

Ocamlviz

Jane Street Summer Project

Julien Robert, Guillaume Von Tokarski
Sylvain Conchon, Jean-Christophe Filliâtre, Fabrice Le Fessant

Gallium seminar
November 9, 2009



How do you tune/debug an OCAML program?

Ocamldebug

```
File Edit Options Buffers Tools Complete In/Out Signals Help

^ let make n =
  let rec makerec avl n =
    the_height := height avl;
    if n = 0 then
      avl
    else
      makerec (add (Random.int n) avl) (n-1)
  in
  makerec Empty n

--- avl.ml 77% L82 (Tuareg Abbrev)-----
^ Time : 627 - pc : 142144 - module Avl
(ocd)
Time : 628 - pc : 142172 - module Avl
(ocd) print n
n : int = 1999998
(ocd) print avl
avl : int t = Node (Empty, 1618234, Node (Empty, 1824681, Empty, 1), 2)
(ocd)

1:** *camldebug-avl.byte* Bot L86 (Caml-Debugger:run)-----
```

Records execution counts for specified parts of the program (if then else, match, functions, ...)

```
let to_list t =  
  (* 589 *) let len = Array.length t in  
  let rec loop acc i =  
    (* 174006503 *) if i = len then (* 589 *) acc  
    else (* 174005914 *) loop (let v = t.(i) in if v >= 1  
                             then (* 142292 *) (i, v) :: acc  
                             else (* 173863622 *) acc) (i+1)  
  in  
  loop [] 0
```

The UNIX profiler: monitors function calls and the time spent inside them

```
Flat profile:
Each sample counts as 0.01 seconds.
%   cumulative   self           calls   self        total   name
time seconds    seconds             s/call       s/call
15.18      1.58      1.58 229988967      0.00      0.00  compare_val
11.91      2.82      1.24 115001981      0.00      0.00  camlMemo1__mark_120
10.09      3.87      1.05 114985760      0.00      0.00  caml_greaterequal
 7.59      4.66      0.79 229988965      0.00      0.00  caml_c_call
 7.49      5.44      0.78 114985530      0.00      0.00  caml_tuplify2
 7.40      6.21      0.77 114985530      0.00      0.00  camlMemo1__code_begin
 6.77      6.92      0.70 115002161      0.00      0.00  caml_lessequal
 5.48      7.49      0.57 114985760      0.00      0.00  camlPervasives__max_53
 5.09      8.02      0.53  101289      0.00      0.00  camlList__iter_102
 4.47      8.48      0.47 81838005      0.00      0.00  caml_initialize
 3.84      8.88      0.40      732      0.00      0.01  camlMemo1__iteri_97
 3.75      9.27      0.39      589      0.00      0.00  camlMemo1__loop_105
 2.98      9.58      0.31 115002161      0.00      0.00  camlPervasives__min_50
 2.83      9.88      0.29      332      0.00      0.00  caml_greaterthan
 2.11     10.10      0.22      332      0.00      0.00  caml_make_vect
 0.96     10.20      0.10      166      0.00      0.00  compare_stack_overflow
 0.86     10.29      0.09      1158      0.00      0.00  mark_slice
 0.43     10.33      0.04      277      0.00      0.00  caml_modify
 0.29     10.36      0.03      99537      0.00      0.00  caml_add_to_heap
 0.19     10.38      0.02      5      0.00      0.00  caml_fl_allocate
 0.19     10.40      0.02      5      0.00      0.00  caml_adjust_gc_speed
 0.10     10.41      0.01      5      0.00      0.00  caml_lessthan
```

even in real-time with OProfile

Drawbacks

Debugging tools

- no real-time analysis
- no nice display

Profiling tools

- usually after the program termination
- no memory profiling
- limited observations
- `ocamlprof` only works with bytecode, `gprof` with native-code

