Nottui & Lwd: A friendly UI toolkit for the ML-programmer

Frédéric Bour, Tarides

May 30, 2020

Abstract

Lwd & Nottui are two small libraries that together help making terminal user interfaces.

We are targeting "developer UI", not general purpose applications. When possible, we favor convenience over expressiveness. In other words we want to fill the gap between non-interactive, printf-based debugging and logging, and rich interactive applications. One step above batch logging, without being too intrusive.

In this presentation we will review the design of the libraries and demonstrate some interesting features and applications.

1 Nottui

Nottui extends the Notty library. Notty is a library for doing "declarative terminal graphics". It provides a set of well-crafted combinators to build *terminal images*: a matrix of styled characters that can be efficiently and faithfully displayed on a computer terminal.

It is very convenient for displaying information, but it falls short for user-interfaces. Nottui covers two new aspects: layout and event dispatch.

1.1 Layout

When writing user interfaces, one generally doesn't care about manually positioning each component. Notty offers horizontal and vertical concatenation operators for placing images next to each other. Notty images are boxes with a fixed width and height. But in general, a user interface does not control the exact surface on which it will be displayed. It must accommodate to the actual dimension of the display.

Nottui loosely borrows from the box and glue model of TeX (see Beebe [2009]). Width is specified by a pair of integers (w, sw). w is the minimum width, expressed as a number of columns, and sw is the *stretchable* width:

- if set to 0, the box has a fixed width
- when the display is larger than the sum of all minimum widths, the sum Sw of all stretchable widths is computed and each stretchable box is offered a fraction sw/Sw of the remaining space.

sw can be interpreted as the strength of a spring that covers free space. Similarly, height is specified by a pair (h, sh) of minimum and stretchable heights.

This model is expressive enough to cover centered and aligned text, flexbox-like layouts, and even text justification (provided that line breaking has already been done).

1.2 Event dispatch

Producing an image is a one way process: we don't expect any feedback from the viewer of the image. But a user-interface should allow interactions with the visual components. We extend Notty images with new constructors that take callbacks that will be used to channel information from the renderer back to the program.

These fall in two main categories:

User interactions. Keyboard or mouse events, as well as some more "semantic events" such as focus and clipboard management.

Context probing. Some UI patterns need to know the actual size of the physical display (for instance to properly render scroll bars).
Others, such as mouse dragging, need to query the positioning of components. These situations depend on the physical layout.

So callbacks are used to transmit information that can only be known after the UI is fully composed: after layout, and after rendering, when the user starts acting on UI components.

1.3 Still declarative

All these features are represented by a single datatype that extends Notty images. Here is a simplified version[1]:

```
type dimension = { fixed : int; stretchable: int }
type ui =
  | Atom of image
  | Size_sensor of ui * (w:int -> h:int -> unit)
  | Resize of ui * dimension * dimension
  | Mouse_handler of ui * (unit -> bool)
  | X of ui * ui
  | Y of ui * ui
```

[1] We have hidden a caching layer as well as some constructors dealing with modal windows and focus.

Leaves are images of the Notty library and internal nodes represent different ways to transform and compose them:

- `Size_sensor (ui, f)` adds a callback that will be invoked when the display width and height is known
- `Resize (ui, width, height)` specifies a custom layout. Otherwise, layout is derived from `Atom` nodes that can only express fixed width and height.
- `Mouse_handler (ui, f)` adds a callback that is invoked when the user clicks on the space where `ui` is displayed. The callback can choose whether the event is consumed or not. If not, it propagates to the next `Mouse_handler` in the branch, if any.

This datatype is central to the Nottui library and to the ease of reasoning it offers. Values are immutable and combinators compose freely, without introducing corner-cases.

The concrete implementation deviates a bit from this definition but retains the nice composability properties. For performance reasons, each internal node propagates the layout information and introduces a caching layer. Fortunately for the user of the library, these changes does not introduce any new observable behavior and are implemented with smart constructors that do not change the asymptotic complexity (constructing a node is an $O(1)$ operation).

