Ocamlviz

Julien Robert and Guillaume Von Tokarski

November 11, 2026

Contents

1 Introduction **2**

2 Installation **2**
2.1 Prerequisites . . . 2
2.2 Compiling from sources . . . 2

3 User Manual **2**
3.1 Instrumenting User Code for Monitoring . . . 2
3.1.1 Module Point . . . 3
3.1.2 Module Time . . . 3
3.1.3 Module Tag . . . 3
3.1.4 Module Value . . . 4
3.1.5 Module Hashtable . . . 4
3.1.6 Module Tree . . . 5
3.1.7 Log . . . 5
3.1.8 Kill . . . 5
3.1.9 Wait_for_connected_clients & wait_for_killed_clients . . . 5
3.1.10 Automating Instrumentation using Camlp4 . . . 5
3.2 Linking with Ocamlviz . . . 6
3.3 Visualizing Monitoring Results . . . 6
3.3.1 GUI . . . 6
3.3.2 ASCII Client . . . 9

4 Developer Manual **10**
4.1 Source Files . . . 10
4.2 Protocol . . . 11
4.3 Binary Implementation of the Protocol . . . 11
4.3.1 Tag . . . 11
4.3.2 Kind . . . 11
4.3.3 Name . . . 12
4.3.4 List . . . 12
4.3.5 Value . . . 12
4.3.6 Command . . . 15
4.4 Architecture . . . 15

1 Introduction

Ocamlviz is a free software funded by Jane Street Capital within the framework of Jane Street Summer Project. It allows the monitoring of Objective Caml programs and values in real time by using the Ocamlviz library. Ocamlviz can also be used as a debugging tool.

2 Installation

2.1 Prerequisites

You need Objective Caml $\geq$ 3.10.0 to compile Ocamlviz. To compile the GUI, you also need Lablgtk2 $\geq$ 2.10.1-2 [1] and Libcairo-ocaml $\geq$ 20070908-1build1 [2]. (If one of these libraries is missing, the compilation will proceed but the GUI will not be compiled.)

To display trees within the GUI, you need Graphviz [3] to be installed. But Graphviz is not required to compile Ocamlviz.

2.2 Compiling from sources

Within Ocamlviz sources, configure with

```
./configure
```

Compile with

```
make
```

As superuser, install with

```
make install
```

3 User Manual

The documentation can be found at the following adress: `http://ocamlviz.lri.fr/doc/Monitor_sig.Monitor.html` or by compiling with the command "make doc" (the index of the documentation will be the file *index.html* inside the folder *doc*).

3.1 Instrumenting User Code for Monitoring

Two libraries are provided:

- Ocamlviz: This is the main library. It uses alarms to collect and send data. If the monitored program also uses alarms, open the Ocamlviz_threads library instead. Ocamlviz may not work properly in native compilation if the monitored program doesn't trigger the GC. If this happens, compile the program in byte code.
- Ocamlviz_threads: This is the library that should be used if the user program already uses alarms. Note that Objective Caml threads are not efficient and this solution is a patch.

You have to call `Ocamlviz.init` (or `Ocamlviz_threads.init`) in your code.

You may use `Ocamlviz.send_now ()` (or `Ocamlviz_threads.send_now ()`) to force a sending.

3.1.1 Module Point

This module is a check-point tool. When putting a Point.observe annotation, it will sum every time the program goes through this line of code. For instance:

```
let point1 = Point.create "observe f"
let _ = Point.observe point1
```

3.1.2 Module Time

This module is a chronograph tool. The timer that was created can be started and stopped. Note that a stopped timer can be restarted. For instance:

```
let timer1 = Time.create "time in function f"

let f () =
begin
Time.start timer1;

...

Time.stop timer1;
end

let g a b c d = a+b+c+d

let _ = (Time.time "time in function g" g) 1 2 3 4
```

