grid = [[Emoji]] in

#### type Action =

- + SelectEmoji(Emoji)
- + StampEmoji(Row, Col)
- + ClearCell(Row, Col)
- + ClearGrid
- + FillRow(Row) in

type Model =

Grid

Emoji, [Emoji] ) in



# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ? ?

#### Fillable by any expression of type (Model, Action) -> Model







let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div()
Style()
Style()
type Row = Int in
, OnClick(fun () -> go(ToggleTodo(idx))) Checkbox((OnClick(fun () -> go(RemoveTodo((dx)))), []), Div([], (Text(descr))), Text(if status then "Completed" else "Pending") et todos\_deck: (Action  $\rightarrow$  Event, (Todo))  $\rightarrow$  Node fun go, todos ->
 if (List.is\_empty(todos)) type Emoji = String in [Create("class", "todos") (Text("todos:"))
@ List.mapi(fun i,t ->todo\_card(go,i,t), todos)) in et add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("conter"),
austifyContent("conter") JustityContent("Center"), BackgroundColor("#986"), BorderRadius("0.3em"), Cursor("pointer"))), OnClick(fun () → go(AddTodo))) (Text("Add Todo"))) in let buffer: (Action -> Event) -> Node
fun go -> Div( (TextInput((OnInput(fun s -> go(UpdateBuffer(s)))), []))) in

#### let bool\_eq: (Bool, Bool) -> Bool fun a, b → a && b \/ !a && !b in

let List.equal: () → Bool, (), () → Bool =
fun p, xs, ys →
case xs, ys
 [], [] ⇒ true
 [ x:txs, y::ys ⇒> p(x, y) && List.equal(p, xs, ys)
 [ \_=> false end in let List.cons: (?, (?)) -> (?)
fun x, xs -> x::xs in

let List.fold\_right: (( $\gamma, \gamma$ )  $\rightarrow \gamma, (\gamma), \gamma$ )  $\rightarrow \gamma = fun f, xs, acc \rightarrow$ case xs
 [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end f

let List.append: (((), ()) -> ()) =
fun xs, ys -> List.fold\_right(List.cons,

let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) = fun f, xs ->

fun f, xs ->
let go: } -> ? = fun idx, xs ->
case xs
 [] => []
 [hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

 $\begin{array}{l} \mbox{let List,filteri: ((Int, ?) \rightarrow Bool, (?)) \rightarrow (?) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List,concat(List,mapi() \\ \mbox{fun } i, x \rightarrow if \mbox{f(i, x) then } (x) \mbox{else [], xs)) in } \end{array}$ 

#### # A todo has a description and a s type Todo = (String, Bool) in

# A description input buffer and a todo list
type Model = (String, (Todol) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Nodel, Action) -> Model i

let Todo.eq: (Todo, Todo) -> Bool :
 fun (d1, s1), (d2, s2) -> dl \$== d2 && bool\_eq(s1, s2) i

let Nodel.eq: (Nodel, Model) -> Boo b1 \$== b2 && List.equal(Todo.eq, ts1, ts2

let Model.init: Model = ("", []) in sype Event = +Inject(String,(Nodel,Action)->Model, Action)

type Event = +Inject(String,(
type Attr =
 OnClick(() -> Event)
 OnClick(() -> Event)
 OnInput(String -> Event)
 Create(String, String)
 OnInput(String)

+ Style((StyleAttri)

+ NumberInput((Attr),(Node)
+ Radio((Attr),(Node))
+ Range((Attr),(Node))

TimeInput((Attr),(Node

View = Model -> Node in



let List.append: (([?]), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: (())) -> () = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**

#### type Grid = [[Emoji]] in

#### type Action =

- + SelectEmoji(Emoji)
- + StampEmoji(Row, Col)
- + ClearCell(Row, Col)
- + ClearGrid
- + FillRow(Row) in

type Model = ( Grid, Emoji

#### **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ? ?

#### Fillable by any expression of type (Model, Action) -> Model







let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div()
Style()
Style()
type Row = Int in
, OnClick(fun () -> go(ToggleTodo(idx))) Checkbox((OnClick(fun () -> go(RemoveTodo((dx)))), []), Div([], (Text(descr)) Text(if status then "Completed" else "Pending") todos\_deck: (Action -> Event, (Todo)) -> Node fun go, todos ->
if (List.is\_empty(todos))
then Text("You're caught up") else Div( (Create("class", "todos")), (Text"todos:")) @ List.mapi(fun i,t →todo\_card(go,i,t), todos)) in let add\_button: (Action -> Event) -> Node =
fun go -> Div(t
Style(:
Display("flex"),
JustifyContent("center"),
BackgroundColor("#986"),
BorderRadius("0.3em"),
Cursor("pointer"))),
OnClick(fun () -> go(AddTodo))),
(Text("Add Todo"))) in et buffer: (Action -> Event) -> Node fun go -> Div C (TextInput(OnInput(fun s -> go(UpdateBuffer(s)))), []))) f let bool\_eq: (Bool, Bool) -> Bool fun a, b → a && b \/ !a && !b in let List.cons: (?, (?)) -> (?)
fun x, xs -> x::xs in let List.fold\_right: (( $\gamma, \gamma$ )  $\rightarrow \gamma, (\gamma), \gamma$ )  $\rightarrow \gamma = fun f, xs, acc \rightarrow$ case xs
 [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end f let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, let List.concat: (())) -> () =
 fun xss -> List.fold\_right(List.append, xss, [] let List.mapi: ((Int, ?) -> ?, (?)) -> (?) = fun f, xs ->
 let go: ¿ -> ? = fun idx, xs ->
 case xs
 [] => []
 [hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in fun f, xs ->  $\begin{array}{l} \mbox{let List,filteri: ((Int, ?) \rightarrow Bool, (?)) \rightarrow (?) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List,concat(List,mapi() \\ \mbox{fun } i, x \rightarrow if \mbox{f(i, x) then } (x) \mbox{else [], xs)) in } \end{array}$ # A todo has a description and a s
type Todo = (String, Bool) in # A description input buffer and a todo list
type Model = (String, (Todol) in type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Nodel, Action) -> Model i let Todo.eq: (Todo, Todo) -> Bool :
 fun (d1, s1), (d2, s2) -> dl \$== d2 && bool\_eq(s1, s2) i let Nodel.eq: (Nodel, Model) -> Boo b1 \$== b2 && List.equal(Todo.eq, ts1, ts2 let Model.init: Model = ("", []) in sype Event = +Inject(String,(Nodel,Action)->Model, Action) type Event = +Inject(String,(
type Attr =
 OnClick(() -> Event)
 OnClick(() -> Event)
 OnInput(String -> Event)
 Create(String, String)
 OnInput(String) + Style((StyleAttri) + NumberInput((Attr),(Node)
+ Radio((Attr),(Node))
+ Range((Attr),(Node)) TimeInput((Attr),(Node View = Model -> Node in

