EVENT PROGRAMMING AND NEXT STEPS

SAM GREEN

1. API Design

Yesterday's lab focused on an implementation of the event system. To this point, we haven't talked much about what makes a useful API or a good interface. Let's spend some time examining this interface.

```
module type WEVENT =
sig
  (* The event listener identifier type. *)
  type id

  (* The event type. *)
  type 'a event

  (* Create a new event. *)
  val new_event : unit -> 'a event

  (* Add a listener to an event which is called every time the event
     is fired. Return an identifier for the listener. *)
  val add_listener : 'a event -> ('a -> unit) -> id

  (* Remove a listener from being called when an event is fired. Has no effect
     if the listener is not waiting for the event. *)
  val remove_listener : 'a event -> id -> unit

  (* Signal that an event has occurred.  The 'a value is passed to each
     function waiting for the event. *)
  val fire_event : 'a event -> 'a -> unit
end
```

Questions to consider:

Date: April 14, 2017.

- Is the API well-documented?
- Are the functions well-named?
- Does the API provide all of the necessary tools?

  Solution:

  In general, I would say yes, but it's offloads some management onto the client. Think of what happened when we had to change all of the listeners for the `newswire` event, for example.
- What, if anything, is missing?

  Solution: Some candidates:
  - One-shot listener function.
  - Get callbacks by id function.
  - Delete all callbacks / reset event method.
  - Fire multiple times function.

Is this module adequate for a full event system?

2. Implementation Questions

Let's look in sequence at each of the portions of the implementation of the WEvent module and see they could be improved or how they are bad.

2.1. **Event and waiter implementation.** We defined the following type to represent events. What do you think of them?

```
type 'a waiter = {id : id ; action : 'a -> unit}
type 'a event = 'a waiter list ref ;;
```

What do you think of this implementation? Are there any problems with scalability here? Readability? Maintainability?

Some things to think about:

- The system is set up around random access. Many event systems require registering lots of events, and graphics systems also in general need to be performant. What are the asymptotic times for lookup and delete for callbacks? What simple data structure could we use instead?
- Why do we need a `ref` event type? Why aren't waiters exposed to the client?

3. Fire event implementation

What do you think of this implementation of the fire_event function?

```
let fire_event (e : 'a event) (v : 'a) : unit =
  let waiters = !e in
  let _ = List.map (fun w -> w.action v)
  waiters in ()
```

4. Error-Message Interpretation

(Thanks to Katherine Binney for this exercise).

Many people in the lab yesterday started with following as their implementation of events.

```
let newswire = fun _ -> WEvent.new_event;
```

but this resulted in the following error message later on:

```
let fnn_listener =
    WEvent.add_listener newswire fakeNewsNetwork ;;
Error: This expression has type 'a -> unit -> 'b WEvent.event
```

As practice for the midterm and the final project, what does this error mean? How do we debug it?

(Hint: I've said it a few times this semester.)

5. Type Annotation Quick Hits

There were a few interesting examples of "delayed" type inference this week. To see what I mean, let's do the following exercises (thanks to Katherine Binney, again):

Exercise 1. *Write an OCaml expression for an empty string list. : Here's an empty list:*

```
# let x = [] ;;
val x : 'a list = []
```

But it's polymorphic! Remember, we can explicitly type it on 2 ways.

```
# let x : string list = [] ;;
val x : string list = []
```

or

```
# let x = [] ;;
val x : 'a list = []
# "hello"::x ;;
```

```
- : string list = ["hello"]
# x ;;
- : 'a list = []
```

Wait, what?? Shouldn't this last `x` *have type-checked as a string list? No! The type of the value* `x` *hasn't actually changed. How does this compare to the way polymorphic events type-checked?* □

Exercise 2. *Write a function* `map_int` *that maps over lists of integers.*

```
# let map_int (f : int -> 'b) (lst : int list) : 'b list =
  let rec map' lst =
    match lst with
    | [] -> []
    | h :: t -> (f h) :: (map' t) in
  map' lst
;;
val map_int : (int -> 'b) -> int list -> 'b list = <fun>
```

□

6. Understanding error messages (ie be the TF)

Note: Taken from (mostly private) piazza questions. We think giving you fish is ok, but we also want to teach you to fish. (Thanks to Katherine Binney, again.)

Exercise 3. *What's the likely cause of this error message?*

```
let m1 = new mass 0. 0. ;;
let _ = m1#move  m1;;

"Error: This expression has type float -> Masses.mass
       It has no method move"
```

□

Exercise 4. *This error message is a bit trickier.*

```
let g = new point -1.0 0.0;;
"Error: This expression has type float -> float -> point
but an expression was expected of type int."
```

□

7. Next Steps

Hopefully CS 51 has you excited about future courses in Computer Science.

Courses I recommend:

- CS 136
- CS 121
- CS 124
- CS 61
- CS 134
- Stat 110
- CS 187 (I hear the usual Prof. is GREAT.)

I am happy to talk about them and others! I also encourage you to be brave about 200-level courses – I wasn't particularly, and I missed out on some good stuff.