3.1.3 Module Tag

The module Tag allows creating sets of Objective Caml data. It's possible to monitor the cardinal number and the size of these sets. The sets can contain any value of any type. For instance:

```
let tag = Tag.create ~size:true ~count:true ~period:1000 "tag example"

let x = Tag.mark tag (true::[])
let y = Tag.mark tag (6. +. 1., 6 + 1)
let z = Tag.mark tag "string"
```

The set *tag* contains these 3 elements. The size and the cardinal number of this set will be monitored. The period is in milliseconds. For each tag, Ocamlviz goes through its elements in the heap. The bigger the elements, the slower the program, so correctly adjust the period.

3.1.4 Module Value

This module allows the monitoring of values with the following Objective Caml types:

- integers
- floating point numbers
- booleans
- strings

For instance:

```
let f x = x *. 0.1
let _ = Value.observe_float_fct ~period:2000 "f 2." (fun () -> f 2.)

let s = "weak"
let _ = Value.observe_string "s" s
let _ = Value.observe_string_fct ~weak:true "fct_s" (fun () -> s)

let a = Value.observe_int_ref "a" (ref 0)

let b = ref true
let _ = Value.observe_float_ref "b" b
```

The argument `weak` means that the value can be attached to a weak pointer and garbage collected.

3.1.5 Module Hashtable

This module is meant to monitor Objective Caml hash tables. It monitors the:

- hash table length (number of elements inside the table)
- array length (number of entries of the table)
- number of empty buckets
- hash table filling rate
- longest bucket length
- mean bucket length

For instance:

```
let h = Hashtable.observe ~period:1000 "h" (Hashtbl.create 17)
```

Ocamlviz goes through the whole hash table in the heap. The bigger the table, the slower the program, so correctly adjust the period.

3.1.6 Module Tree

This module allows the monitoring of polymorphic variants, once they were changed into the following type:

```
type variant = Node of string * variant list
```

For instance:

```
let tree1 = (Protocol.Node ("1",[
                                  Protocol.Node ("1.1",[]);
                                  Protocol.Node ("1.2",[]);
                                 ]))

let _ = Tree.observe "tree1" (fun () -> tree1)
```

3.1.7 Log

This function builds a log and expands it. For each call, it will store the string along with its time.

```
let _ = log "%d This is how we use %s in %s" 1 "log" "ocamlviz";
        log "%f It is %b that log works like ocaml printf functions" 2. true
```

3.1.8 Kill

In some modules, there are functions called "killed". Calling this function will stop the monitoring of a data. This can be usefull if the data won't change anymore and if its monitoring costs a lot of ressources.

3.1.9 Wait_for_connected_clients & wait_for_killed_clients

OCAMLVIZ provides two functions to blocks the program execution:

- wait_for_connected_clients i: this hangs up the program execution until i clients are connected
- wait_for_killed_clients () : this hangs up the program execution until every clients are disconnected

3.1.10 Automating Instrumentation using Camlp4

It is possible to instrument automatically a file using `camlp4`. For this purpose, a pre-processor called `pa_ocamlviz` is provided. It is used as follows:

```
ocamlopt -c -pp "camlp4 pa_o.cmo str.cma pa_ocamlviz.cmo pr_o.cmo" source_file.ml
```

This will modify the following top-level instructions:

- References on integers, floating points, booleans, strings

- Hash tables
- Functions (time and calls monitoring)

If the data are visualized through the GUI for a file called "file", data' names will be "file_name". For example, a function "f" from a file "g.ml" will be displayed as "g_f".

3.2 Linking with Ocamlviz

To link the user code with Ocamlviz, use

```
ocamlc unix.cma libocamlviz.cma <your files>
```

in bytecode, and

```
ocamlopt unix.cmxa libocamlviz.cmxa <your files>
```

in native-code.

Note that `Ocamlviz.init` (or `Ocamlviz_threads.init`) must be called somewhere in the user code.