**Relevant Types:** 

type Emoji = String in

type Grid = [[Emoji]] in

#### type Action =

- + SelectEmoji(Emoji)
- + StampEmoji(Row, Col)
- + ClearCell(Row, Col)
- + ClearGrid
- + FillRow(Row) in

type Model = ( Grid, Emoji





# Update the EmojiPainter app #

let update: (Model, Action) -> Model =
 ??

let List.append: (((?), (?)) -> (?)) =
 fun xs, ys -> List.fold\_right(List.cons, xs, ys) in
 let List.concat: ((?)) -> (?) =
 fun xss -> List.fold\_right(List.append, xss, []) in

## Fillable by any expression of type (Model, Action) -> Model





let todo\_card : (Action -> Event, Int, Todo) -> Node =
fun (go,idx, (descr: String, status: Bool)) ->
Div((
Style()
Style()
type Row = Int in
), OnClick(fun () -> go(ToggleTodo(idx))) Checkbox(:OnClick(fun () → go(RemoveTodo(idx)))), []), Div([], [Text(descr))), Text(if status then "Completed" else "Pending") ) in t todos\_deck: (Action -> Event, (Todo)) -> Node = fun go, todos ->
 if (List.is\_empty(todos))
 then Text("You're caught up") else Div( (Create("class", "todos")), (Text"todos:")) @ List.mapi(fun i,t →todo\_card(go,i,t), todos)) in et add\_button: (Action -> Event) -> Node =
fun go -> Div()
Style()
Display("flex"),
JustifyContent("conter"),
austifyContent("conter") BackgroundColor("#986"), BorderRadius("0.3em"), Cursor("pointer")1), OnClick(fun () -> go(AddTodo)) (Text("Add Todo")1) in let buffer: (Action -> Event) -> Node
fun go -> Div( (TextInput(OnInput(fun s -> go(UpdateBuffer(s)))), []))) i

#### let bool\_eq: (Bool, Bool) -> Bool fun a, b -> a && b \/ !a && !b in

let List.equal: (? → Bool, (?), (?)) → Bool =
fun p, xs, ys →
case xs, ys
 [], [] ⇒ True
 [ x::xs, y::ys ⇒> p(x, y) && List.equal(p, xs, ys)
 [ \_=> false end in let List.cons: (?, (?)) -> (?) =
fun x, xs -> x::xs in

let List.is\_empty: (?) → Bool =
 fun xs →
 case xs
 | [] => true
 | \_::\_ => false end in

case xs
 [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end in let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys)

let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) =

 $\begin{array}{l} \mbox{let List.filteri: ((Int, \gamma) \rightarrow Bool, (\gamma)) \rightarrow (\gamma) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List.concat(List.mapi() \\ \mbox{fun } i, x \rightarrow \mbox{if } f(i, x) \mbox{then } (x) \mbox{else } [], xs)) \mbox{in } \end{array}$ 

#### # A todo has a description and a sta type Todo = (String, Bool) in

# A description input buffer and a todo list :
type Model = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1, s1), (d2, s2) -> dl \$== d2 && bool\_eq(s1, s2) in

let Nodel.eq: (Nodel, Model) -> Bool b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) i

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Nodel,Action)->Model, Action) in type Attr = + OnClick(() -> Event) + OnNouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Strike(String, String)

+ Style((StyleAttri)

NumberInput((Attr),(Node))
Radio((Attr),(Node))
Range((Attr),(Node))

TimeInput((Attr),(Node

View = Model -> Node in



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: ((?)) -> (?) = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



type Model = ( Grid, Emoji, [Emoji] ) in

#### **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ??



Fillable by any expression of type (Model, Action) -> Model

SelectEmoji : Emoji -> Actio model\_init : Model updateGrid : (Grid, Row, Col clearGrid : Grid -> Grid fillRowInGrid : (Grid, Row, **EXP** ? Empty expression hole









let List.cons: (?, (?)) -> (?)
fun x, xs -> x::xs in

let List.fold\_right: (( $\gamma, \gamma$ )  $\rightarrow \gamma, (\gamma), \gamma$ )  $\rightarrow \gamma = fun f, xs, acc \rightarrow$ case xs
[ [] => acc
[ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end i

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, y)

let List.concat: (()) -> () =
 fun xss -> List.fold\_right(List.append, xss, [])

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) = fun f, xs →
 let go: } → } = fun idx, xs →
 case xs
 [] => []
 hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in fun f, xs ->

let List.filteri: ((Int, ?) → Bool, (?)) → (?) =
fun f, xs →
List.concat(List.mapi(
fun i, x → if f(i, x) then (x) else [], xs)) in

#### # A todo has a description and a s type Todo = (String, Bool) in

# A description input buffer and a todo list
type Model = (String, (Todol) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Nodel, Action) -> Model i

let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1 e1) (d2 e2) => d1 \$== d2 && bool\_eq(s1, s2) 1

let Model.eq: (Model, Model) -> Bool b1 \$== b2 && List.equal(Todo.eq, ts1, ts2)

let Model.init: Model = ("", []) in sype Event = +Inject(String,(Model,Action)->Model, Action

type Event = +Inject(String, type Attr = + OnClick(() -> Event) + OnNouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) - Create(String, String)

+ Style((StyleAttr))

NumberInput((Attr),(No Radio((Attr),(Node)) Range((Attr),(Node))

