Programming Languages and Techniques (CIS120)

Lecture 20
March 18, 2019

GUI: Events & State
Chapter 18
Announcements

• Please fill out the mid-semester survey
  – Posted on Piazza
  – Should only take a couple minutes to complete
  – We take your feedback very seriously – help improve the remainder of the course!

• HW05: GUI programming
  – Due: next Tuesday (March 25) at 11:59:59pm
  – Graded (mostly) manually
    • Submission checks for compilation, few auto tests
    • Only LAST submission will be graded
  – This project is challenging:
    • IT IS NOW TOO LATE TO START EARLY!
Stateful Widgets

How can widgets react to events?
A stateful label Widget

```ocaml
let label (s: string) : widget =
  let r = { contents = s } in
  { repaint = (fun (g: Gctx.gctx) ->
    Gctx.draw_string g (0,0) r.contents);
    handle = (fun _ _ -> ());
    size = (fun () -> Gctx.text_size r.contents)
  }
```

- The label object can make its string mutable. The “methods” can refer to this mutable string.
- But how can we change this string in response to an event?
A stateful label Widget

```ml
type label_controller = { set_label: string -> unit }

let label (s: string) : widget * label_controller =
let r = { contents = s } in

{ repaint = (fun (g: Gctx.gctx) ->
    Gctx.draw_string g (0,0) r.contents);
  handle = (fun _ _ -> ());
  size = (fun () -> Gctx.text_size r.contents)
}

' { set_label = fun (s: string) -> r.contents <- s }
```

- A **controller** gives access to the shared state.
  - Here, the `label_controller` object returned by `label` provides a way to set the label string
Event Listeners

How to react to events in a modular way?

See notifierdemo.ml for the rest of the lecture!
How much did this lecture help?

- Not at all
- Somewhat, but need to think about it some more
- Totally clear now
- Already understood it perfectly
Listeners

(* Performs an action upon receiving a mouse click. *)

let mouseclick_listener (action: unit -> unit) : event_listener =
  fun (g: Gctx.gctx) (e: Gctx.event) ->
    if Gctx.event_type e = Gctx.MouseDown
    then action ()
Handling multiple event types

• **Problem:** *Widgets may want to react to many different sorts of events*

• **Example:** Button
  – button click: changes the state of the paint program and button label
  – mouse movement: tooltip? highlight?
  – key press: provide keyboard access to the button functionality?

• **These reactions should be independent**
  – Each sort of event handled by a different *event listener* (i.e. a first-class function)
  – Reactive widgets may have *several* listeners to handle a triggered event
  – Listeners react in sequence, all have a chance to see the event

• **Solution:** notifier
Listeners and Notifiers Pictorially

User clicks, generating event $e$

| Hello | World |

Widget tree

On the screen
Notifiers

• A **notifier** is a container widget that adds event listeners to a node in the widget hierarchy
  – Note: this way of structuring event listeners is based on Java’s Swing Library design (we use Swing terminology).

• **Event listeners** “eavesdrop” on the events flowing through the notifier
  – The event listeners are stored in a list
  – They react in order
  – Then the event is passed down to the child widget

• Event listeners can be added by using a notifier_controller
Notifiers and Notifier Controllers

```ocaml
type notifier_controller =
  { add_listener : event_listener -> unit }

let notifier (w: widget) : widget * notifier_controller =
  let listeners = { contents = [] } in
  { repaint = w.repaint;
    size = w.size
    handle =
      (fun (g: Gctx.gctx) (e: Gctx.event) ->
        List.iter (fun h -> h g e) listeners.contents;
        w.handle g e);
  },
  { add_event_listener =
    fun (newl: event_listener) ->
      listeners.contents <-
        newl :: listeners.contents
  }
```

The notifier_controller allows new listeners to be added to the list.

Loop through the list of listeners, allowing each one to process the event. Then pass the event to the child.
A button widget is just a label wrapped in a notifier.

Add a mouseclick_listener to the button using the notifier_controller.