Once linked with Ocamlviz, the user code acts like a server. The default port used by this server is 51000. Another port can be specified using the `OCAMLVIZ_PORT` environment variable.

The server's default timer is 0.1 seconds, you can specify another timer by changing the `OCAMLVIZ_PERIOD` environment variable. We advise to keep a timer greater or equal than 0.1 seconds.

Calculing the size of living data in the heap can cost a lot of ressources and considerably affect the program execution. The computational complexity of this calculus is O(n), n being the number of blocks of the heap. The default period of this calculus is 1.0 second. You can specify another period by changing the `OCAMLVIZ_GC_PERIOD` environment variable. We advise to keep a period greater or equal than 0.1 seconds. NB: this doesn't affect the heap's total size, which is get according to server timer.

3.3 Visualizing Monitoring Results

Ocamlviz provides two clients to visualize the monitored data.

3.3.1 GUI

The GUI is launched with

```
ocamlviz-gui [options]
```

Command line options are

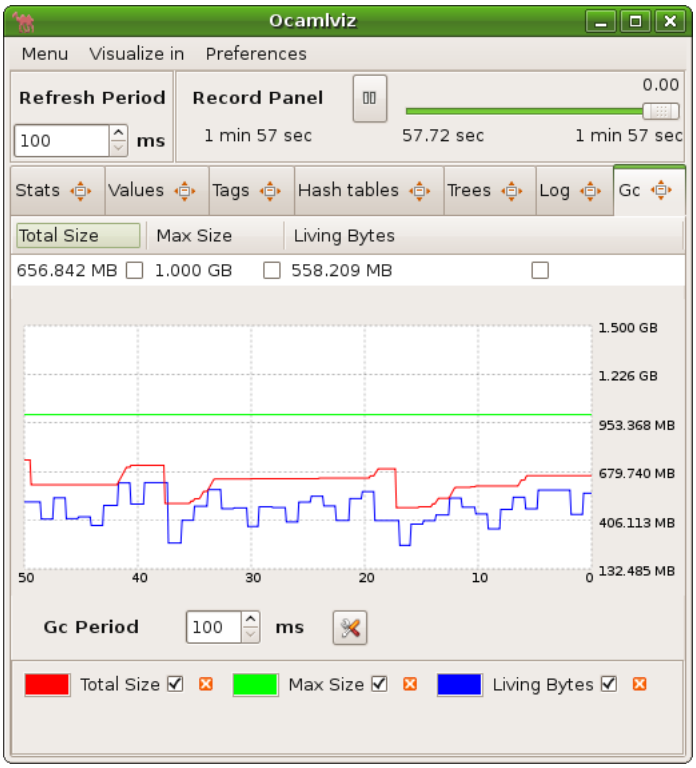
`-server` to specify the server machine (the default is the local host)

`-port` to specify the server port (the default value is 51000)

If no Ocamlviz server is running, the GUI fails with the error message

```
connection: couldn't connect to the server machine:port
```

Otherwise, it opens a main window which looks like:



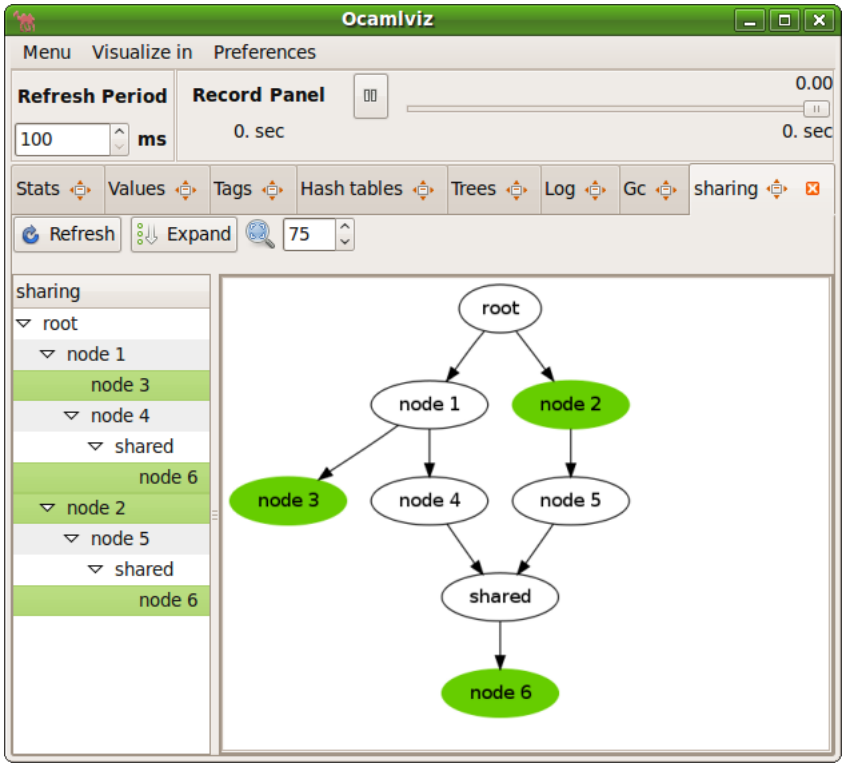
The data are displayed in a notebook, in the following pages:

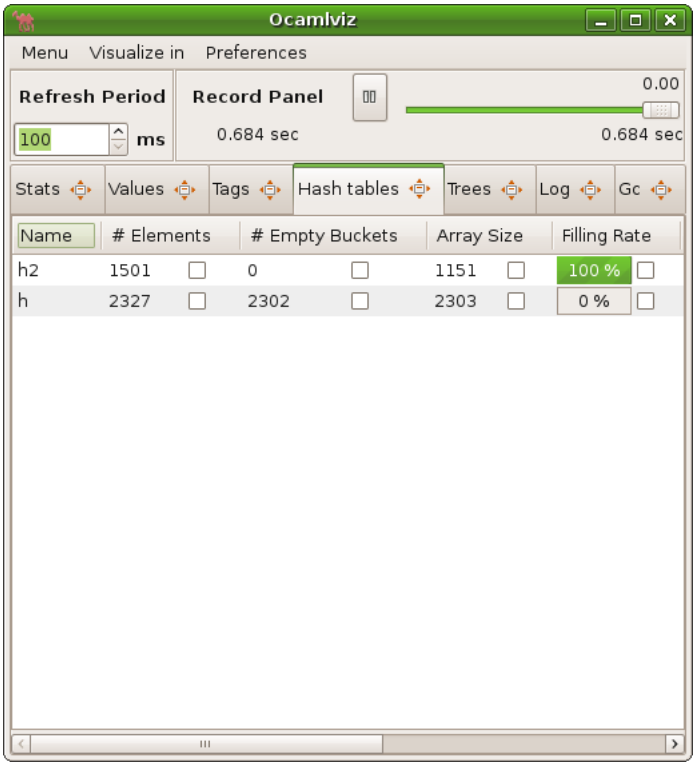
- Stats: displays `Point` and `Time`
- Values: displays `Value`
- Tags: displays `Tag`
- Hash tables: displays `Hashtable`
- Trees: displays `Tree`
- Log: displays the log
- Gc: displays the garbage collector informations about the size of the heap, the size of living data in the heap, along with their representation on a graph

Inside some cells, there is a second information which is the last time the data was modified. The color of the text can be red (value was killed) or green (value was garbage collected). Cells can also contain check boxes. These check boxes, once checked, allow to create graphs and lists in new pages or existing pages, through the menu "Visualize in"

or shortcuts. A list can contain any data, but a graph can only display data of the same type, representing integers, floating-points, percentages or bytes.

It is possible to pause the GUI and even to travel back in time through the record panel. The database will store one minute of data by default, but this can be changed in the menu preferences. The maximum window is one hour.