TimeInput((Attr),(Node

/iew = Model -> Node in



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: ((?)) -> (?) = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



type Model = ( Grid, Emoji, [Emoji] → in

## **Relevant Headers:**

Type of hole = (Model, Action) -> Model

#### **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ? ?



Fillable by any expression of type (Model, Action) -> Model

SelectEmoji : Emoji -> Ac model\_init : Model updateGrid : (Grid, Row, clearGrid : Grid -> Grid fillRowInGrid : (Grid, Ro EXP ? Empty expression



С	t	i		
	С	0		
C	w	,		
١	h	0	le	•





#### NumberInput((Attr),(No Radio((Attr),(Node)) Range((Attr),(Node)) TimeInput((Attr),(Node /iew = Model -> Node in



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: ((?)) -> (?) = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



type Model = ( Grid, Emoji, [Emoji] → in

## **Relevant Headers:**

Type of hole = (Model, Action) -> Model

Target types =

#### **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ? ?



Fillable by any expression of type (Model, Action) -> Model

SelectEmoji : Emoji -> Ac model\_init : Model updateGrid : (Grid, Row, clearGrid : Grid -> Grid fillRowInGrid : (Grid, Ro EXP ? Empty expression



С	t	i		
	С	0		
C	w	,		
١	h	0	le	•







let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: ((?)) -> (?) = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**

type Emoji = String in	
<pre>type Grid = [[Emoji]] in</pre>	
<pre>type Action =     + SelectEmoji(Emoji)     + StampEmoji(Row, Col)     + ClearCell(Row, Col)     + ClearGrid     + FillRow(Row) in</pre>	
<b>type</b> Model = ( Grid, Emo-	ji, [Emoji] ) ir

## **Relevant Headers:**

Type of hole = (Model, Action) -> Model

Target types = (Model, Action) -> Model

#### **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ? ?



Fillable by any expression of type (Model, Action) -> Model

SelectEmoji : Emoji -> Actio model\_init : Model updateGrid : (Grid, Row, Col clearGrid : Grid -> Grid fillRowInGrid : (Grid, Row, **EXP** (?) Empty expression hole











let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: ((?)) -> (?) = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



## **Relevant Headers:**

Type of hole = (Model, Action) -> Model

Target types = (Model, Action) -> Model, Model

## **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ? ?



Fillable by any expression of type (Model, Action) -> Model

SelectEmoji : Emoji -> Ac model\_init : Model updateGrid : (Grid, Row, clearGrid : Grid -> Grid fillRowInGrid : (Grid, Ro EXP ? Empty expression



С	t	i		
	С	0		
C	w	,		
١	h	0	le	





let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: ((?)) -> (?) = fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



## **Relevant Headers:**

Type of hole = Model Action) -> Model

Target types = (Model, Action) -> Model, Model

## **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ??



Fillable by any expression of type (Model, Action) -> Model

SelectEmoji : Emoji -> Actio model\_init : Model updateGrid : (Grid, Row, Col clearGrid : Grid -> Grid fillRowInGrid : (Grid, Row, EXP (?) Empty expression hole













<pre>type Grid = [[Emoji]] in type Action =     + SelectEmoji(Emoji)     + StampEmoji(Row, Col)     + ClearCell(Row, Col)     + ClearGrid     + FillRow(Row) in</pre>	type Emoji = String in
<pre>type Action =     + SelectEmoji(Emoji)     + StampEmoji(Row, Col)     + ClearCell(Row, Col)     + ClearGrid     + FillRow(Row) in</pre>	<pre>type Grid = [[Emoji]] in</pre>
	<pre>type Action =     + SelectEmoji(Emoji)     + StampEmoji(Row, Col)     + ClearCell(Row, Col)     + ClearGrid     + FillRow(Row) in</pre>

type Model = ( Grid, Emoji, [Emoji] ) in

## **Relevant Headers:**

Type of hole = ((String, (Grid, Emoji, [Emoji])), Action) -> (Grid, Emoji, [Emoji])

Target types = (Model, Action) -> Model, Model



# Update the EmojiPainter app #













ype Node = 

pe View = Model -> Node in



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



type Model = ( Grid, Emoji, [Emoji] ) in

## **Relevant Headers:**

Type of hole = ((String, (Grid, Emoji, [Emoji]))

Target types = (Model, Action) -> Model, Model,



# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ??

, Action) -> (Grid, Emoji, [Emo	ji])	Fillable by any expression of (Model, Action) -> M
Grid, Emoji, [Emoji]	Sel mod upd cle fil	LectEmoji : Emoji -> A del_init : Model dateGrid : (Grid, Row, earGrid : Grid -> Grid LlRowInGrid : (Grid, R EXP 2 Empty expression



· -

n.

+.0









let List.mapi: ((Int, ?) -> ?, (?)) -> (?) =
fun f, xs ->
 let go: ? -> ? = fun idx, xs ->
 case xs
 [] => []
 | hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

#### # A todo has a description and a sta type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Nodel, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1, s1), (d2, s2) ->
 d1 \$== d2 && bool\_eq(s1, s2) in

let Nodel.eq: (Model, Model) -> Bool = b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr = + OnClick(() -> Event) + OnHouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

in
type.Node =
 Div(iAttr), (Node))
 Text(String)
 Button:(Attr), (Node))
 Checkbox((Attr), (Node))
 ColorInput:(Attr), (Node))
 DateInput(iAttr), (Node))
 Range(iAttr), (Node))
 Range(iAttr), (Node))
 TextInput(iAttr), (Node))
 TimeInput(iAttr), (Node))
in

type View = Model -> Node in type Render =





let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**

<pre>type Grid = [[Emoji]] in type Action =     + SelectEmoji(Emoji)     + StampEmoji(Row, Col)     + ClearCell(Row, Col)     + ClearGrid     + FillRow(Row) in</pre>	<b>type</b> Emoji = String <b>in</b>
<pre>type Action =   + SelectEmoji(Emoji)   + StampEmoji(Row, Col)   + ClearCell(Row, Col)   + ClearGrid   + FillRow(Row) in</pre>	<pre>type Grid = [[Emoji]] in</pre>
	<pre>type Action =   + SelectEmoji(Emoji)   + StampEmoji(Row, Col)   + ClearCell(Row, Col)   + ClearGrid   + FillRow(Row) in</pre>

#### type Model = ( Grid, Emoji, [Emoji] ) in

## **Relevant Headers:**

Type of hole = ((String, (Grid, Emoji, [Emoji]))

