

Nottui & Lwd

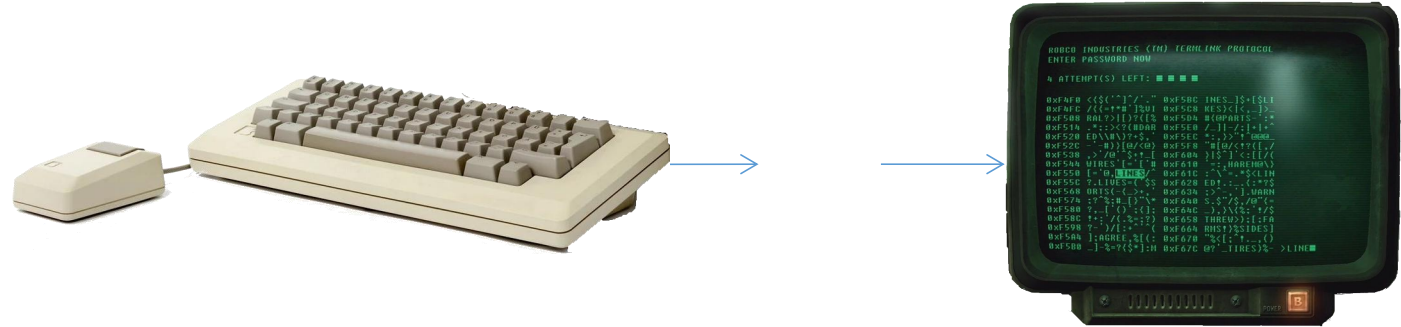
A friendly UI toolkit for the ML-programmer

Frédéric Bour, Tarides

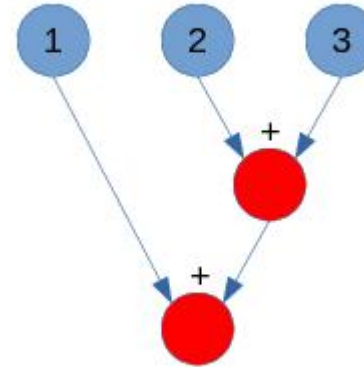
ML 2020 workshop, August 27th 2020

Two libraries

- Nottui
 - “Notty UI”
 - Terminal output
 - Keyboard/mouse input



- Lwd
 - “Lightweight document”
 - a form of incremental computation



Citty: terminal client to OCaml Labs Continuous integration

```
ci.ocamlabs.io
0install/0install
CraigFe/alcotest
CraigFe/bechamel
CraigFe/irmin
CraigFe/oskel
Julow/rss_to_mail
MagnusS/bmtune
NathanReb/ppx_yojson
avsm/ocaml
avsm/ocaml-ctypes
avsm/ocaml-dockerfile
avsm/ocaml-yaml
avsm/opam-tools
avsm/osrelease
capnproto/capnp-ocaml
dune-universe/ppx-dunive
favonia/bantorra
favonia/kusariyarou
favonia/yuujinchou
gpetiot/ocaml-weather
gs0510/index
janestreet/spawn
kit-ty-kate/labrys
mirage/alcotest
mirage/arp
mirage/awa-ssh
mirage/bigarray-compat
mirage/capnp-rpc
```

Nottui

Notty UI: drawing user interfaces in the terminal

Starting point: Notty

An OCaml library for “*Declaring* terminals”, by David Kaloper Meršinjak.

<https://github.com/pqwy/notty/>

Solves two problems:

- Making “terminal images”, with a set of pure combinators
- Managing UNIX TTYs (terminal devices):
 - setup and restore terminal contents
 - render images
 - pump events (keyboard input, mouse movements, display resize...)

Image combinators

- Primitive images: styled characters and strings

```
# let hello = I.string A.(bg red) "Hello";;  
val hello : image = Hello  
# let world = I.string A.(bg green ++ fg black) "World";;  
val world : image = World
```

- Make complex images by combining simpler images:

- horizontal and vertical concatenation
- superposition
- cropping, ...

```
# hello <|> world;;  
- : image = HelloWorld  
# hello <-> world;;  
- : image = Hello  
World
```

Very functional, a pleasure to work with :-)

Shortcomings for UI

- **Fixed size:** all images have a fixed width and height.
 - But the display size is not known in advance
 - ... and the content size might also vary dynamically
- **Only visual** information.
We would like to attach extra behaviors to the structure of the image:
 - reacting to mouse events
 - focusable areas
 - ...