(For aesthetic purposes, we could also put a border around the label widget.)

```ml
let button (s: string) : widget
    * label_controller
    * notifier_controller =

    let (w, lc) = label s in
    let (w', nc) = notifier w in
    (w', lc, nc)
```

(* A text button *)
lightbulb demo

Clicking here makes the "lightbulb" turn on and changes label text.

Clicking again makes it turn back off.
Typechecking Revisited

How does OCaml* typecheck your code?

*Historical aside: the algorithm we are about to see is known as the Damas-Hindley-Milner type inference algorithm. Turing Award winner Robin Milner was, among other things, the inventor of "ML" (for "meta language"), from which OCaml gets its "ml".
OCaml Typechecking Errors

```
let length (q: 'a queue) : int =
  let rec loop (qn : 'a qnode option) (acc: int) : int = int
  | None -> int
  | Some n -> loop n.next (1 + acc)

let empty () = []
```

```
let rec mem k mem =
  let begin map |
    | [] -> |
    | (k,v) |
    | if key |
    | (key = k) || (mem key rest)
  end

;; run_test "mem test" (fun () ->
  mem "b" ["a",3]; ("b",4)]
```
How do you determine the type of an expression?

1. Recursively determine the types of all of the sub-expressions
   - Some expressions have “obvious” types:
     3 : int    “foo” : string    true : bool
   - Identifiers have the types assigned where they are bound
     • let and function arguments have type annotations
     • Or, take the types from the module signature

2. Expressions that construct structured values have compound types built from the types of sub-expressions:
   (3, “foo”) : int * string
   (fun (x:int) -> x + 1) : int -> int
   Node(Empty, (3, “foo”), Empty) : (int * string) tree
3. The type of a function-application expression is obtained as the result from the function type:

- Given an expression of function type $f : T_1 \rightarrow T_2$
- and an argument expression $e : T_1$ (of the input type)
- $(f \ e) : T_2$

$$((\text{fun } (x:\text{int}) \ (y:\text{bool}) \rightarrow y) \ 3) : ??$$
3. The type of a function-application expression is obtained as the result from the function type:
- Given a function \( f : T_1 \rightarrow T_2 \)
- and an argument \( e : T_1 \) of the input type
- \( (f \, e) : T_2 \)

\(((\text{fun} (x:\text{int}) (y:\text{bool}) \rightarrow y) \; 3) : ??\)
3. The type of a function-application expression is obtained as the result from the function type:

- Given a function \( f : T_1 \rightarrow T_2 \)
- and an argument \( e : T_1 \) of the input type
- \((f \ e) : T_2\)

\[
((\text{fun } (x:\text{int}) (y:\text{bool}) \rightarrow y) \ 3) : ??
\]

\[
\text{int} \rightarrow \text{bool} \rightarrow ??
\]

??
3. The type of a function-application expression is obtained as the result from the function type:

- Given a function \( f : T_1 \rightarrow T_2 \)
- and an argument \( e : T_1 \) of the input type
- \((f e) : T_2\)

\[
((\text{fun} \ (x:\text{int}) \ (y:\text{bool}) \ \rightarrow \ y) \ 3) \ : \ ??
\]

```
int -> bool -> ??
```

```
bool
```

```
??
```
3. The type of a function-application expression is obtained as the result from the function type:
   – Given a function \( f : T_1 \rightarrow T_2 \)
   – and an argument \( e : T_1 \) of the input type
   – \((f e) : T_2\)

\[
((\text{fun } (x:\text{int}) (y:\text{bool}) \rightarrow y) \ 3) : ??
\]
3. The type of a function-application expression is obtained as the result from the function type:

- Given a function \( f : T_1 \to T_2 \)
- and an argument \( e : T_1 \) of the input type
- \( (f \ e) : T_2 \)

\[
\text{(fun (x:int) (y:bool) \to y) 3) : ??}
\]

\[
\text{int \to bool \to bool \to int}
\]

\[
\text{??}
\]
3. The type of a function-application expression is obtained as the result from the function type:

- Given a function \( f : T_1 \rightarrow T_2 \)
- and an argument \( e : T_1 \) of the input type
- \( (f \, e) : T_2 \)

\[
\text{((fun (x:int) (y:bool) -> y) 3) : ???}
\]

