Introduction to Lablgtk (from its internals)

Adrien Nader - OCaml Users in PariS

Table Of Contents

What is GTK+

- Graphical toolkit available on many platforms
- C code with an object layer done through macros and runtime checks
- Objects have methods, signals, properties (with automatic getters, setters and `notify::${property-name}` signals)
- Uses an event loop to trigger callbacks on the emission of signals

Lablgtk

- Bindings to GTK+-2 (and GTK+-1); GTK+-3 WIP
- and gtkGL, glade, gnomecanvas, gnomehtml, gnomeui, gtkspell, rsvg, gtksourceview(2)
- Provides a high-level OCaml-style API (e.g labels are used everywhere)

Development

- Project is hosted on the OCaml Forge with Git
- Development depends on demand
- API coverage increased only on-demand to avoid bit-rot of unused code
- Additions most usually simple to do: they're done quickly
- Join to work on what you want to change

GTK+-3 Status

- New GTK+ version which was released around 2 years ago
- Not that many big changes but a few annoying ones
- Pushes the use of gobject-introspection
- ... which is a good idea but has its issues: undocumented, pythonic, not meant for compiled bindings

Lablgtk Primer

Lablgtk Primer

A basic example

A window with a big text zone and a button.
When the button is clicked, a modal window pops up and asks the user for text to put in the text zone below.
Step-by-step in the slides that follow.

How to browse the API

- ocamlfind ocamlbrowser -package lablgtk2
- Merlin for emacs and vim, ocp-index, ...
- gtk.org for detailled explanation: lablgtk's API follows the C API very closely

Handy additions

- GToolbox has convenience functions: menus, dialog boxes, lists and trees, shortcuts, ... yours
- Extra libs like lablgtk-extras
- Integrate lwt/ocamlnet into glib's main loop
- Glade, either by loading the description at runtime or translating it to OCaml

About the code in these slides

To run the codes, load ocamlfind, lablgtk2 and react in the toplevel:

```
#use "topfind";;
#require "lablgtk2.auto-init";;
#require "react";;
```

Initialize glib and GTK+ (requires a display).

```
let () =
  GMain.init ();
```

Create a window with some default values.

```
let () =
  GMain.init ();
  let w = GWindow.window ~width:320 ~height:240 ~title:"Mini demo" () in
```

Windows can only contain objects; insert a box which can hold many objects and arrange them.

```
let () =
  GMain.init ();
  let w = GWindow.window ~width:320 ~height:240 ~title:"Mini demo" () in
  let vbox = GPack.vbox ~packing:w#add () in
```

Add a label in the box with default settings and make it take as much space as available.

```
let () =
  GMain.init ();
  let w = GWindow.window ~width:320 ~height:240 ~title:"Mini demo" () in
  let vbox = GPack.vbox ~packing:w#add () in
  let label = GMisc.label ~text:"<empty>" ~selectable:true ~line_wrap:true
    ~justify:`CENTER ~packing:(vbox#pack ~expand:true) () in
```

Add a button at the end of the box.

```
let () =
  GMain.init ();
  let w = GWindow.window ~width:320 ~height:240 ~title:"Mini demo" () in
  let vbox = GPack.vbox ~packing:w#add () in
  let label = GMisc.label ~text:"<empty>" ~selectable:true ~line_wrap:true
    ~justify:`CENTER ~packing:(vbox#pack ~expand:true) () in
  let t = "Change text" in
  let btn = GButton.button ~packing:vbox#pack ~label:t () in
```

Add a callback which is triggered upon clicking the button.

```
let () =
  GMain.init ();
  let w = GWindow.window ~width:320 ~height:240 ~title:"Mini demo" () in
  let vbox = GPack.vbox ~packing:w#add () in
  let label = GMisc.label ~text:"<empty>" ~selectable:true ~line_wrap:true
    ~justify:`CENTER ~packing:(vbox#pack ~expand:true) () in
  let t = "Change text" in
  let btn = GButton.button ~packing:vbox#pack ~label:t () in
  btn#connect#clicked (fun () ->
```

The callback spawns a toolbox asking for text which will replace the text in our label above.

```
let () =
  GMain.init ();
  let w = GWindow.window ~width:320 ~height:240 ~title:"Mini demo" () in
  let vbox = GPack.vbox ~packing:w#add () in
  let label = GMisc.label ~text:"<empty>" ~selectable:true ~line_wrap:true
    ~justify:`CENTER ~packing:(vbox#pack ~expand:true) () in
  let t = "Change text" in
  let btn = GButton.button ~packing:vbox#pack ~label:t () in
  btn#connect#clicked (fun () -> match GToolbox.input_text ~title:t t with
    | Some text -> label#set_text text | None -> ());
```

Show the window.

```
let () =
  GMain.init ();
  let w = GWindow.window ~width:320 ~height:240 ~title:"Mini demo" () in
  let vbox = GPack.vbox ~packing:w#add () in
  let label = GMisc.label ~text:"<empty>" ~selectable:true ~line_wrap:true
    ~justify:`CENTER ~packing:(vbox#pack ~expand:true) () in
  let t = "Change text" in
  let btn = GButton.button ~packing:vbox#pack ~label:t () in
  btn#connect#clicked (fun () -> match GToolbox.input_text ~title:t t with
    | Some text -> label#set_text text | None -> ());
  w#show ();
```

