

Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 3 - Sequence 3: Case study: a story teller



Case study: a story teller

- ▶ Types play a central part in *OCaml* programming.
- ▶ Writing the right type declarations helps the programmer implement the right program.
- ▶ We will now illustrate this **type-directed** programming.

A story teller I

```
type story = {  
  context          : context;  
  perturbation     : event;  
  adventure        : event list;  
  conclusion       : context;  
}  
  
and context = { characters : character list }  
and character = { name  : string; state : state; location : location }  
and event = Change of character * state | Action of character * action  
and state = Happy | Hungry  
and action = Eat | GoToRestaurant  
and location = Apartment | Restaurant;;
```

A story teller II

```
# type story = {  
  context : context;  
  perturbation : event;  
  adventure : event list;  
  conclusion : context;  
}  
  
and context = { characters : character list; }  
and character = {  
  name : string;  
  state : state;  
  location : location;  
}  
  
and event =  
  Change of character * state  
  | Action of character * action  
and state = Happy | Hungry
```

A story teller III

```
and action = Eat | GoToRestaurant  
and location = Apartment | Restaurant
```

A story teller IV

```
let compatible_actions_for_character character context =  
  match character with  
    | { location = Restaurant } -> [Eat]  
    | { location = Apartment } -> [GoToRestaurant]  
;;  
# val compatible_actions_for_character :  
  character -> 'a -> action list = <fun>
```

A story teller V

```
let apply_action character = function
  | Eat ->
    { state = Happy;
      location = character.location; name = character.name }
  | GoToRestaurant ->
    { location = Restaurant;
      state = character.state; name = character.name }
;;
# val apply_action : character -> action -> character = <fun>
```

A story teller VI

```
let compatible_actions context =  
  let rec aux = function  
    | [] -> []  
    | character :: cs ->  
      let can_do = compatible_actions_for_character character  
context in  
      let rec aux' = function  
        | [] -> []  
        | a :: actions -> Action (character, a) :: aux' actions  
      in  
      aux' can_do  
  in  
  aux context.characters  
;;  
# val compatible_actions : context -> event list = <fun>
```


A story teller VII

```
let possible_changes_for_character character context =  
  match character with  
  | { state = Happy } -> [Hungry]  
  | { state = Hungry } -> []  
;;  
# val possible_changes_for_character :  
  character -> 'a -> state list = <fun>  
let apply_change character state =  
  { name = character.name; state = state; location =  
    character.location }  
;;  
# val apply_change : character -> state -> character = <fun>
```

A story teller VIII

```
let possible_changes context =  
  let rec aux = function  
    | [] -> []  
    | character :: cs ->  
      let possible_changes = possible_changes_for_character  
character context in  
      let rec aux' = function  
        | [] -> []  
        | c :: changes -> Change (character, c) :: aux' changes  
      in  
      aux' possible_changes  
  in  
  aux context.characters  
;;  
# val possible_changes : context -> event list = <fun>
```

A story teller IX

```
let character_of_event = function
  | Action (character, _) -> character
  | Change (character, _) -> character
;;
# val character_of_event : event -> character = <fun>
```

A story teller X

```
let apply event context =  
  let rec aux = function  
    | [] -> assert false  
    | character :: cs ->  
      if character = character_of_event event then  
        match event with  
          | Action (_, action) -> apply_action character action :: cs  
          | Change (_, change) -> apply_change character change :: cs  
      else  
        character :: aux cs  
  in  
    { characters = aux context.characters }  
;;  
# val apply : event -> context -> context = <fun>
```

A story teller XI

```
let rec is_one_of state states =  
  match states with  
  | [] -> false  
  | state' :: ss -> state = state' || is_one_of state ss  
;;  
# val is_one_of : 'a -> 'a list -> bool = <fun>
```

A story teller XII

```
let rec all_characters_are states = function
  | [] ->
    true
  | character :: cs ->
    is_one_of character.state states && all_characters_are states cs
;;
# val all_characters_are :
  state list -> character list -> bool = <fun>
```