3.3.2 ASCII Client

This client logs the monitored data into a file.

The ASCII client is launched with

```
ocamlviz-ascii [options]
```

Command line options are

`-server` to specify the server machine (the default is the local host)

`-port` to specify the server port (the default value is 51000)

`-o` to specify the output file (the default value is `ascii.log`)

If no Ocamlviz server is running, the ASCII client fails with the error message

```
connection: couldn't connect to the server machine:port
```

4 Developer Manual

4.1 Source Files

- `ascii.ml`: this is the ASCII client, it writes monitored data into a file
- `binary.ml`: contains functions that code and decode several Objective Caml types in a buffer
- `bproto.ml`: contains the functions that code and decode the Ocamlviz messages (see protocol.mli)
- `db.ml`: the client database that stores the data and gives functions to acces them.
- `dot.ml`: contains functions that create dot files (graphviz) from a variant (see protocol.mli)
- `graph.ml`: a module that create a graph on a cairo canvas, and functions to manage the graph
- `gui_misc.ml`: contains miscellaneous functions for the GUI
- `gui.ml`: the main file of the GUI, containing the main and the functions to build the notebook and export data into graphs and pages
- `gui_models.ml`: contains the functions that create the models and refresh them
- `gui_pref.ml`: contains the functions that create the preferences dialog windows, and manage preferences
- `gui_view.ml`: contains the functions that create the views associated to the models (see gui_models.ml)
- `monitor_impl.ml`: contains the monitoring API
- `net.ml`: contains the client-side network

- `ocamlviz.ml`: includes monitor_impl.ml and contains the server for alarms
- `ocamlviz_threads.ml`: includes monitor_impl.ml and contains the server for threads
- `preflexer.mll`: parses the file called "preferences" (if it exists) to apply the user preferences
- `protocol.mli`: contains the protocol types
- `timemap.ml`: a module to store data in an array and retrieve them with a logarithmic complexity
- `tree_panel.ml`: contains the functions to create and display a tree container

4.2 Protocol

The protocol is made of three types of messages:

- Declare, to declare a new tag to a client
- Send, to send a tag's value (only after this tag was declared)
- Bind, to bind tags together (optionnal)

These 3 messages have the following structure:

command	arguments
`Declare`	tag, kind, name
`Send`	tag, value
`Bind`	tag list

4.3 Binary Implementation of the Protocol

4.3.1 Tag

A tag is an integer coded on 2 bytes.

4.3.2 Kind

Each kind is assigned to an integer. This integer is then coded on 1 byte.

	Kind
0	Point
1	Time
2	Value_int
3	Value_float
4	Value_bool
5	Value_string
6	Tag_count
7	Tag_size
8	Special
9	KTree
10	Hash
11	KLog

4.3.3 Name

A name is coded into two parts, the first part being the string's length on 4 bytes, and the second being the string itself on `length` bytes.

Bytes	4	n
Value	length (n)	contents

4.3.4 List

A list is coded into two parts, the first part being the list's length on 2 bytes, and the second being the elements. The way the elements are coded will depend on their types.

Bytes	2	?	...	?
Value	length	element #1	...	elements #n

4.3.5 Value

- Int

Bytes	1	4
Native Int 31	0	i

Bytes	1	8
Native Int 63	1	i

- Float

Bytes	1	8
Float	2	f

- String

`Bytes`	1	4	n
`String`	3	length (n)	s

- Bool

`Bytes`	1	1
`Bool`	4	b

- Int64

`Bytes`	1	8
`Int64`	5	i

- Collected

`Bytes`	1
`Collected`	6

- Killed

`Bytes`	1
`Killed`	7

- Tree

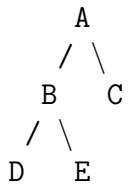
`Bytes`	1	1	?
`Tree`	8	# nodes	`Node List`

`Bytes`	4	length(s)	1	?
`Node` (s,l)	length(s)	s	length(l)	l (`Child List`)

`Bytes`	2
`Child`	index

This coding allows to keep the sharing.