Layout DSL

Draw inspiration from TeX boxes and glue model:

fixed size objects & stretchable spaces (springs)

Each **dimension** (width or height) is a pair of integers:

```
type dimension = { fixed : int; stretch : int }
```

Fixed a number of columns (or rows), reserved for the object

Stretch a factor to determine of remaining space is split among objects

Layout DSL: Intuition

Layout for :



1. Start from line width, say 20

total = 20

2. Subtract all fixed dimension

remaining = total - fixed(Hello) - fixed(World)

remaining = 20 - 5 - 5

remaining = 10

3. Sum all stretch dimension

total stretch = 2 + 1 + 2

total stretch = 5

4. Give a ratio of remaining space

remaining * object stretch / total stretch

stretch=2 $\rightarrow 10 * 2 / 5 = 4$ whitespaces

stretch=1 $\rightarrow 10 * 1 / 5 = 2$ whitespaces

____Hello__World____

Layout DSL: Properties

- Composable specification:

$$\begin{array}{l} \text{Hello} \\ \text{fixed}=5 \\ \text{stretch}=0 \end{array} \oplus \begin{array}{l} \text{stretch}=1 \\ \text{fixed}=0 \\ \text{stretch}=1 \end{array} = \begin{array}{l} \text{fixed}=5 \\ \text{stretch}=1 \end{array}$$

- Decomposable solution:

$$\begin{array}{l} \text{fixed}=5 \\ \text{stretch}=1 \\ \text{space}=8 \end{array} = \begin{array}{l} \text{fixed}=5 \\ \text{stretch}=0 \\ \text{space}=5 \end{array} \oplus \begin{array}{l} \text{fixed}=0 \\ \text{stretch}=1 \\ \text{space}=3 \end{array}$$

- Associativity:

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

Layout DSL: Benefits

- **Simple** yet expressive
Represent left, centered, right, justified text, easily emulate some flexbox-like layout, ...
- **Efficient** and suitable for incremental updates
All basic operations are $O(1)$, can be rebalanced thanks to associativity.
- **Straightforward** implementation:
 - smart constructors for specification
 - direct recursion for decomposing solution

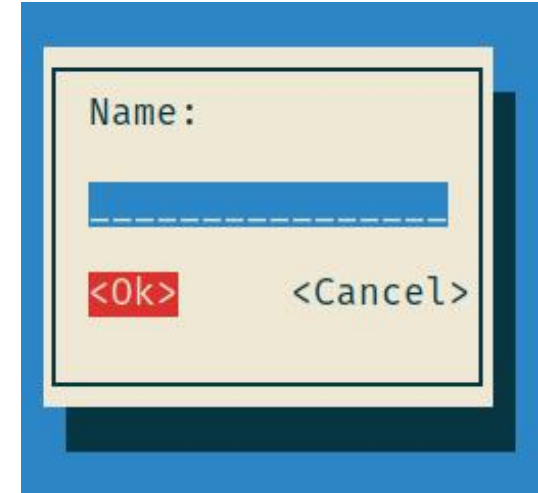
Event dispatch

Behaviors to attach to Notty images:

- focus: input field, “Ok” and “Cancel” buttons can be made active
- keyboard input:
 - pressing “tab” should switch between focus receivers
 - if input field is active, it should receive key presses
 - if a button is active, pressing enter should trigger it
- mouse click: ...

Extra constructors for non-visual behaviors:

```
type mouse_handler = x:int -> y:int -> button ->  
                    [ `Handled | `Unhandled ]  
  
val mouse_area : mouse_handler -> ui -> ui  
val focus_area, size_sensor : ...
```



Nottui: summary

Nottui	=	Notty
	+	Layout DSL
	+	Event dispatch / context probing

Still about static images: no way to update display from events.

Lwd

adding interactivity

Starting from syntax

Not a new problem:

	Terminal	Web
Syntax	Nottui.ui	HTML
Interactivity	?	DOM

The DOM: make everything mutable (change children, change attribute).

Pros: regularity

Cons: prone to spaghetti code

Starting from syntax

Not a new problem:

	Terminal	Web
Syntax	<code>Nottui.ui</code>	HTML
Interactivity	?	DOM

Starting from syntax

Not a new problem:

	Terminal	Web
Syntax	<code>Nottui.ui</code>	HTML
Interactivity	<code>Nottui.ui Lwd.t</code>	DOM

Use a type transformer: `ui Lwd.t` is a “ui” value that can change over time.

Primitives: mutable variables

```
type 'a Lwd.var  
val var : 'a -> 'a Lwd.var  
val set : 'a Lwd.var -> 'a -> unit  
val peek : 'a Lwd.var -> 'a
```

Just like an ML reference, “’a ref”. But...

```
val get : 'a Lwd.var -> 'a Lwd.t
```

get introduces a “changing value”: a value that is updated when the variable is mutated.