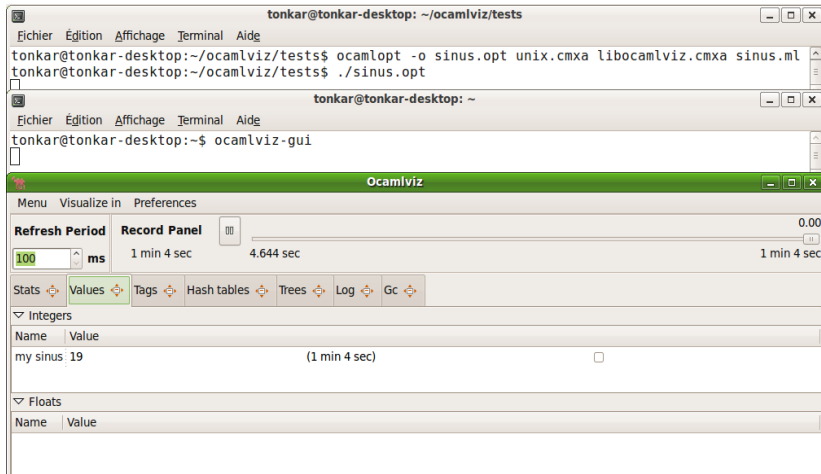
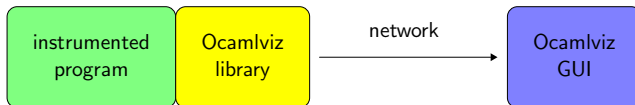
Ocamlviz intends to fill this gap

it provides

- real-time monitoring
- fine grain and customized observations
- a nice GUI to display results
- remote monitoring, at any time

This talk

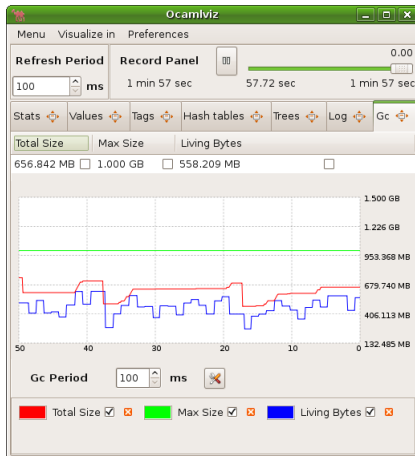
- ① overview of OCAMLVIZ
- ② some implementation details



A First Glance at OCAMLVIZ

Just by linking OCAMLVIZ to
your code, you gain access to
GC information

```
Ocamlviz.init ()
```



Program Points & Time Measuring

OCAMLVIZ provides 2 modules to monitor program points and durations

Points:

```
let p =  
    Ocamlviz.Point.create "p"  
  
let f x =  
    Ocamlviz.Point.observe p;  
    g x
```

Stats	Values	Tags	Hash tables	Trees	Log	Gc
Name	Count	Time	Overall Time			
p	803800206	40.84 sec				
t		40.946559	40.94 sec	100. %		

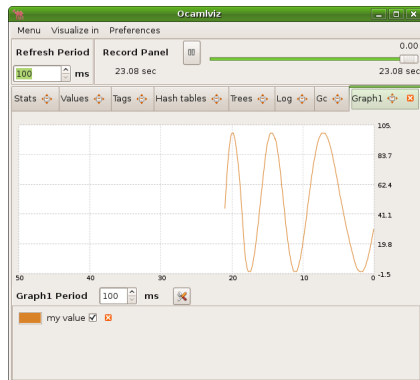
Durations:

```
let t =  
    Ocamlviz.Time.create "t"  
  
let f () =  
    ...  
    Ocamlviz.Time.start t;  
    ...  
  
let g () =  
    ...  
    Ocamlviz.Time.stop t;  
    ...
```

Observing Values

```
Ocamlviz.Value.observe_float_fct  
  "my value"  
  (fun () -> mysine !v)
```

```
Ocamlviz.Value.observe_float_ref  
  "v" v
```



similar features to observe integers, booleans and strings

Complex Values

Complex data types can be turned into **our tree type**, and then observed

Requires some work on the programmer side:

```
type t = A of string | B of t*t

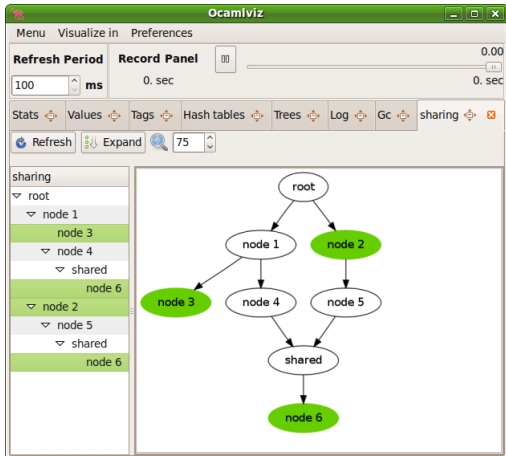
let node (x,y) = { Protocol.variant_name = x;
                  Protocol.variant_children = y }

let rec transform = function
  | A(x) -> node (x,[])
  | B(x,y) -> node ("B",[transform x ; transform y])
```