A story teller XIII

```
let random_pick xs =  
  List.nth xs (Random.int (List.length xs))  
;;  
# val random_pick : 'a list -> 'a = <fun>  
let something_happens context =  
  let what_can_happen = compatible_actions context @  
    possible_changes context in  
  let event = random_pick what_can_happen in  
  event, apply event context  
;;  
# val something_happens : context -> event * context = <fun>  
let happy context =  
  all_characters_are [Happy] context.characters  
;;  
# val happy : context -> bool = <fun>
```

A story teller XIV

```
let rec end_story events context =  
  if happy context then  
    context, List.rev events  
  else  
    let event, context = something_happens context in  
    end_story (event :: events) context  
;;  
# val end_story :  
  event list -> context -> context * event list = <fun>
```


A story teller XV

```
let make_story initial_context =  
  let perturbation, context = something_happens initial_context in  
  let conclusion, adventure = end_story [] context in  
  {  
    context = initial_context;  
    perturbation = perturbation;  
    adventure = adventure;  
    conclusion = conclusion  
  }  
;;  
# val make_story : context -> story = <fun>
```

A story teller XVI

```
let describe_location = function
  | Apartment -> "at_home"
  | Restaurant -> "at_the_restaurant"
;;
# val describe_location : location -> string = <fun>
let describe_state = function
  | Happy -> "happy"
  | Hungry -> "hungry"
;;
# val describe_state : state -> string = <fun>
```

A story teller XVII

```
let describe character =  
  character.name ^ " was "  
  ^ describe_location character.location  
  ^ " and was " ^ describe_state character.state ^ ". "  
;;  
# val describe : character -> string = <fun>
```

A story teller XVIII

```
let tell_context context =  
  let rec aux = function  
    | [] -> ""  
    | character :: characters -> describe character ^ aux characters  
  in  
    aux context.characters  
;;  
# val tell_context : context -> string = <fun>
```

A story teller XIX

```
let tell_action = function
  | Eat -> "ate"
  | GoToRestaurant -> "went_to_the_restaurant"
;;
# val tell_action : action -> string = <fun>
```

A story teller XX

```
let tell_event = function
  | Action (character, action) ->
    character.name ^ " " ^ tell_action action ^ ". "
  | Change (character, state) ->
    character.name ^ " was made " ^ describe_state state ^ ". "
;;
# val tell_event : event -> string = <fun>
```

A story teller XXI

```
let rec tell_adventure = function
  | [] -> ""
  | event :: adventure -> tell_event event ^ tell_adventure adventure
;;
# val tell_adventure : event list -> string = <fun>
```

A story teller XXII

```
let tell story =  
  "Once upon a time, "  
  ^ tell_context story.context  
  ^ "One day, something wrong happened. "  
  ^ tell_event story.perturbation  
  ^ tell_adventure story.adventure  
  ^ "At the end, the peace was restored. "  
  ^ tell_context story.conclusion  
;;  
# val tell : story -> string = <fun>
```


A story teller XXIII

```
let story = tell (make_story {  
  characters = [  
    { name = "Sophie"; location = Apartment; state = Happy };  
    { name = "Socrate"; location = Apartment; state = Happy };  
  ]  
});;  
# val story : string =  
  "Once upon a time, Sophie was at home and was happy. Socrate was  
  at home and was happy. One day, something wrong happened. Sophie  
  went to the restaurant. At the end, the peace was restored.  
  Sophie was at the restaurant and was happy. Socrate was at home  
  and was happy."
```

Assisted incremental programming

- ▶ Our stories are a bit boring...
- ▶ If we change type declarations, the **type-checker will assist us** by pinpointing what part of the code must be updated.
- ▶ Assume now that our characters are zombies and the restaurant can be closed...

Higher-order programming

- ▶ The first version of our story teller is 178 lines long.
- ▶ A skilled *OCaml* programmer writes the same code in 133 lines (-25%).

```
let rec all_characters_are states = function
| [] -> true
| character :: cs ->
    is_one_of states character.state && all_characters_are states cs
```

... can be written using a **higher-order function**:

```
let all_characters_are states =
    List.for_all (is_one_of states)
```

Next week
you will learn how to write
your own higher-order function!