Composition: an applicative functor

```
type 'a Lwd.t
val pure : 'a -> 'a Lwd.t
val map : ('a -> 'b) -> 'a Lwd.t -> 'b Lwd.t
val map2 : ('a -> 'b -> 'c) -> 'a Lwd.t -> 'b Lwd.t -> 'c Lwd.t
```

Also a monad:

```
val join : 'a Lwd.t Lwd.t -> 'a Lwd.t
```

Observing results: roots

```
type 'a Lwd.root
```

```
val observe : 'a Lwd.t -> 'a Lwd.root
```

```
val sample : 'a Lwd.root -> 'a
```

```
val release : 'a Lwd.root -> unit
```

Evaluation strategy:

- `sample` only re-evaluates parts of a computation for which at least one variable was set.
- it only cares about dependencies, not about values (e.g., use “unit Lwd.var” for events).

Example: counting clicks (direct)

```
type syntax =  
  | Text of string  
  | Link of syntax * (unit -> unit)  
  | Cat of syntax * syntax  
  
let label = Text "Click to increment: "  
let counter = ref 0  
let on_click () = counter := !counter + 1  
let link =  
  Link (Text (string_of_int !counter), on_click)  
let document = Cat (label, link)
```

Click to increment: [0](#)

Example: counting clicks (Lwd)

```
type syntax =  
  | Text of string  
  | Link of syntax * (unit -> unit)  
  | Cat of syntax * syntax  
  
let label = Text "Click to increment: "  
let counter = Lwd.var 0  
let on_click () = Lwd.set counter (Lwd.peek counter + 1)  
let link =  
  Link (Text (string_of_int !counter), on_click)  
let document = Cat (label, link)
```

Click to increment: 0

Example: counting clicks (Lwd)

```
type syntax =  
  | Text of string  
  | Link of syntax * (unit -> unit)  
  | Cat of syntax * syntax  
  
let label = Text "Click to increment: "  
let counter = Lwd.var 0  
let on_click () = Lwd.set counter (Lwd.peek counter + 1)  
let link = Lwd.map  
  (fun c -> Link (Text (string_of_int c), on_click)) (get counter)  
let document = Lwd.map (fun l -> Cat (label, l)) link
```

Click to increment: 0

Example: counting clicks (Lwd)

```
type syntax =  
  | Text of string  
  | Link of syntax * (unit -> unit)  
  | Cat of syntax * syntax  
  
let label = Text "Click to increment: "  
let counter = Lwd.var 0  
let on_click () = Lwd.set counter (Lwd.peek counter + 1)  
let link = Lwd.map  
  (fun c -> Link (Text (string_of_int c), on_click)) (get counter)  
let document = Lwd.map (fun l -> Cat (label, l)) link
```

Click to increment: [0](#)

Example: counting clicks (Lwd)

```
type syntax =  
  | Text of string  
  | Link of syntax * (unit -> unit)  
  | Cat of syntax * syntax  
  
let label = Text "Click to increment: "  
let counter = Lwd.var 0  
let on_click () = Lwd.set counter (Lwd.peek counter + 1)  
let link = Lwd.map  
  (fun c -> Link (Text (string_of_int c), on_click)) (get counter)  
let document = Lwd.map (fun l -> Cat (label, l)) link
```

Click to increment: [1](#)

Comparison to other libraries

- Self adjusting computations / Janestreet's Incremental:
 - Lwd is only about dependencies, not values: no need to check for equality or memoize
 - Lwd is about observing and reacting to updates,
 - Incremental is about speeding up evaluation of a pure function, the change should not be observable from within the computation
 - In practice: Lwd is really just a subset, with simpler evaluation strategy, smaller overhead, but more recomputation
- React.js:

React.js is an abstraction over the DOM, Lwd aims to replace the DOM:
React.js could use Lwd as a backend!
- Svelte.js:

I did not know about it, thanks to reviewers for pointing it out!
It is very similar: same intention, different implementation.
I am looking forward to their progress.

