lablgtk2-react, adding Functional Reactive Programming to lablgtk2

Adrien Nader

October 2, 2011

Abstract

Functional Reactive Programming (FRP) libraries have been available for several years, but their use has remained limited in the context of GUI programming that is characterized by a high number of inputs and outputs, which can all be entangled together.

Contents

1 Presentation **2**

2 A guide for a first try: tab-based interfaces **2**

3 Benefits **4**

3.1 Separation from the UI code 4

3.2 Number of input → output links in the program 4

4 Complements on the earlier example **5**

4.1 Identifiers 5

4.2 Nested data structures 6

1 Presentation

Lablgtk-react is mostly a way to structure programs in order to take advantage of the FRP model.

Tools to create graphical user interfaces by dragging and dropping visual elements to a canvas have existed for a long time and have been working fairly well for programs which are mostly interfaces without much other code.

Whenever there is work to do outside of them, they tend to create unusable code because they entangle the UI code with everything else.

The UI code is usually mostly dumb and well-suited for such tools. Everything not dumb doesn't fit however.

LablgtkReact helps separate (isolate) the UI code from the rest. The UI code can be written in an imperative and/or object style while the rest can be written in a purely functionally.

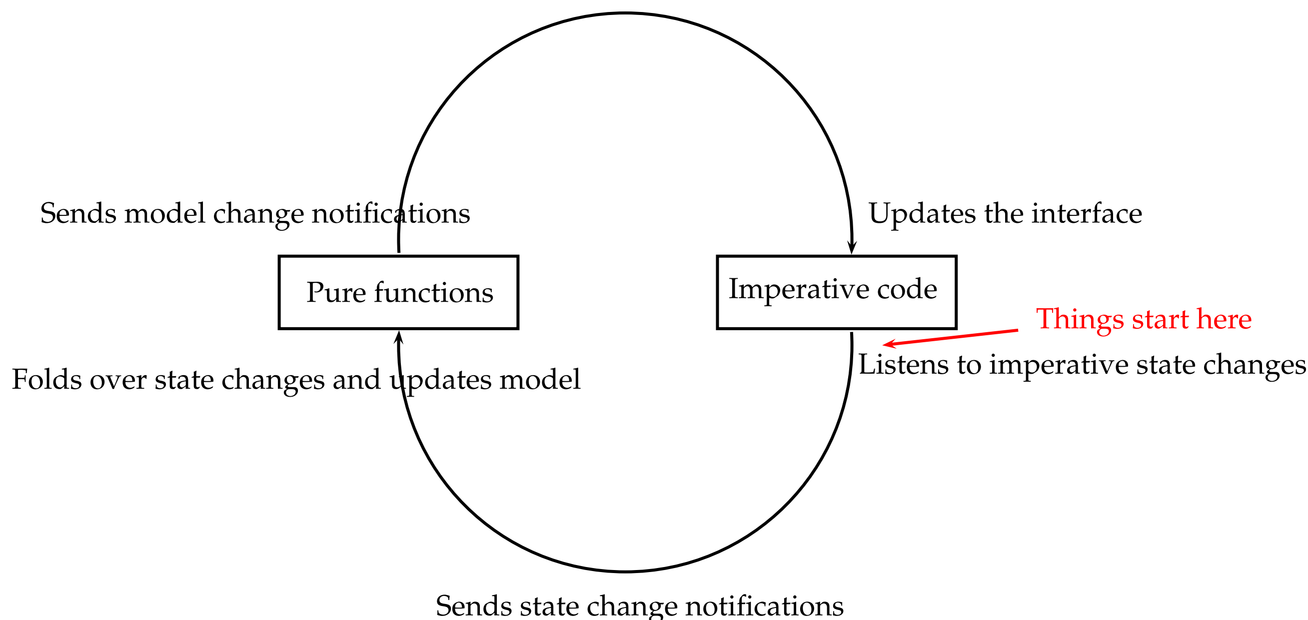

Figure 1: Organization of the paradigm

2 A guide for a first try: tab-based interfaces

The first step is to have a model for the application that you intend to write. FRP is all about making it possible to use functional code so this shouldn't come as a surprise.