Then observe:

```
let _ = Ocamlviz.Tree.observe
  "my value" (fun () -> transform value)
```

Complex Values: Sharing



Memory Profiling

OCAML values can be tagged to profile memory

```
let t = Tag.create ~size:true ~period:300 "foo"  
let l = Tag.mark t  
      [Random.float 10.; Random.float 45.]  
let x = Tag.mark t (f ())  
let tuple = Tag.mark t (1,true,ref [x])
```

Stats	Values	Tags	Hash tables	Trees	Log	Gc
Name	Count	Max Count	Size	Max Size	Overall Size	
foo	6032 6.900 sec <input type="checkbox"/>	6032 6.900 sec <input type="checkbox"/>	289.560 KB 6.900 sec <input type="checkbox"/>	289.560 KB 6.900 sec <input type="checkbox"/>	0.03 %	<input type="checkbox"/>

for each tag, the total number of values and the total space can be monitored

Hash Tables

One instruction to collect several informations about a major OCAML data structure

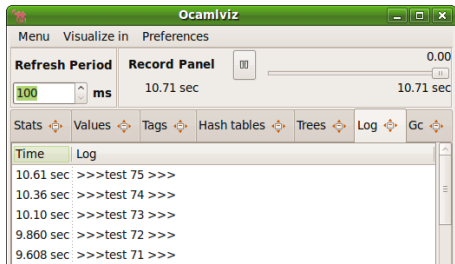
```
let h =  
  Ocamlviz.Hashtable.observe "h"  
  (Hashtbl.create 17)
```

Stats	Values	Tags	Hash tables	Trees	Log	Gc					
Name	# Elements	# Empty Buckets	# Entries	Filling Rate	Longest Bucket	Buckets Mean Length	Last Modified				
h2	10867	7372	9215	20 %	6	5.8963646229	(1.900 sec)				
h	18526	16431	18431	10 %	21	9.263	(3.196 sec)				

Logging Messages

A printf-like function with timestamp logs

```
let f i =  
  Ocamlviz.log  
  ">>> test %d >>>" i;  
  ...
```



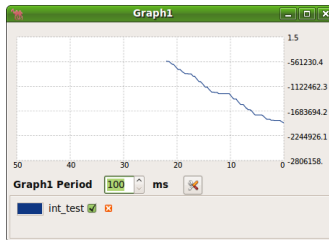
GUI Additional Features

- recording mechanism, pause and backward/forward features



- select values and create real-time customizable plots

Name	Value
int_test	-740074 (6.204 sec) <input checked="" type="checkbox"/>

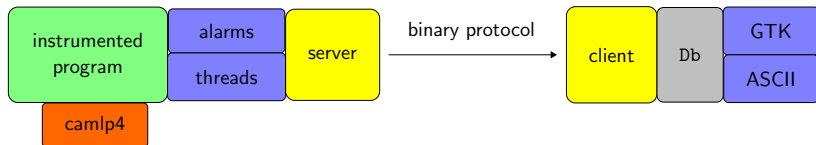


With reasonable server settings:

- program linked with OCAMLVIZ :
 - no client : $\leq +1\%$
 - one connected client (remote or not) : $\simeq +6\%$
- program reasonably instrumented with OCAMLVIZ:
 - no client : $\simeq +2\%$
 - one connected client (remote or not) : $\simeq +12\%$

- ① overview of OCAMLVIZ
- ② some implementation details
 - communication protocol
 - server implementation
 - observation techniques on the server side
 - storing data on the client side

A Modular Client-Server Architecture



- binary protocol to save bandwidth (documented)
- architecture independent, 32/64 bits compatible
- Ocaml independent

as a consequence

- servers could be written for other languages, and reuse our GUI
- clients could be written in other languages

Protocol Implementation

Protocol Messages

```
type msg =  
  | Declare of uid * kind * string (* New Observation *)  
  | Send of uid * value              (* Update Observation *)  
  | Bind of uid list                 (* Linked Values *)
```