1.4 Dynamic documents?

So far we were able to preserve the declarative nature of Notty while extending it to better suit user interface purposes. However we are still limited to static documents: they can be rendered and the program can be notified of interesting events, but there is no way to provide visual feedback and update the display.

Dealing with such changes is the work of Lwd:

- Nottui focuses on the document representation
- Lwd focuses on efficient and coherent updates

2 Lwd, lightweight documents

Changing the content of user interfaces has always been a central problem of UI libraries. Lwd draws inspiration from two approaches:

- Object-oriented toolkits that generally rely on a big mutable tree. Then they track the mutations and update the display accordingly.
- Self-Adjusting Computations introduced by Acar [2005], and in particular the Incremental implementation. These tools allow to implement computations that efficiently handle changes to their input.

Lwd can be seen as a generic and reusable presentation of the mutation tracking of OOP toolkits. Or as a very limited form of Self-Adjusting Computation.

To avoid revisiting the whole tree, OOP toolkits store a few bits of information in each node to determine which properties are out-of-date. Changing the width a child can affect the layout of the parent that, in turn, can move all other nodes. Doing that naively would be detrimental to performance: consecutive local changes would trigger a lot of global recomputations. Thus, these operations tend to be done lazily, to batch as many changes as possible in a single computation.

SAC captures very general patterns of computations sensitive to changes of input values. Lwd captures just the subset that coincides with the way invalidation tend to be tracked in object-oriented toolkits: when in doubt, throw away the previous work. It doesn't try to memoize intermediate values and thus does not need to test equality or compute "diffs". Update scheduling is trivial: one invalidation pass, one evaluation pass. Book-keeping overhead is minimal.

This works because in our setting we don't mind recomputing things that might have changed: constructors are $O(1)$, so deciding whether we should apply one or not doesn't affect the asymptotic complexity. However it is important to not visit the parts of the tree that have surely not changed, as they could be arbitrarily big.

`Lwd` takes the form of a monad although most of the time applicative style described by Conor McBride [2007] is sufficient and encouraged:

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

A value of type `'a Lwd.t` can be thought of as a value of type `'a` that can change over time. Primitive changes are introduced by the `var` type:

```
type 'a Lwd.var
val get : 'a Lwd.var -> 'a Lwd.t
val set : 'a Lwd.var -> 'a -> unit
val peek : 'a Lwd.var -> 'a
```

`var` is the equivalent of builtin `ref` type lifted to the `Lwd` monad. `peek` and `set` are exactly `(!)` and `(:=)`. Their return type is not "in the monad". Indeed, their intended use is to update the input of the graph from the outside. The added value comes from the `get` operation that lifts a variable to a *changing value*.

Finally, observing the result of an `'a Lwd.t` computation is done using explicit roots:

```
type 'a Lwd.root
val observe : ?on_invalidate:('a -> unit) ->
  'a Lwd.t -> 'a Lwd.root
val sample : 'a Lwd.root -> 'a
val release : 'a Lwd.root -> unit
```

A root is created by *observing* a changing value with the observe function. It is important to call the release function when a root is no longer useful to prevent memory leaks.

sample evaluates the graph using as input the values of variables at the time of a call. Evaluation is lazy in the sense that computation does not happen when the graph is constructed but when sample is called. After sampling, if a variable changes, all derived values, from intermediate results up to the root are thrown away. The on_invalidate function, if it is was provided, is called with the previous value of the root. Derived values that depend on other variables are kept. A later call to sample will recompute just what is needed.

The strategy can be summarized as: eager invalidation, lazy evaluation.