Interesting features

- Synchronous and asynchronous main loops:


```
val Ui_loop.run : ?quit:bool Lwd.var -> ui Lwd.t -> unit
```

```
val Nottui_lwt.run : ?quit:unit Lwt.t -> ui Lwd.t -> unit Lwt.t
```

- Incremental collections
 - Lwd_table: mutable, doubly-linked list
 - Lwd_seq: immutable, pure-tree
 - Observe them by doing map/reduce (monoid homomorphism)
 - Incremental and efficient: minimal recomputation
- Nottui_pretty: live pretty-printing
 - Incremental version of Pottier & Pouillard / Leijen / Wadler lineage of pretty printers.
 - Can print any widget and not just text, layout with DSL and not just whitespace.
 - Cons: different from Format (OCaml built-in pretty-printer)

Nottui_pretty: Live pretty printing

```
let x =  
  match free with "foo" -> 0 | _ -> 42  
in  
x
```



Applications

BetterBoy: debugger frontend to a GameBoy emulator

A:8F F:40 B:09 C:01 D:54 E:00 H:54 L:48 PC:0040 SP:DFF1 Z:X N:✓ HC:X CA:X IF:00 IE:0D IME:X

> tile 0x0A



> tiles 0x80 0xDF

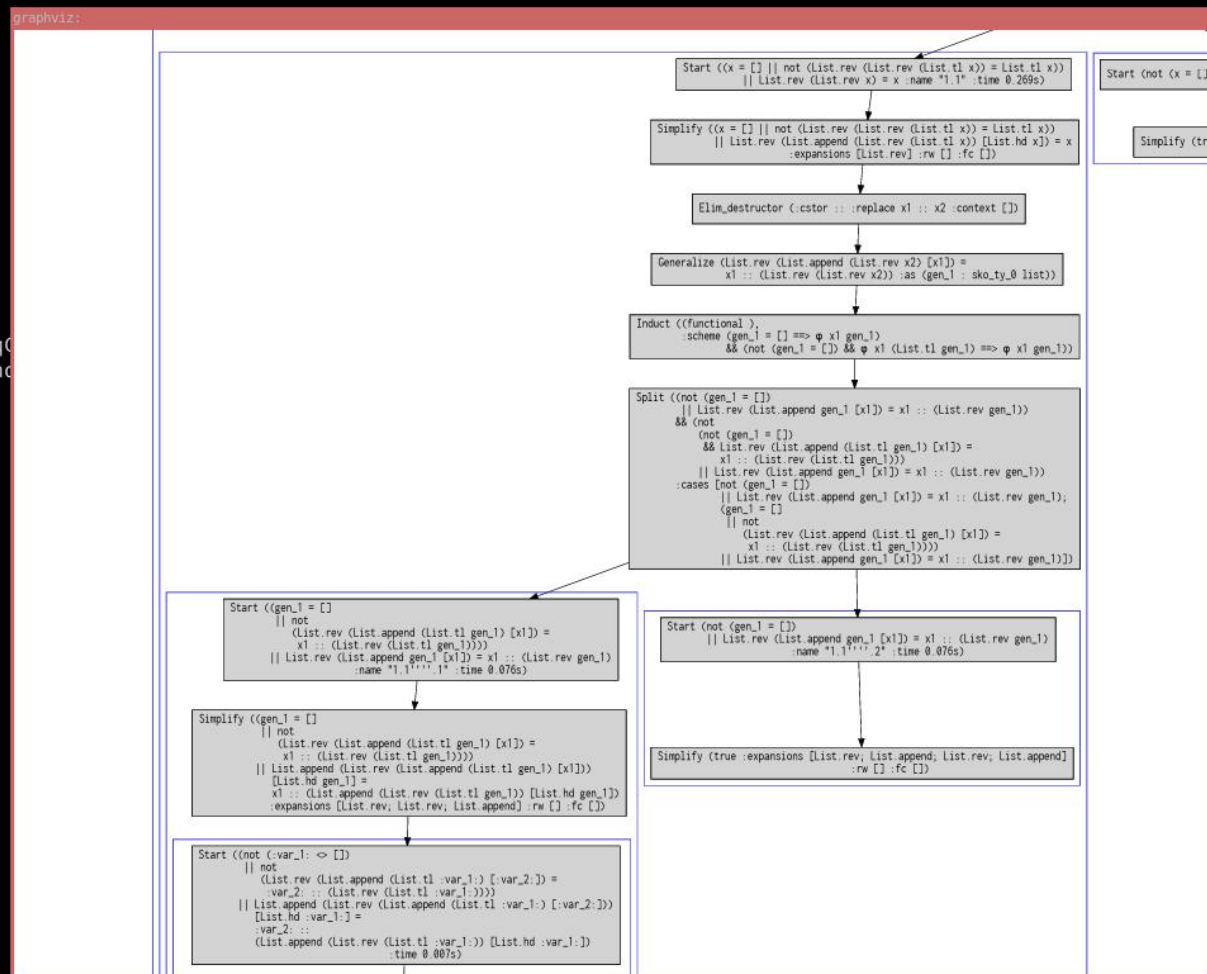


BetterBoy: debugger frontend to a GameBoy emulator

[shell][disassembler][vram viewer]											
PC/BP	REGION	ADDR	VAL	OP	SIZE	OP1	OP2	IMM	IMM	NAME	
	CART	20AE	D9	RETI	1						
	CART	20AF	3E	LD	2	A	d8	01		DelayFrame	
	CART	20B1	E0	LDH	2	(a8)	A	D6			
	CART	20B3	76	HALT	1					DelayFrame.halt	
▶	CART	20B4	F0	LDH	2	A	(a8)	D6			
	CART	20B6	A7	AND	1	A					
	CART	20B7	20	JR	2	NZ	r8	FA			
	CART	20B9	C9	RET	1						
●	CART	20BA	FA	LD	3	A	(a16)	5D	D3	LoadGBPal	
	CART	20BD	47	LD	1	B	A				
	CART	20BE	21	LD	3	HL	d16	16	21		
	CART	20C1	7D	LD	1	A	L				
	CART	20C2	90	SUB	1	B					
	CART	20C3	6F	LD	1	L	A				
	CART	20C4	30	JR	2	NC	r8	01			
	CART	20C6	25	DEC	1	H					
	CART	20C7	2A	LD	1	A	(HL+)			LoadGBPal.ok	
	CART	20C8	E0	LDH	2	(a8)	A	47			
	CART	20CA	2A	LD	1	A	(HL+)				
	CART	20CB	E0	LDH	2	(a8)	A	48			
	CART	20CD	2A	LD	1	A	(HL+)				
	CART	20CE	E0	LDH	2	(a8)	A	49			
	CART	20D0	C9	RET	1						