Target types = (Model, Action) -> Model

## **Program Sketch:**

# Update the EmojiPainter app #

				$(\tilde{g}, r) =$ $\tilde{g}, r) =$	Aline 4/4 EEE A B B B B B B B B B B B B B B B B B
, Action) -> (Grid, Emoji, [Emoj	i])	Fillable by (Model,	any express Action)	sion ( ->	of type Model
Grid, Emoji, [Emoji]	Sel mod upd cle fil	LectEmoji del_init dateGrid earGrid : LlRowInGr	: Emoji : Model : (Grid, Grid -> id : (Gr Empty exp	-> Row Gro id,	Actio w, Col id Row, ion hole









let List.is\_empty: (?) -> Bool =
fun xs ->
case xs
 | ] => true
 | \_::\_ => false end in

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) i

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) = let List.map1: ((int, ?) -> ?, (?)) -> (?) =
fun f, xs ->
let go: ? -> ? = fun idx, xs ->
case xs
 [ [] => []
 [ hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

#### # A todo has a description and a sta type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Nodel = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Nodel, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1, s1), (d2, s2) ->
 d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool = b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr = + OnClick(() -> Event) + OnToput(String -> Event) + OrToput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

in
type.Node =
 Div((Attr), (Node))
 Text(String)
 Button((Attr), (Node))
 Checkbox((Attr), (Node))
 ColorInput((Attr), (Node))
 DateInput((Attr), (Node))
 NumberInput((Attr), (Node))
 Range((Attr), (Node))
 TextInput((Attr), (Node))
 TimeInput((Attr), (Node))
n

ype View = Model -> Node in



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



type Model = ( Grid, Emoji, [Emoji] ) in

## **Relevant Headers:**

Type of hole = ((String, (Grid, Emoji, [Emoji]))

Model

Target types = (Model, Action) -> Model

#### **Program Sketch:**

# Update the EmojiPainter app #

				$     \begin{array}{c}             \hat{x}_{1}^{2} \left( x_{1}^{2} \right) = x_{1}^{2} \\             \hat{x}_{1}^{2} \left( x_{1}^{2} \right) = x_{1}^{2} \\$	4-4/2 ==== ((0 + 1) + 4) 
, Action) -> (Grid, Emoji, [Emo	ji])	Fillable by (Model,	any express Action)	sion o ->	of t M
Grid, Emoji, [Emoji]	Sel mod upd cle fil	ectEmoji del_init dateGrid earGrid : lRowInGr	i : Emoji : Model : (Grid, Grid -> id : (Gr Empty exp	-> Row Gri id, ressi	A d R or











let List.is\_empty: (?) → Bool =
 fun xs →
 case xs
 | [] => true
 | \_::\_ => false end in

let List.append: (([]), (]) -> (]) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys)

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

let List.mapi: ((Int, ?) -> ?, (?)) -> (?) = tex list.map1: (int, i -> i, (i) -> i; =
fun f, xs ->
 let go: ? -> ? = fun idx, xs ->
 case xs
 [ [] => []
 [ hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

 $\begin{array}{l} \mbox{let List.filteri: ((Int, \ensuremath{\gamma}) \rightarrow \mbox{Bool}, \ensuremath{\langle\gamma\rangle}) \rightarrow \ensuremath{\langle\gamma\rangle} = $$ fun f, xs \rightarrow $$ List.concat(List.mapi($$ fun i, x \rightarrow \ensuremath{if} f(i, x) \mbox{then } (x) \mbox{else [], xs)} \ensuremath{in} i \ensuremath{n} \ensu$ 

#### # A todo has a description and a sta type Todo = (String, Bool) in

# A description input buffer and a todo list :
type Model = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Nodel, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1, s1), (d2, s2) -> d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool + bl \$== b2 && List.equal(Todo.eq, tsl, ts2) in

let Nodel.init: Model = ("", []) in

type Event = +Inject(String,(Nodel,Action)->Model, Action) in type Attr = + OnClick(() -> Event) + OnMouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

ype Node =

pe View = Model -> Node in





let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in et List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



type Model = ( Grid, Emoji, [Emoji] ) in

## **Relevant Headers:**

Type of hole = ((String, (Grid, Emoji, [Emoji]))

, Model

Target types = (Model, Action) -> Model

#### let model\_init: Model = ... in



# Update the EmojiPainter app #

					$e^{1/2} (GF * 3) A_{44} F$ $e^{1/2} (GF * $
, Action) -> (Grid, Emoji, [Emo	ji])	Fillable by (Model,	any express Action)	sion c	of 1 M
Grid, Emoji, [Emoji]	Sel mod upd cle fil	LectEmoji del_init dateGrid earGrid : LlRowInGr	: Emoji : Model : (Grid, Grid -> id : (Gr Empty exp	-> Rov Gr- id,	A v, id R





С	t	i		
	С	0		
C	w	,		
١	h	0	le	•





let List.cons: (?, (?)) -> (?) =
fun x, xs -> x::xs in

let List.is\_empty: (?) -> Bool =
fun xs ->
case xs
 | ] => true
 | \_::\_ => false end in

let List.append: (([]), (]) -> (]) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys)

let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in let List.mapi: ((Int, ?) -> ?, (?)) -> (?) =

tex list.map1: (int, i -> i, (i) -> i; =
fun f, xs ->
 let go: ? -> ? = fun idx, xs ->
 case xs
 [ [] => []
 [ hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

 $\begin{array}{l} \mbox{let List.filteri: ((Int, \gamma) \rightarrow Bool, (\gamma)) \rightarrow (\gamma) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List.concat(List.mapi( \\ \mbox{fun } i, x \rightarrow ) \mbox{if } f(i, x) \mbox{then } (x) \mbox{else } [], xs)) \mbox{in } \end{array}$ 

#### # A todo has a description and a stat type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Nodel = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Nodel, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
 fun (d1, s1), (d2, s2) ->
 d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool = b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Nodel,Action)->Model, Action) in type Attr = + OnCl(ck(() -> Event) + OnHouseDown(() -> Event) + OnIput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

in
type.Node =
Div((Attr), (Node))
Text(String)
Button((Attr), (Node))
Checkbox((Attr), (Node))
Checkbox((Attr), (Node))
DateInput((Attr), (Node))
NumberInput((Attr), (Node))
Range((Attr), (Node))
TextInput((Attr), (Node))
TextInput((Attr), (Node))
N

ype View = Model -> Node in



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



type Model = ( Grid, Emoji, [Emoji] ) in

## **Relevant Headers:**

Type of hole = ((String, (Grid, Emoji, [Emoji]))

Target types = (Model, Action) -> Model, Model,

#### let model\_init: Model = ... in



# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ??