3 Sample code

Just to get a feeling of what UI code can look like, here is a little snippet that displays a custom type of possibly infinite trees:

```
type tree = Tree of string * (unit -> tree list)

let rec tree_ui (Tree (lbl, child)) : ui Lwd.t =
  let opened = Lwd.var false in
  let render is_opened =
    let btn_text =
      if is_opened then "[-]␣" else "[+]␣" in
    let btn_action () =
      Lwd.set opened (not is_opened) in
    let btn = Widget.button
                (btn_text ^ lbl) btn_action in
    let layout node forest =
      Ui.join_y node
        (Ui.join_x (Ui.space 2 0) forest) in
    if is_opened
    then Lwd.map (layout btn) (forest_ui(child()))
    else Lwd.pure btn
  in
  Lwd.join (Lwd.map render (Lwd.get opened))

and forest_ui nodes = Lwd_utils.pack Ui.pack_y
                        (List.map ui_tree nodes)
```

Each node is printed on a separate line, indented and then prefixed with [+] or [-] to show or hide sub-trees. Most of the code is made of pure combinators, but each node allocates an opened variable. All dynamism comes from it: clicking the label changes its status, which is used to display children selectively.

4 Extra features

The library comes with usual widgets (though the API is still in development). But we have also developed a few less common features that proved convenient.

4.1 Dynamic collections

User interfaces often deal with a dynamic data source, with the number of items to be displayed not known in advance. For instance the content of a list box, a tree or a grid.

Fixed parts of a user interface can have their content updated without changing the shape of the computation. The Lwd graph stays the same. For dynamic parts, the graph has to be adjusted. Sub-graphs are regularly created or dropped.

Lwd offers two facilities for dealing with dynamic collections: Lwd_table for imperative collections and Lwd_seq for pure collections.

Both collections are built on the idea of transforming user-defined data using “map/reduce” functions. Given a collection whose elements have type elt:

- map applies a function of type elt -> result to each element
- reduce combines intermediate results by repeated applying a function of type result -> result -> result

For instance to display a collection of strings:

- map with Widget.string : string -> ui to transform strings to visual elements
- reduce with Ui.join_y : ui -> ui -> ui to concatenate visual elements vertically, producing a vertical list of strings.

Lwd_table represents a collection as a doubly-linked list. Each element is a node in this list. They are easy to move around, to remove and to insert at arbitrary places. The map function is applied once per element. The reduce function is applied whenever the list changes. Internally, a balanced structure is maintained to bound the depth of reductions to the logarithm of the number of nodes.

Lwd_seq is a bit more subtle. Its interface looks like:

```
type +'a seq
val empty : 'a seq
val element : 'a -> 'a seq
val concat : 'a seq -> 'a seq -> 'a seq
```

empty is the empty collection, element is a singleton collection and concat concatenate the elements from two collections, in order. The physical identity of a seq value is used to determine which map/reduce needs to be recomputed:

- new elements are mapped, new concats are reduced
- if element and concat nodes are reused, the results of map/reduce are reused too.

The incrementality comes from building new seq by sharing old and new values. The update algorithm takes a time proportional to the number of changes: the bookkeeping overhead does not affect the asymptotic complexity of the whole computation.

4.2 Lwt integration

Lwt is a popular OCaml library that implements monadic concurrency. It is very convenient for expressing asynchronous computations.

While Nottui and Lwd libraries are independent of Lwt they can work well together. `Nottui_lwt` is a thin layer that runs the main user interface loop as an `Lwt.t` computation.

An application in this style is structured as follow:

- Lwd evaluation and rendering is done synchronously (from the point of view of Lwt).
- Event dispatch is synchronous too but can spawn asynchronous computations.
- Asynchronous computations can change Lwd variables. Lwd graph invalidation schedules a new asynchronous computation to render a new frame.

4.3 Live pretty printing

`Nottui_pretty` is an adaptation of the Pottier & Pouillard / Leijen / Wadler [1998]lineage of pretty printers.

The core combinators are the same but the atomic primitives are Nottui elements rather than character strings. The pretty-printing algorithm has been made incremental to better suit interactive use.