	CART	20D1	21	LD	3	HL	d16	0D	21	GBFadeInFromBlack	
	CART	20D4	06	LD	2	B	d8	04			
	CART	20D6	18	JR	2	r8		05			
	CART	20D8	21	LD	3	HL	d16	1C	21	GBFadeOutToWhite	
	CART	20DB	06	LD	2	B	d8	03			
	CART	20DD	2A	LD	1	A	(HL+)			GBFadeIncCommon	
	CART	20DE	E0	LDH	2	(a8)	A	47			
	CART	20E0	2A	LD	1	A	(HL+)				

Imandra: visualizing internal proof state

```
[history][control]
▼ [1]      Thm: revrev
name      revrev
type      'a list -> bool
recursive false
def-depth 5
def-ty-depth 2
call signature revrev (x : 'a list)
body      List.rev (List.rev x) = x
parametric true
validated in 0.633s
proofs    ▼ 1 proofs
           [summary][full][graph]
           [graph]
hashes    revrev CzvG+6z560Sc8Nt9jLzCF6IC9fqQ
▶ [0]      Import: "/home/simon/.opam/4.06.1/lib/mandra
▶ [-1]     Mod: Pervasives
▶ [-2]     Mod: Reflect
▶ [-3]     Mod: Float
▶ [-4]     Fun: snd
▶ [-5]     Fun: fst
▶ [-6]     Fun: pred
▶ [-7]     Fun: succ
▶ [-8]     Fun: ^
▶ [-9]     Mod: String
▶ [-10]    Mod: Set
▶ [-11]    Mod: Multiset
▶ [-12]    Mod: Map
▶ [-13]    Mod: Real
▶ [-14]    Mod: Option
▶ [-15]    Mod: Bool
▶ [-16]    Mod: Int
▶ [-17]    Fun: --
▶ [-18]    Fun: @
▶ [-19]    Mod: List
▶ [-20]    Fun: <==>
▶ [-21]    Fun: <==
▶ [-22]    Fun: ==>
▶ [-23]    Fun: /.
▶ [-24]    Fun: *.
▶ [-25]    Fun: ~-.
▶ [-26]    Fun: -.
▶ [-27]    Fun: +.
```



Conclusion

UI & interaction in ML

- Nottui & Lwd are two 100% OCaml libraries for making user interfaces.
- Compose well (so far)
- Handle quite complex user interfaces.

Future work

Polish current implementation:

- **cleanup:** resource management, context probing
- **widgets:** blessed set of default widgets (WIP)

Target the **web**:

- Manage the DOM with Lwd
- Share libraries between native and web UIs, expose similar interface

Open question: bidirectional data-bindings?

Thanks !

Thanks to Simon Cruanes and Enguerrand Decorne for being early adopters.

Thanks to you for watching!

Do you have any **questions**?