Start the mainloop.

```
let () =
  GMain.init ();
  let w = GWindow.window ~width:320 ~height:240 ~title:"Mini demo" () in
  let vbox = GPack.vbox ~packing:w#add () in
  let label = GMisc.label ~text:"<empty>" ~selectable:true ~line_wrap:true
    ~justify:`CENTER ~packing:(vbox#pack ~expand:true) () in
  let t = "Change text" in
  let btn = GButton.button ~packing:vbox#pack ~label:t () in
  btn#connect#clicked (fun () -> match GToolbox.input_text ~title:t t with
    | Some text -> label#set_text text | None -> ());
  w#show ();
  GMain.main ()
```

Recap: most common constructs

- new widgets: `GWindow.window`, `GPack.vbox`
- properties:
  - set when creating the widget: `GWidget.widget ~property:value ()`
  - set later on: `widget#set_property value`
  - get : `widget#property`
- register callbacks:
  - `#connect#clicked (fun () -> eprintf "Clicked!")`
  - `#connect#notify_${property} ~callback:()`
- add items to containers:
  - When creating the widget, `#pack` if available, `#add` otherwise:
    - `GButton.button ~text:"42" ~packing:(box#pack ~expand:false) ()`
    - `GButton.button ~text:"42" ~packing:win#add ()`
  - Or afterwards: `#coerce` the object to the base `widget` type:

    `box#pack button#coerce`

Functional Reactive Programming for GUIs

Functional Reactive Programming for GUIs

Functional vs. imperative mismatch

- Callbacks for signals usually don't return a value:

```
(GButton.button ())#connect#clicked;;
- : callback:(unit -> unit) -> GtkSignal.id = <fun>
```

- We have to use imperative code to count the number of clicks on a button:

```
type state = { count : int } ;;
let s = ref { count = 0 } ;;
let () =
  let w = GWindow.window ~show:true () in
  let b = GButton.button ~packing:w#add () in
  let callback () =
    s := { !s with count = !s.count + 1 } in
  b#connect#clicked ~callback;
  GMain.main ()
```

Functional vs. imperative mismatch (cont.)

- But OCaml is multi-paradigm!
- Impossible to be both imperative and functional for the program architecture.
- Once `type state` becomes more complicated, initialization becomes

```
let state = ref (Obj.magic 0)
```

or

```
type state1 = state Lazy.t ;;
let state = ref (lazy { init with values you hope to be "ready" })
```

- Having laziness by default like in Haskell probably helps but only slightly.

Functional vs. imperative mismatch - some FRP

- We want functional updates: callbacks would be `(state -> state)`:
  ```
  ~callback:(fun state0 -> { state0 with count = state0.count + 1 })
  ```
  And no init to `Obj.magic` or using `lazy`.
- FRP can help (and it's simple)

Functional vs. imperative mismatch - some FRP

A trivial example which counts the number of clicks on a button.

```
type state = { count : int }

let event, event_send = React.E.create ()

let state_machine (s : state) event =
  Printf.printf "Count was %d. %!" s.count;
  match event with
  | `Incr -> { s with count = s.count + 1 }
  | `Decr -> { s with count = s.count - 1 }

let () =
  let w = GWindow.window ~show:true () in
  let b = GButton.button ~packing:w#add () in
  b#connect#clicked (fun () -> event_send `Incr);
  let _state = React.E.fold state_machine { count = 0 } event in
  GMain.main ()
```

Bindings creation

Bindings creation

Bindings creation

- Several layers: C stubs, object layer, signals, properties...
- Mostly generated
- Some hand-written code, especially for the higher-level layers (convenience functions)

Stubs and low-level API - 1

- C macros:

```
ML_1 (gtk_window_new, Window_type_val, Val_GtkWidget_window)
```

- A DSL, varcc:

```
type arrow_type = "GTK_ARROW_"
  [ `UP | `DOWN | `LEFT | `RIGHT ]
```

- Another DSL, propcc:

```
class Window set wrap : Bin {
  "title" gchararray : Read / Write
  method resize : "width:int -> height:int -> unit"
  signal activate_default
}
```

Stubs and low-level API - 2

- gtkWindow.ml:

```
module Window = struct
  include Window (* from propcc's output. *)
  external get_wmclass_name : [> `window ] obj -> string
      = "ml_gtk_window_get_wmclass_name"
    (* From before propcc *)
end
```

- Inheritance is handled through polymorphic variants:

```
type text_view = [ container | `textview ]
```

Functions then require values of type `[> `textview] Gtk.obj`.

Object API

Mostly boiler-plate apart from the convenience functions:

- From gWindow.ml:

```
class window_skel obj = object
  inherit window_props (* comes from propcc in ogtkBaseProps.ml *)
  method resize = Window.resize obj
end
[...]
let window ?kind =
  make_window [] ~create:(fun p -> new window (Window.create ?kind p))
```

Signals

Again, mostly boiler-plate code:

- From ogtkBaseProps.ml (generated by propcc):

```
class virtual container_sigs = object (self)
  method add = self#connect
    { Container.S.add with
      marshaller = fun f -> marshal1 conv_widget "GtkContainer::add" f }
end
```

- From gObj.ml:

```
class ['a] gobject_signals obj = object
  method private connect =
    fun sgn ~callback -> GtkSignal.connect obj ~sgn ~callback
end
```

Links

- lablgtk sources: README (must-read!), doc/
- Dawid Toton's description on how to bind GtkPrint (need to find it again)
- cowboy(-glib): http://git.ocamlcore.org/cgi-bin/gitweb.cgi?p=cowboy/cowboy.git
- lablgtk-extras: http://gtk-extras.forge.ocamlcore.org/
- lablgtk-react: http://git.ocamlcore.org/cgi-bin/gitweb.cgi?p=lablgtk-react/lablgtk-react.git
- `#self`: will probably appear as part of the documentation

Questions?
(and Patoline is nice)