	F	illable by	any express	ion of type
, Action) -> (Grid, Emoji, [Emo	ji]) (	Model,	Action)	-> Model
Grid Emoji, [Emoji]	Sele	ctEmoji	i : Emoji	-> Actio
•	mode	l_init	: Model	
	upda	teGrid	: (Grid,	Row, Col
	clea	rGrid :	Grid ->	Grid
	fill	RowInGr	rid : (Gri	id, Row,
	ΓΕΧ		Empty expr	ression hole



·H

n.

t.0







#### let List.equal: (? -> Bool, (?), (?)) -> Bool = fun p, xs, ys -> case xs, ys [], [] => true [ x::xs, y::ys => p(x, y) && List.equal(p, xs, ys) [ \_ => false end in let List.cons: (?, (?)) -> (?) = fun x, xs -> x::xs in

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

#### # A todo has a description and a stat type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, (Todo)) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
fun (d1, s1), (d2, s2) ->
d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool = fun (b1, ts1), (b2, ts2) ->
 b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in type Attr = + OnClick(() -> Event) + OnInput() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

in type Node = Div(Attr), (Node)) + Text(String) + Button(Attr), (Node)) + Checkbox((Attr), (Node)) + Checkbox((Attr), (Node)) + DateInput((Attr), (Node)) + Range((Attr), (Node)) + Range((Attr), (Node)) + TextInput((Attr), (Node)) in

in
type View = Model -> Node in
type Render =
 + Render(String, Model )



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

## **Relevant Types:**



<pre>+ ClearGrid + FillRow(Row) in</pre>	
type Model = ( Grid, Emoji, [Emoji] ) in	
Relevant Headers: Type of hole = ((String, (Grid, Emoji, [Emoji])), Action) -> (Grid, Emoji, [Emo	<pre>Fillable by any expression of type (Model, Action) -&gt; Model</pre>
<pre>Target types = (Model, Action) -&gt; Model, Model, Grid, Emoji, [Emoji]</pre>	SelectEmoji : Emoji -> Actio
<pre>let model_init: Model = in</pre>	<pre>model_init : Model</pre>
<pre>let updateGrid: (Grid, Row, Col, Emoji) -&gt; Grid = in.</pre>	updateGrid : (Grid, Row, Col
Let clearGrid: Grid -> Grid = … in Let fillPowInGrid: (Grid Row Emoji) -> Grid -  in	clearGrid : Grid -> Grid
Let IIIIKOWINGIIU. (GIIU, KOW, ENOJI) -> GIIU = III	<pre>fillRowInGrid : (Grid, Row,</pre>
	<b>Γ EXP</b> ? Empty expression hole



# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ??



· +.+

n.

+.0







#### let List.equal: (? -> Bool, (?), (?)) -> Bool = fun p, xs, ys -> case xs, ys [], [] => true [ x::xs, y::ys => p(x, y) && List.equal(p, xs, ys) [ \_ => false end in let List.cons: $(r, (r)) \rightarrow (r) = fun x, xs \rightarrow x::xs in$

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in

 $\begin{array}{l} \mbox{let List.filteri: ((Int, ?) \rightarrow Bool, (?)) \rightarrow (?) = \\ \mbox{fun } f, xs \rightarrow \\ \mbox{List.concat:List.mapi(} \\ \mbox{fun } i, x \rightarrow \mbox{if } f(i, x) \mbox{then } (x) \mbox{else } [], xs)) \mbox{in } \end{array}$ 

#### # A todo has a description and a statu type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Nodel = (String, (Todol) in

type Action =
 AddTodo
 RemoveTodo(Int)
 ToggleTodo(Int)
 UpdateBuffer(String) in

type Update = (Nodel, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
fun (d1, s1), (d2, s2) ->
d1 \$== d2 && bool\_eq(s1, s2) in

let Model.eq: (Nodel, Model) -> Bool = fun (b1, ts1), (b2, ts2) ->
 b1 \$== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Nodel,Action)->Model, Action) in type Attr = + OnCl(dk(() -> Event) + OnMouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Style((StyleAttr)) in

in type Node = Div(Attr), (Node)) + Text(String) + Button(Attr), (Node)) + Checkbox((Attr), (Node)) + Checkbox((Attr), (Node)) + DateInput((Attr), (Node)) + Range((Attr), (Node)) + Range((Attr), (Node)) + TextInput((Attr), (Node)) in

in
type View = Model -> Node in
type Render =
 + Render(String, Node)



let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons, xs, ys) in let List.concat: (()) -> () =
fun xss -> List.fold\_right(List.append, xss, []) in



<pre>type Emoji = String in</pre>
<pre>type Grid = [[Emoji]] in</pre>
<pre>type Action =   + SelectEmoji(Emoji)   + StampEmoji(Row, Col)   + ClearCell(Row, Col)   + ClearGrid   + FillRow(Row) in</pre>

<pre>+ StampEmoji(Row, Cot) + ClearCell(Row, Col) + ClearGrid + FillRow(Row) in</pre>	
<pre>type Model = ( Grid, Emoji, (Emoji) ) in Relevant Headers: Fype of hole = ((String, (Grid, Emoji, [Emoji])), Action) -&gt; (Grid, Emoji, [Emoji]))</pre>	<pre>Fillable by any expression of type (Model, Action) -&gt; Model</pre>
Target types = (Model, Action) -> Model, Model, Grid, Emoji, [Emoji]	SelectEmoji : Emoji -> Actio
<pre>let model_init: Model = in</pre>	<pre>model_init : Model</pre>
<pre>let updateGrid: (Grid, Row, Col, Emoji) -&gt; Grid = in.</pre>	updateGrid : (Grid, Row, Col
<pre>let clearGrid: Grid -&gt; Grid = in let fillPowIpGrid: (Grid Row Emoji) -&gt; Grid - in</pre>	clearGrid : Grid -> Grid
rec rilikowinoriu. (oriu, kow, choji) -> oriu in	fillRowInGrid : (Grid, Row,
	<b>Γ EXP</b> ? Empty expression hole



# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ??