Binary Protocol

Simple values (32/64bits, little-endian/big-endian):

```
val get_ttype : string → int → ttype × int  
val buf_ttype : Buffer.t → ttype → unit
```

Size-prefixed messages:

- Parsing only when message completely received
- Small messages reuse small 64k buffer (fewer allocs)

Alarms vs Threads

OCAMLVIZ provides two libraries to collect and send data

- `Ocamlviz`: an **alarm-based** solution
 - ⇒ OCAMLVIZ handler may conflict with a user's handler since only one handler is allowed per alarm
- `Ocamlviz_threads`: a **thread-based** alternative for programs using alarms
 - ⇒ context switch for threads seems to be less efficient than for alarms

fairness issue

- signal handling or threads scheduling can only occur at allocations or i/o operations
- the `Ocamlviz.yield` function can be used to enforce OCAMLVIZ's alarm handler or observation thread to be executed

Observation Techniques

- collecting data for program points and timers is easy
- collecting data for tagged values is more difficult
 - we need to count the number of **live** values
 - we need to count the total size of these values

Observation Techniques

tagged values are stored in **weak hash tables**

```
module WeakHash = Weak.Make(struct
  type t = Obj.t
  let hash = Hashtbl.hash
  let equal = (==)
end)
```

but this solution is incorrect (and inefficient)

```
# let t = WeakHash.create 17;;
val t : WeakHash.t = <abstr>
# let v = Obj.repr [1];;
val v : Obj.t = <abstr>
# WeakHash.add t v;;
- : unit = ()
# WeakHash.mem t v;;
_ : bool = false
```

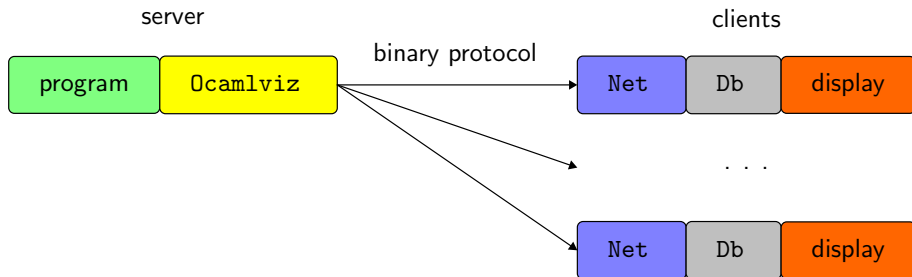
Observations Techniques

`WeakHash.mem t v` compares `v` to **copies** of the values pointed to by the weak pointers of `t`

solution: outermost blocks are duplicated and inner pointers are shared

```
let copy_equal x y =  
  Obj.is_block x && Obj.is_block y &&  
  let len = Obj.size x in  
  Obj.size y = len &&  
  let rec loop i =  
    i = len || Obj.field x i == Obj.field y i && loop (i+1)  
  in  
  loop 0
```

The Client Implementation



- the module `Net` connects to the server, reads and decodes messages
- the module `Db` stores data into an internal database

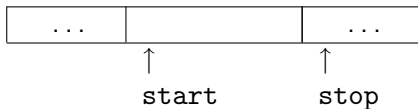
- keeps all data within a specified time windows
- allows requests for any execution time within this frame (and thus makes the time-shifting feature easy)

based on a time-indexed dictionary structure Timemap:

```
type 'a t
val create : ?size:int -> 'a -> 'a t
val add : 'a t -> float -> 'a -> unit
val find : 'a t -> float -> float * 'a
val remove_before : 'a t -> float -> unit
```

Timemap Implementation

- values stored consecutively in two arrays (times and values), used circularly



- insertion in constant time, unless we have to resize the array (we still have *amortized* constant time)
- find uses a binary search (logarithmic)
- remove_before searches for the right position and shifts the start index (logarithmic)

Conclusion

- OCAMLVIZ provides many features to profile and debug
- OCAMLVIZ is modular and can be easily extended
- OCAMLVIZ does a lot more than the existing tools

<http://ocamlviz.forge.ocamlcore.org/>

possible extensions :

- information about the time spent in the GC
- break points
- bi-directional communication

Acknowledgments

the authors are grateful to Jane Street Capital for funding this project