For an interface with tabs and a representation of the elements, the following module signature would work:

```
 (* We use a Zipper to model tabs as an ordered set which has one element
  * focused. We can move the focus to another element and we can add and remove
  * elements (and move them). *)
module Zipper : sig
  type 'a t
  val length : 'a t -> int
```

```
    val add : 'a t -> 'a -> 'a t
    val rm : 'a t -> 'a t (* removes the current element *)
    val previous : 'a t -> 'a t (* focus (click) the previous element *)
    val next : 'a t -> 'a t (* focus (click) the next element *)
    val nth : 'a t -> int -> 'a t (* focus the nth element *)
  end
```

Next you need to express all the operations (called "actions" later on) that are done on your model with a single type. For instance, with the previous example, we would augment our Zipper module like:

```
module ZipperWithAction = struct
  include Zipper
  type 'a action =
    | `Add of 'a
    | `Rm
    | `Previous
    | `Next
    | `Nth of int
end
```

The modules values could probably be wrapped into (fun () → ...) but that seemed more annoying to do and less readable.

Of course, this type has to be used somewhere and you need to provide a function to match its members to operations in the module:

```
let on_action set = function
  | `Add e -> fun set -> add set e
  | `Rm -> rm
  | `Previous -> previous
  | `Next -> next
  | `Nth i -> fun set -> nth set i
```

And that's it for the functional side of the program.

This is enough to instantiate the LablgtkReact.Core functor:

```
module Set = LablgtkReact.Core (React.E) (ZipperWithAction)
```

Next step is to create the corresponding core object :

```
let core : unit Page.t Set.core = new Set.core () in
...
```

(the "unit" in "unit Page.t" will go away in the future)

Now, the functional part is completely done and the imperative side needs to send action messages to make it evolve.

For instance, the following code snippets associates Ctrl+t (create new page) and Ctrl+w (close page) shortcuts to `Add and `Remove actions respectively:

```
(* core#send has type: 'a action -> unit *)
ignore (GtkSignal.connect ~sgn:create_new_page
  ~callback:(fun () -> core#send (`Add Page.new_)) tabs#as_notebook);
ignore (GtkSignal.connect ~sgn:close_page
  ~callback:(fun () -> core#send `Rm) tabs#as_notebook);
```

The last step is to make the imperative side react to changes from the functional part: messages originate from the imperative world, are handled in the functional world and are then handled in the imperative world too. This gives a lot of control on how to react to events, including silencing events. (this is circular and a mechanism to handle that properly is mentioned later on)

```
let tabs = GPack.notebook () in
(* sink has type LablgtkReact.Core.callback:
     type 'a callback = 'a t -> 'a action -> unit *)
let sink set = function
  | `Rm -> tabs#remove_page tabs#current_page
  | `Add (options, e) -> add tabs (new GWebkit.webview e)
  | `Nth i -> tabs#goto_page i
  [ ... ]
in
core#connect sink
```

That's pretty much all it takes to use lablgtk-react.

3 Benefits

3.1 Separation from the UI code

As stated earlier, by using LablgtkReact, it is possible to write everything in a functional style without mentioning the UI toolkit except for the code which is directly handling the interface.

It's actually possible (and recommended) to compile most things without mentionning lablgtk2 in the compilation command.

3.2 Number of input → output links in the program

A mostly unexpected consequences of how the separation has been implemented is that the various input UI elements have to be channeled through a single data flow. Similarly, the outputs from the functional code and to the UI code originate from a single data flow.

In this diagram, we have n inputs (buttons, text fields, mouse buttons, ...) and m outputs (mostly display elements). Any input element can act on any output element, forcing us to use $O(n * m)$ links.