· +.\*

n.

+.0







#### **Expected Type:**

Fillable by any expression of type (Model, Action) -> Model

#### **Relevant Types:**

type Emoji = String in

type Grid = [[Emoji]] in

type Action =

- + SelectEmoji(Emoji)
- + StampEmoji(Row, Col)
- + ClearCell(Row, Col)
- + ClearGrid
- + FillRow(Row) in

**type** Model = ( Grid, Emoji, [Emoji] ) **in** 

#### **Relevant Headers:**

- let model\_init: Model = ... in
- let updateGrid: (Grid, Row, Col, Emoji) -> Grid = ... in
- let clearGrid: Grid -> Grid = ... in
- let fillRowInGrid: (Grid, Row, Emoji) -> Grid = ... in

## **Program Sketch:**

# Update the EmojiPainter app #

in

let update: (Model, Action) -> Model =
 ??

let List.append: (((?), (?)) → (?)) =
 fun xs, ys → List.fold\_right(List.cons, xs, ys) in
let List.concat: ((?)) → (?) =
 fun xss → List.fold\_right(List.append, xss, []) in

Grid = ... in id = ... in





vpe Update = (Model, Action) -> Model et Todo.eq: (Todo, Todo) -> Bool d1 \$== d2 && bool\_eq(s1, s2)

b1 \$== b2 && List.equal(Todo.eq, ts1, t

ype Event = +Inject(String,(Model,A

ype Event = \*inject(String, ype Attr = + OnClick(() -> Event) + OnNouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) + Stylec(StyleAttr)) Style((StyleAttri))





et List.append: (((?), (?)) -> (?)) = fun xs, ys -> List.fold\_right(List.cons, xs, ys) in t List.concat: ((?)) -> (?) =
fun xss -> List.fold\_right(List.append, xss, []) in

## **Expected Type:**

Fillable by any expression of type (Model, Action) -> Model

#### **Relevant Types:**

type Emoji = String in

type Grid = [[Emoji]] in

type Action =

- + SelectEmoji(Emoji)
- + StampEmoji(Row, Col)
- + ClearCell(Row, Col)
- + ClearGrid
- + FillRow(Row) in

**type** Model = ( Grid, Emoji, [Emoji] ) **in** 

#### **Relevant Headers:**

- let model\_init: Model = ... in
- let updateGrid: (Grid, Row, Col, Emoji) ->
- let clearGrid: Grid -> Grid = ... in
- let fillRowInGrid: (Grid, Row, Emoji) -> Gri

## **Program Sketch:**

# Update the EmojiPainter app #

let update: (Model, Action) -> Model = ??

#### **ChatLSP:** The hole **??** expects a value having a type consistent with (Model, Action) -> Model. # The following type definitions are likely relevant: type Emoji = String in type Action = SelectEmoji(Emoji) + StampEmoji(Row, Col.. type Grid = [[Emoji]] in type Model = (Grid, Emoji, [Emoji]) in Consider these values relevant to the expected type # let model\_init: Model = ... in let updateGrid: (Grid, Row, Col, Emoji) -> Grid = ... in let clearGrid: Grid -> Grid = ... in let fillRowInGrid: (Grid, Row, Emoji) -> Grid = ... in # Complete the following program sketch: # Update the EmojiPainter app # let update: (Model, Action) -> Model = ??









let List.is\_empty: (?) -> Bool fun xs ->
 case xs
 | [] => true
 | \_::\_ => false end in case xs
 [ [] => acc
 [ hd::tl => f(hd, List.fold\_right(f, tl, acc)) end d

let List.append: (((?), (?)) -> (?)) =
fun xs, ys -> List.fold\_right(List.cons,

et List.mapi: ((Int, ?) -> ?, (?)) -> (?) =

fun f, xs →
 let go: Y → Y = fun idx, xs →
 case xs
 [] => []
 | hd::tl => f(idx, hd)::go(idx + 1, tl) end in
 go(0, xs) in

et List.filteri: ((Int,  $\gamma$ )  $\rightarrow$  Bool, ( $\gamma$ ))  $\rightarrow$  ( $\gamma$ ) = fun f, xs -> fun f, xs ->
List.concat(List.mapi(
 fun i, x -> if f(i, x) then (x) else [], xs)) in

# A todo has a description and a type Todo = (String, Bool) in

# A description input buffer and a todo lis
type Nodel = (String, (Todo:) in

type Action =
 + AddTodo
 + RemoveTodo(Int) + ToggleTodo(Int) + UpdateBuffer(String) in

sype Update = (Model, Action) -> Model -

et Todo.eq: (Todo, Todo) -> Bool ( d1 \$== d2 && bool\_eq(s1, s2)

et Model.eq: (Model, Model) -> Bo b1 \$== b2 && List.equal(Todo.eq, ts1, t

et Model.init: Model = ("", []) in ype Event = +Inject(String,(Model,Action)->Mode

ype Event = \*inject(String, ype Attr = + OnClick(() -> Event) + OnNouseDown(() -> Event) + OnInput(String -> Event) + Create(String, String) - Create(String, String)

Style((StyleAttr))



et List.append: (((?), (?)) -> (?)) = fun xs, ys -> List.fold\_right(List.cons, xs, ys) in t List.concat: (())) -> ()) = fun xss -> List.fold\_right(List.append, xss, []) in

```
let update: (Model, Action) -> Model =
  fun model, action ->
    case action
     SelectEmoji(emoji) =>
     let (_, _, emoji_palette) = model in
      (updateGrid(model, -1, -1, emoji), emoji, emoji_palette)
      StampEmoji((row, col)) =>
     let (grid, selected_emoji, emoji_palette) = model in
      (updateGrid(grid, row, col, selected_emoji),
      selected_emoji, emoji_palette>
      ClearCell((row, col)) =>
     let (grid, selected_emoji, emoji_palette) = model in
      (updateGrid(grid, row, col, ""), selected_emoji,
       emoji_palette)
      ClearGrid =>
     let (_, selected_emoji, emoji_palette) = model in
      (clearGrid(grid), selected_emoji, emoji_palette)
      FillRow(row) =>
     let (grid, selected_emoji, emoji_palette) = model in
      (fillRowInGrid(grid, row, selected_emoji), selected_emoji,
       emoji_palette)
    end in
```