Tree coding example:



Nodes	Value to code	Meaning
	8	Tree
	5	# nodes
0	1	length D
	D	
	0	0 child
1	1	length E
	E	
	0	0 child
2	1	length B
	B	
	2	2 children
	0	node 0
	1	node 1
3	1	length C
	C	
	0	0 child
4	1	length A
	A	
	2	2 children
	2	node 2
	3	node 3

- Hashtable

Bytes	1	4	4	4	4
Hashtable	9	# entries	# elements	# empty buckets	max bucket length

- Log

Bytes	1	?
Log	10	Float * String List

4.3.6 Command

`Declare`

`Bytes`	1	2	1	?
`Value`	0	tag	kind	string

`Send`

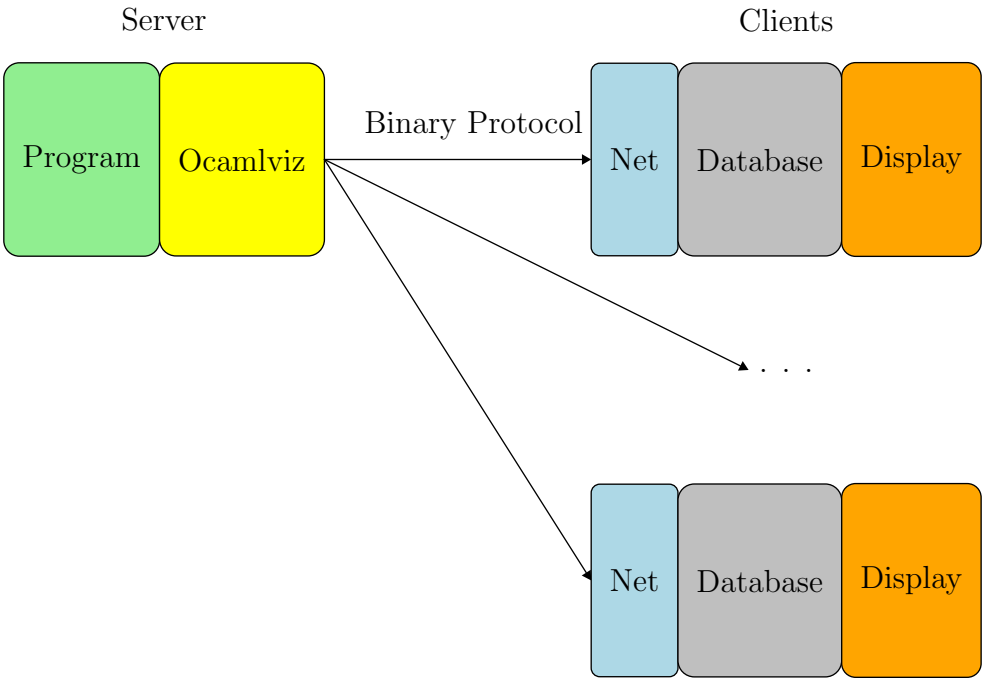
`Bytes`	1	2	?
`Value`	1	tag	value

`Bind`

`Bytes`	1	2
`Value`	1	tag

4.4 Architecture

This is the architecture of OCAMLVIZ. When a program is monitored, a server is created, sending the binary data on the network to its clients. Each client will decode every binary data and store them into its own database.



References

[1] Jacques Garrigue *Lablgtk*, an Objective Caml interface to Gtk+
`http://wwwfun.kurims.kyoto-u.ac.jp/soft/lsl/lablgtk.html`

[2] *Cairo*, a 2D graphics library with support for multiple output devices
`http://cairographics.org/cairo-ocaml/`

[3] *Graphviz*, an open source graph visualization software
`http://www.graphviz.org/`