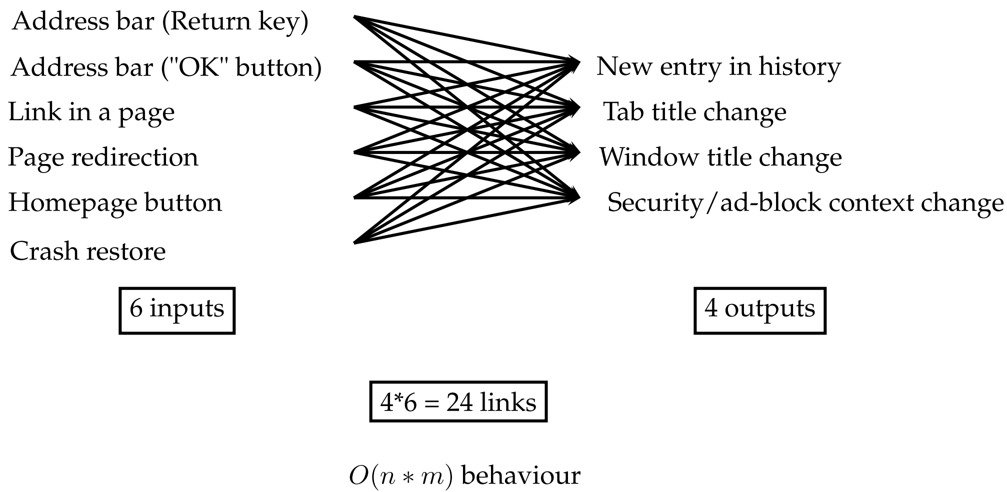

Figure 2: Usual number of links between input and output visual elements

However, with the organization for lablgtk-react, the links are like this:

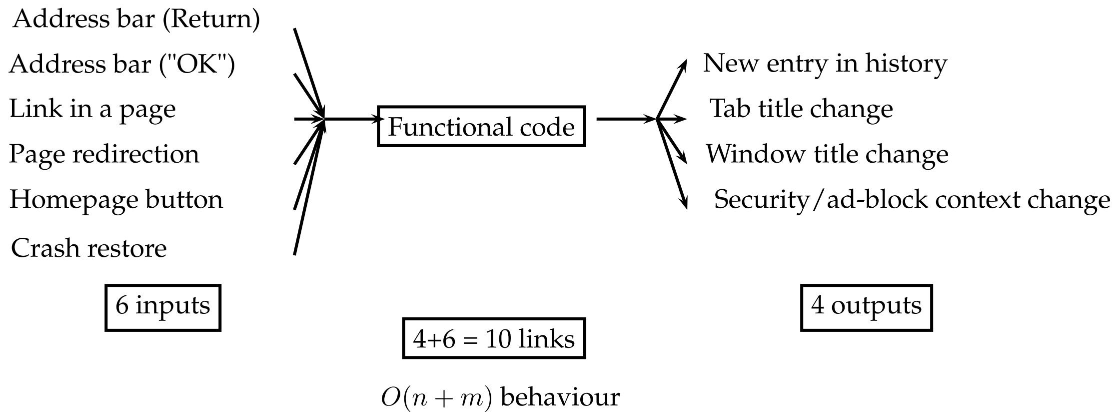

Figure 3: Reduced number of links between input and output elements

It turns out there are less links to create and the asymptotic behaviour is much better.

Some peoples have told me that this reminded them of the MVC (Model-View-Controller) pattern. It's probably not too different but that's not something I was actively after. The only thing I'm sure of is that the code is simple and easy to use.

4 Complements on the earlier example

4.1 Identifiers

The first diagram shows that there is a loop in the event chain. For instance, graphical element A could send a message M to the functional code that would then forward the message M' to all graphical elements. The element A often needs to know if it is the origin of the message or not.

The type of on_action is actually:

```
val on_action : ('a * 'b) t -> 'a -> 'b action -> ('a * 'b) t
```

The 'a type parameter is actually an identifier which is provided as LablgtkReact.Core.id. It is opaque and shouldn't be accessed directly.

Both action emitters and receivers have an associated identifier.

Similarly, the type of callbacks which are arguments to the core#connect method are actually:

```
type 'a callback = bool -> (id * 'a) t -> 'a action -> unit
```

The bool parameter indicates whether the message receiver was also the original message emitter: from_self = (id_message = id_own).

As said earlier, the id shouldn't be used directly: only through the bool parameter.

NOTE: this is currently a bit broken and identifiers are not correctly generated and given to emitters and receivers. The API will change to fix that but only slightly. You can simply discard the value of the boolean from_self parameter; it is not required for most cases.

4.2 Nested data structures

Since we are using pure data structures, when we nest them and change the inner ones, we need to propagate the changes to the parent ones.

This is achieved through so called "container_link":

```
type 'a container_link = (id * 'a) t -> 'a action -> unit
class ['a] core : ?container_link:'a container_link -> unit -> object
  val mutable container_link : 'a container_link option
  method set_container_link : 'a container_link option -> unit
end
```

The container_link is called each time the data structure is changed. As such, data structures will usually have to handle setting values directly (content change, replace old content with new content).

You could have an action (`Set of 'a) for that for instance.

Of course, you are free to use imperative data structures in some places too. Actually, you are free to do whatever you're used to do in OCaml.