**ChatLSP:** The hole **??** expects a value having # a type consistent with (Model, Action) -> Model. # The following type definitions are likely relevant: # type Emoji = String in type Action = SelectEmoji(Emoji) + StampEmoji(Row, Col... type Grid = [[Emoji]] in type Model = (Grid, Emoji, [Emoji]) in # Consider these values relevant to the expected type # let model\_init: Model = ... in let updateGrid: (Grid, Row, Col, Emoji) -> Grid = ... in **let** clearGrid: Grid -> Grid = ... in let fillRowInGrid: (Grid, Row, Emoji) -> Grid = ... in # Complete the following program sketch: # Update the EmojiPainter app # let update: (Model, Action) -> Model = ??







## Evaluation Design





## Evaluation **Problems with existing evals**

- (e.g. HumanEval, EvalPlus, LiveCodeBench)
- (e.g. RepoEval, RepoBench, CrossCodeEval)

Standard evals generally consist of single-file tasks which are low-context

• Repo-level evals tend to be language exclusive (often focusing on dynamic langs like Python), and focus on syntactic rather than semantic (unit) testing





## **MVUBench Model-View-Update Web Apps**

- New suite of small-but-full applications developed from scratch
- Minimal external dependencies (easy to port across languages)
- Many domain-specific data types defined across different files
  - **Todo** (*TO*): Maintains a list of tasks
  - **Room Booking (BO)**: Manages a room booking schedule
  - Emoji Painter (EM): Paints emoji stamps on a small canvas
  - Playlist Manager (PL): Manages a music playlist
  - Password Strength Checker (PA): Rates a password via a dynamic set of properties



# **Evaluation Results**







## **Results: Hazel + GPT-4**

Retr	Retrieval Error			<b>Average % Tests Pass</b>					
Types	Headers	Rounds	BO	EM	PA	PL	TO	ALL	
20	no	no	0	0	0	0	0	0	
no	no	yes	0	0	0	0	0	0	
	VOS	no	1	0	0	0	0	0	
	yes	yes	0	0	0	0	0	0	
			26				15	12	

Results for guided completion (20 trials per, temperature 0.6)



Relevant values but no types



## **Results: Hazel + GPT-4**

Retri	ieval	Error	Average % Tests Pass			ass		
Types	Headers	Rounds	BO	EM	PA	PL	TO	ALL
200	no	no	0	0	0	0	0	0
no	no	yes	0	0	0	0	0	0
	yes	no	1	0	0	0	0	0
		yes	0	0	0	0	0	0
	20	no	26	0	7	10	15	12
yes	no	yes	31	1	40	53	47	34
			28				35	48

Results for guided completion (20 trials per, temperature 0.6)

```
let update: (Model, Action) -> Model =
 fun model, action ->
   let (grid, selected_emoji, emoji_palette), (action) =
     model, action in
   case action
     SelectEmoji(emoji) => (grid, emoji, emoji_palette)
     StampEmoji((row, col)) =>
   let grid = Grid.stamp(row, col, selected_emoji, grid) in
    (grid, selected_emoji, emoji_palette)
    ClearCell((row, col)) =>
   let grid = Grid.clear(row, col, grid) in
   (grid, selected_emoji, emoji_palette)
     ClearGrid => (Grid.empty_grid(), selected_emoji, emoji_palette)
```

Type definitions but no values





## **Results: Hazel + GPT-4**

Retrieval		Error	Ave	ass				
Types	Headers	Rounds	BO	EM	PA	PL	ТО	ALL
200	no	no	0	0	0	0	0	0
no	no	yes	0	0	0	0	0	0
	VOO	no	1	0	0	0	0	0
	yes	yes	0	0	0	0	0	0
	no	no	26	0	7	10	15	12
yes	ПО	yes	31	1	40	53	47	34
	NOO	no	28	79	29	68	35	48
	yes	yes	41	95	93	81	71	76

Results for guided completion (20 trials per, temperature 0.6)

```
let update: (Model, Action) -> Model =
 fun model, action ->
   let (grid, current_emoji, emoji_list) = model in
   case action
     SelectEmoji(emoji) => (grid, emoji, emoji_list)
     StampEmoji((row, col)) =>
    (updateGrid(grid, row, col, current_emoji), current_emoji, emoji_list)
    ClearCell((row, col)) =>
    (updateGrid(grid,row, col, ""), current_emoji, emoji_list)
    ClearGrid => (clearGrid(grid), current_emoji, emoji_list)
     FillRow(row) =>
    (fillRowInGrid(grid, row, current_emoji), current_emoji, emoji_list)
   end in
```

Both values and types (correct implementation)





## **Results: Hazel + GPT-4 vs Baselines**

Retr	ieval	Error	Ave	Average % Tests Pass			ass	
Types	Headers	Rounds	BO	ΕM	PA	PL	ТО	ALL
							47	34
		no	28	79	29	68	35	48
yes	yes	yes	41	95	93	81	71	76

Results for guided completion (20 trials per, temperature 0.6)

Vector	Error	Average % Tests Pass					
Retrieval	Rounds	BO	EM	PA	PL	ТО	ALL
yes	no	0	0	0	15	33	10
	yes	0	3	0	20	67	18



## **Results: Hazel + GPT-4 vs Baselines**

Retr	ieval	Error	Ave	Average % Tests Pass			ass	
Types	Headers	Rounds	BO	ΕM	PA	PL	ТО	ALL
							47	34
		no	28	79	29	68	35	48
yes	yes	yes	41	95	93	81	71	76

Results for guided completion (20 trials per, temperature 0.6)

```
1 # SNIPPET 1 #
2 , ts2) in
3
4 let Model.init: Model = ("", []) in
5
6 type Action =
7 + AddTodo
8 + RemoveTodo(Int)
9 + ToggleTodo(Int)
10 + UpdateBuffer(String) in
```