Pretty-printed documents can include arbitrary widgets, even those that dynamically change their size, without significant loss of efficiency.

5 Applications

While Nottui and Lwd are still in their infancies, their development was started and shaped by the needs of *Citty*, a terminal client for a continuous integration system. Later it proved useful for other applications.

5.1 Citty: a client for OCamllabs continuous integration platform

Citty connects to instances of OCurrent, a continuous integration platform for OCaml projects in development by OCamllabs.

It is structured as a multi-pane interface:

- The left-most one list known repositories.
- When a repository is selected, the list of monitored references (in general git branches) opens in a new pane. This is shown in figure 1.
- When a reference is selected, the list of continous integration jobs (linting, build status on different platforms and OCaml versions, ...) associated to it opens in a new pane.

```
ci.ocamllabs.io
              0install/0install          refs/heads/master          #f76495
              CraigFe/oskel              refs/pull/144/head         #2f8676
              CraigFe/ppx_irmin          refs/pull/189/head         #80c470
              NathanReb/ppx_yojson
              aantron/lambdasoup
              avsm/duniverse
              avsm/ocaml-ctypes
              avsm/ocaml-yaml
              avsm/osrelease
              gpetiot/ocaml-weather
              gpetiot/parse-wyc
              gs0510/index
              kit-ty-kate/labrys
              mirage/alcotest
              mirage/arp
              mirage/bigarray-compat
              mirage/bloomf
              mirage/ca-certs
              mirage/canopy-data
              mirage/capnp-rpc
              mirage/charrua
              mirage/checkseum
              mirage/coin
```

Figure 1: Citty listing repositories and references of “mirage/capnp-rpc”.

- When a job is selected, the output log is displayed. If the job is still running, it can be cancelled. If the job has finished, it can be rebuilt (figure 2).
- Output log is displayed in an Nottui widget for displaying long word-wrapped text, but an extra action allows opening the log in system `$EDITOR`, for convenience.

Many Nottui/Lwd features are exercised in the application logic:

- Each pane is a collection (of repository, of references, of jobs and of log lines).
- All content is fetched from the network. Lwt is used to keep the interface responsive will waiting for answers.
- Logs are often too big to be fetched in a single request. Instead they are streamed. This is almost transparent to the application, display is done progressively without extra effort.

The interface can be navigated using keyboard and mouse, and large contents can be scrolled.

5.2 Imandra: visualizing internal proof state

Imandra is an interactive theorem prover that focuses on ease of use and powerful automation. Simon Cruanes started using Nottui internally to develop an experimental proof exploration UI, shown in figure 3 , and a tool to explore the content of a production database. Both use the current set of widgets extensively with unfoldable trees and scrollable lists of entries.

The proof exploration UI can be started directly from within an interactive Imandra session, taking over the

```
ci.ocamllabs.io
refs/heads/ma [✓] (analysis)            Job "2020-05-25/102750-ci-build-0d3eb1":
refs/pull/144 [✓] (lint-doc)              Description: test mirage/capnp-rpc f76495a48fc05376018fa8c35a364f4a3a07a
refs/pull/189 [✓] (lint-fmt)
              [X] alpine-3.11-ocaml-4. [Rebuild]
              [✓] centos-8-ocaml-4.10  ↳.5.1).
              [✓] debian-10-ocaml-4.07 #17 DONE 38.4s
              [✓] debian-10-ocaml-4.08
              [✓] debian-10-ocaml-4.09 #18 [stage-0 11/11] RUN ODOC_WARN_ERROR=true opam exec -- dune build @do↵
              [✓] debian-10-ocaml-4.10 ↳c || (echo "dune build @doc failed"; exit 2)
              [X] fedora-31-ocaml-4.10 #18 sha256:840ed474364f544c1a2f4280715fd60812a6c824eaa8fdd3d4de02b93b7d1↵
              [✓] opensuse-15.1-ocaml- ↳b65
              [✓] ubuntu-20.04-ocaml-4 #18 DONE 1.8s

                                       #19 exporting to image
                                       #19 sha256:e8c613e07b0b7ff33893b694f7759a10d42e180f2b4dc349fb57dc6b71dca↵
                                       ↳b00
                                       #19 exporting layers
                                       #19 exporting layers 9.9s done
                                       #19 writing image sha256:8db0f8424a4045ac70307edfc455ed20b336ddccb02c267↵
                                       ↳c1ed12b08c57800d7 done
                                       #19 DONE 10.0s
                                       2020-05-25 11:54.40: Job succeeded
```