Here:
\( T_1 = \text{int} \)
\( T_2 = \text{bool} \rightarrow \text{bool} \)
Typechecking III

• For generic expressions:
  – *Unify* the types based on use:
  – Given a function \( f : T_1 \rightarrow T_2 \)
  – and an argument \( e : U_1 \) of the input type

  • “match up” \( T_1 \) and \( U_1 \) to obtain information about type parameters in \( T_1 \) and \( U_1 \) based on their usage

  – Obtain an *instantiation*: e.g. ‘\( \alpha = \text{int list} \)
  – *Propagate* that information to all occurrences of ‘\( \alpha \)
empty : ('k,'v) map
add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)
Example Typechecking Problem

empty : ('k,'v) map
add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)
empty : (‘k,’v) map
add : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list

fun (x:’v) -> entries (add 3 x empty)

‘v -> ??
Example Typechecking Problem

empty : (\('k, 'v\)) map
add : 'k -> 'v -> (\('k, 'v\)) map -> (\('k, 'v\)) map
entries : (\('k, 'v\)) map -> (\('k * 'v\)) list

fun (x:'v) -> entries (add 3 x empty)

\('v -> ??\)
Example Typechecking Problem

empty : ('k,'v) map
add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)
Example Typechecking Problem

empty : (‘k,’v) map
add : ‘k' -> ‘v' -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v). map -> (‘k * ‘v) list

fun (x:'v) -> entries (add 3 x empty)
Example Typechecking Problem

```plaintext
empty : ('k, 'v) map
add : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)
```

```
int 'v ('k, 'v) map

'v -> ??
```
Example Typechecking Problem

empty : ('k, 'v) map
add : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)
Example Typechecking Problem

empty : ('k,'v) map
add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)

Application:
T₁ = 'k
T₂ = 'v -> ('k,'v) map -> ('k,'v) map

Instantiate: 'k = int

'v -> ??
Example Typechecking Problem

defining types:

- `empty : ('k, 'v) map`
- `add : ('k -> 'v) -> ('k, 'v) map -> ('k, 'v) map`
- `entries : ('k, 'v) map -> ('k * 'v) list`

fun `(x : 'v) -> entries (add 3 x empty)`

Another Application:

- `T_1 = 'v`
- `T_2 = (int, 'v) map -> (int, 'v) map`

Instantiate: `'v = 'v`

`'v -> ??`
Example Typechecking Problem

empty : ('k, 'v) map
add : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)

A third Application:
T''_1 = (int,'v) map
T''_2 = (int,'v) map

Argument and argument type already agree
empty : ('k,'v) map
add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)

'v -> ??
Example Typechecking Problem

```
empty : ('k,'v) map
add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)
```

Another Application:

U₁ = ('k,'v) map
U₂ = ('k * 'v) list

Unify U₁ with T''₂
('k,'v) map ~~~ (int,'v) map

Instantiate 'k = int

'v -> ??
Example Typechecking Problem

empty : ('k,'v) map
add : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)

Another Application:
U₁ = (int,'v) map
U₂ = (int * 'v) list

U₁ -> U₂

T₂

int 'v ('int,'v) map

T'₂

U₂ = (int * 'v) list

U₂ = (int * 'v) list

T''₂ = ('int,'v) map

'v -> ??
empty : ('k,'v) map
dd : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries : ('k, 'v) map -> ('k * 'v) list

fun (x:'v) -> entries (add 3 x empty)

Another Application:
U₁ = (int,'v) map
U₂ = (int * 'v) list

U₁ -> U₂

int 'v ('int,'v) map

T₂

U₁ -> U₂

int 'v ('int,'v) map

T₂

T''₂= ('int,'v) map

U₂= (int * 'v) list

'v -> (int * 'v) list
Ill-typed Expressions?

- An expression is ill-typed if, during this type checking process, inconsistent constraints are encountered:

```haskell
empty    : ('k,'v) map
add      : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries  : ('k,'v) map -> ('k * 'v) list
```

```haskell
add 3 true (add "foo" false empty)
```

Error: found int but expected string