Vector	Error Average % Tests Pass						
Retrieval	Rounds	BO	EM	PA	PL	TO	ALL
yes	no	0	0	0	15	33	10
	yes	0	3	0	20	67	18



## **Results: Hazel + GPT-4 vs Baselines**

Retr	ieval	Error	Ave	Average % Tests Pass			ass	
Types	Headers	Rounds	BO	ΕM	PA	PL	ТО	ALL
							47	34
		no	28	79	29	68	35	48
yes	yes	yes	41	95	93	81	71	76

Results for guided completion (20 trials per, temperature 0.6)

Exhaustive	Error	Average % Tests Pass					
Retrieval	Rounds	BO	EM	PA	PL	ТО	ALL
yes	no	9	90	61	43	69	54
	yes	43	100	91	81	80	79



## **Results: Typescript + GPT-4**

Retrieval		Error	Average % Tests Pass						Retr	Error	Average % Tests Pass						
Types	Headers	Rounds	BO	EM	PA	PL	ТО	ALL	Types	Headers	Rounds	BO	EM	PA	PL	TO	ALL
no	no	no	0	0	0	0	0	0		20	no	0	0	0	0	0	0
		yes	0	0	0	0	0	0	no	ПО	yes	0	0	0	0	0	0
	yes	no	0	0	0	0	0	0		yes	no	1	0	0	0	0	0
		yes	0	4	0	3	0	1			yes	0	0	0	0	0	0
yes	no	no	46	26	43	85	66	53		no	no	26	0	7	10	15	12
		yes	69	39	46	93	66	63	yes		yes	31	1	40	53	47	34
	yes	no	47	95	83	64	75	73			no	28	79	29	68	35	48
		yes	58	99	85	86	75	80		yes	yes	41	95	93	81	71	76

#### TypeScript GPT-4

Hazel GPT-4



## **Results: Typescript + GPT4**

Retrieval		Error	Average % Tests Pass						Retr	Error	Average % Tests Pass						
Types	Headers	Rounds	BO	EM	PA	PL	ТО	ALL	Types	Headers	Rounds	BO	EM	PA	PL	ТО	ALL
no	no	no	0	0	0	0	0	0		no	no	0	0	0	0	0	0
		yes	0	0	0	0	0	0	no		yes	0	0	0	0	0	0
	yes	no	0	0	0	0	0	0		yes	no	1	0	0	0	0	0
		yes	0	4	0	3	0	1			yes	0	0	0	0	0	0
yes	no	no	46	26	43	85	66	53			no	26	0	7	10	15	12
		yes	69	39	46	93	66	63	yes	no	yes	31	1	40	53	47	34
	yes	no	47	95	83	64	75	73			no	28	79	29	68	35	48
		yes	58	99	85	86	75	80		yes	yes	41	95	93	81	71	76

TypeScript GPT-4

 See paper for more results, including re-running the GPT-4 experiments using the open source StarCoder2-15B LLM

Hazel GPT-4

# **ChatLSP**2

#### **ChatLSP API Methods** 5.1

- (1) **aiTutorial**: A constant (lexical-context-independent) method for low resource languages (like Hazel) to specify a textual tutorial intended for LLMs having robust support for in-context learning. For high resource languages, the default implementation will simply return a string stating which language is in use.
- (2) **expectedType**: Returns a string specifying the expected type at the cursor, if available
- (3) **retrieveRelevantTypes**: Returns a string containing type definitions that may be relevant at the cursor location
- (4) **retrieveRelevantHeaders**: Returns a string containing headers that may be relevant at the cursor location
- (5) **errorReport**: Returns an error report that can be used to determine if an error round is needed, and if so, how the feedback should be presented to the LLM.




# **ChatLSP**??

### **ChatLSP API Methods** 5.1

- (1) **aiTutorial**: A constant (lexical-context-independent) method for low resource languages (like Hazel) to specify a textual tutorial intended for LLMs having robust support for in-context learning. For high resource languages, the default implementation will simply return a string stating which language is in use.
- (2) **expectedType**: Returns a string specifying the expected type at the cursor, if available
- (3) **retrieveRelevantTypes**: Returns a string containing type definitions that may be relevant at the cursor location
- (4) **retrieveRelevantHeaders**: Returns a string containing headers that may be relevant at the cursor location
- (5) **errorReport**: Returns an error report that can be used to determine if an error round is needed, and if so, how the feedback should be presented to the LLM.



**Talk to Jacob VSCode Extension** 



# **TypeScript Static Contextualization**,



### **Related Work**

**Repository-level:** 

RepoCoder (Zhang et al. EMNLP23) CoCoMIC (Ding et al. LREC-COLING 2024) Repo-Level Prompt Gen (Shrivastava et al. ICML23)

Programming (Chakraborty et al. ICSE2024) (with emphasis on) **Semantic Context:** STALL+ (Liu et al. 2024) RLCoder (Wang et al. 2024) **Program Repair / Error Correction**: The Fact Better Context Makes Better Code Language Selection Problem in LLM-Based Program Repair Models (Pei et al. AAAI23) (Parasaram et al. 2024) CodeTrek (Pashakhanloo et al. ICLR22) Repair is nearly generation (Joshi et al. AAAI'23) Dehallucinator (Eghbali et al. 2024) Copiloting the Copilots And in industry: (Wei et al, ESEC/FSE 2023) Sourcegraph Cody, Cursor, Zed, Aider IDECoder (Li et al. LLM4Code24)

AutoCodeRover (Zhang et al. 2024) Private-library-oriented code generation with LLMs (Zan et al. 2023)

**Contextualizing Proofs:** Towards Neural Synthesis for SMT-Assisted Proof-Oriented







74



## Thanks for Listening!

### **Conclusion & Future Directions**

- LLMs need IDEs, too: Giving models access to a full TUI (Text User Interface) version of Hazel; fine-tuning models on Hazel user edit actions histories
- semantic edit actions (leveraging explicit programming strategies ala LaToza)

### Future of Programming Lab: fplab.mplse.org



 (In)human factors: When considering IDE features for LLMs, we've found it useful to consider how they would play out for humans. Where does this framing work/fail?

Scaffolding and refining complex code changes using type-driven development and

andrewblinn.com