Figure 2: Successful results of "lint-doc" target on current `HEAD`.

terminal, and gracefully restoring it back when "quit" is clicked. The session, which is a modified OCaml toplevel with some autocompletion, can then be resumed normally.

5.3 BetterBoy: Gameboy Emulator tooling

BetterBoy is a Gameboy Emulator implemented in OCaml by Enguerrand Decorne. The main renderer outputs to an SDL (graphical) window, though it can optionally be displayed in the terminal.

However, some tooling was developed to debug the emulator itself using Nottui for the interface. The main features are selected via tabs that switch between:

- A debugger shell with a prompt to input commands and a log of results mixing of textual and "graphical" (as far as the terminal can be used for graphics) contents (figure 4).
- A video ram viewer for Gameboy graphic chipset.
- An interactive disassembler for Gameboy CPU (figure 5).

6 Shortcomings and future work

While Lwd & Nottui have been very satisfying for the small applications we are working on, we encountered and worked around a few limitations.

Context. The main one is the general notion of "context dependence", a generalization of the "context probing" events. Example of widgets that depends heavily on their context are world maps. like Google Maps or OpenStreetMaps, or large spreadsheets.

Their content can be arbitrarily big and it is not feasible to make a concrete representation of it. To compute the actual content to display, it is necessary to know how much space is available for the widget. This goes in the opposite direction of the normal Nottui workflow: building a tree from the leaves, accumulating constraints to know how to properly display the resulting interface. Here we start from the root, splitting space according to layout rules, until we reach the Map widget. And only then can the widget actually compute its content.

This is not a pressing issue, but this is certainly a problem to address if we want the toolkit to be general purpose.

Sharing. Another minor issue has been sharing. The UI is built in a functional style with each part of the interface being directly manipulated as a value. In some places, it is tempting to reuse the same value, when possible. For instance, some decorations are reused in many places.

Sharing pure values is not a problem, but occasionally stateful components are shared too. The toolkit handles this situation without problem, but the observed behavior can be surprising. For instance, Nottui has a premade menu widget that opens a sub-menu popup when clicked. If the same instance is put in two different places of the UI, clicking one of them activates both at the same time, displaying two popups.

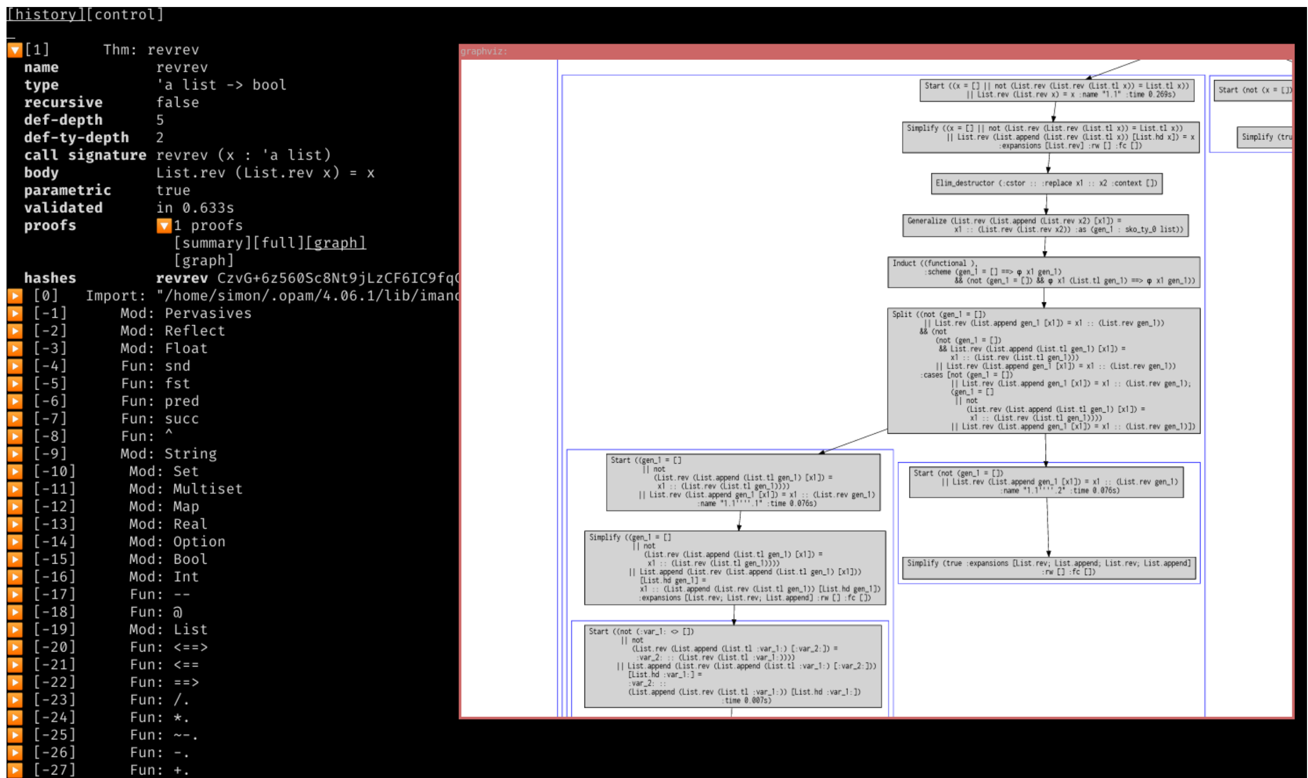

Figure 3: The imandra proof exploration UI.

This is a minor issue, but it would be nice to clarify the semantics of shared visual components and to provide means to prevent accidental sharing.

Batteries included. We are aiming to release the first stable version of Nottui & Lwd before the end of this year. For user convenience, we would like to have a well-defined set of standard widgets (button, edit field, lists, check box, menus, etc). This requires careful API design, to avoid regrettable decisions or breaking backward compatibility. In turn, this needs lots of testing to make sure we properly cover common cases.

This is the main blocking point for the first public release.

Web integration. We have tested Lwd in the Web browser, using Js_of_ocaml. It is already in a good enough shape to implement simple web applications. While there is no need for a low-level Nottui-like library as we can target the DOM directly, a common set of widgets is even more important.

Once the terminal version is released, we plan to provide an API as close as possible to Nottui's principles that targets web platforms. The goal is not to provide a drop-in alternative backend but rather to make it easy to transfer knowledge from one backend to the other.

Acknowledgements

Thanks to Simon Cruanes and Enguerrand Decorne for being early adopters, helping with the design of the libraries, and for code contribution. Thanks also to François Pottier, Arthur Wendling, and Gabriel Scherer for their valuable feedback.

References

Umut A. Acar. Self-adjusting computation. Technical report, In ACM SIGPLAN Workshop on ML, 2005.

Nelson H. F. Beebe. Using boxes and glue in tex and latex, 2009.

Ross Paterson Conor McBride. Applicative programming with effects. *Journal of Functional Programming*, 18, 2007.

Philip Wadler. A prettier printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.

Figure 4: Using BetterBoy shell to explore internal state

[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+)			

Figure 5: BetterBoy disassembler showing instructions around program counter.