Slide 1

A Portable Implementation for Objective Caml Flight

Emmanuel Chailloux
`http://www.pps.jussieu.fr/~emmanuel`
Equipe PPS (CNRS UMR 7126) - University Paris 6 - France
and
Christian Foisy
`Digital Fountain, Fremont, CA, USA`

2nd HLPP - June 2003 - Paris - France

Slide 2

SUMMARY

1. Objective Caml in a few words
2. Caml-Flight
3. Objective Caml Flight = Objective Caml + Caml-Flight
4. Conclusion and future work

Slide 3

Objective Caml in practice

- One of the most popular ML dialect:
  - efficient code,
  - large set of general purpose and domain specific libraries,
  - automatic memory management,
  - used both for teaching (academy) and for writing high-tech applications (industry)
- Product of research results since 80's in: type theory, language design and implementation.
- Developped at INRIA (France).

Slide 4

Objective Caml features

- Functional language + imperative extension,
- High-level datatypes + pattern-matching,
- Polymorphic + *implicit* typing:
  - strongly and static typed,
  - types are inferred,
  - types are polymorphic (the most general ones).
- Different programming styles (in a common typing framework):
  - Class based object oriented programming,
  - High-level modules (SML style)
  - *More recently*: labels and variants added.

Slide 5

A Small Example (1)

```
# let rec map f l =
  if (l == []) then []
  else (f   (List.hd l)) :: (map f (List.tl l));;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

`map` : $\forall \alpha\beta.(\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$

```
# ( map (fun x -> x + 1) [1;2;3],
    map (fun x -> not x) [true; false]);;
- : int list * bool list = ([2; 3; 4], [false; true])
```

Slide 6

A Small Example (2)

```
let rec map f l =
  match l with
    [] -> []
  | h::t -> (f h) :: (map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Slide 7

Caml-Flight(1)

- based on the SPMD programming model;
- data-parallel extension for Caml;
- explicit parallelism;
- preserves the property of determinism;

First implementation in 1994 (Foisy's thesis).

Slide 8

Caml-Flight program

- runs n copies of itself
- with a fixed numbers of processes

All processes are created at the beginning of the computation and remain active until the end.

Each copy is parameterized by its local address and knows the total number of processes.

It introduces only two operations : an operation of synchronization (`sync`) and an operation for communications (`get`).

Slide 9

Syntax Extension

```
Expr ::= Simple_expr
      ...
      | sync Expr
      | get  Expr from Expr

Simple_expr ::= ...
      | local | nodes
```

- 2 constants : `local` and `nodes`
- `sync` : synchronization block
- `get e from i` : a request for a remote computation of `e` at `i`.

Slide 10

Typechecking

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash sync(e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : int\, \Gamma \vdash e_2 : \tau \;\; V(\tau) = \phi}{\Gamma \vdash get(e_1, e_2) : \tau}$$

$$\vdash local : int$$

$$\vdash nodes : int$$

- The two constants `local` and `nodes` are integers.
- `sync(e)` has type of `e`
- `get e2 from e1` has type of e_2 (monomorphic $V(\tau) = \phi$)

Slide 11

Synchronization Block, Communication Environment

`sync(e)` :

- creates a communication environment (CE)
- allows in its scope distant computations from the same `sync`
- no variable declaration inside, no abstraction
- no nested `sync`

Slide 12

Requests

`get e from i` :

- request for a distant computation of `e` at processor i
- must be inside scope of a `sync`
- no scope extension for a `get` outside a `sync`
- waits for the result : a value or a remote exception which will raise locally

---

wave :

- w-th encountered block of synchronization

Slide 13

Example 1

```
open Flight;;

let app_array f v =
  let len  = Array.length v in
  let step = ref 0 in
  let index = ref (!step + local) in
     while (!index  < len) do
       v.(!index) <- f (v.(!index));
       step := !step + nodes;
       index := !step + local
     done;;
```

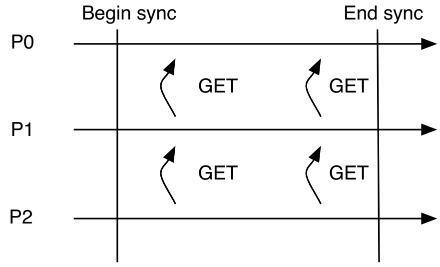
no communication!!!

Slide 14

Example 2

```
open Flight;;
let f x =
  sync (
    if local == 0 then 0
    else (get x + 1
           from (local - 1))
       + (get x + 2
           from (local - 1))
  );;

(f local);;
close_Flight();;
```



Slide 15

Example 3

```
let f x = sync ( if local <> 0 then
                   get x from (local -1)
                 else 0);;

let g x = sync ( if local <> 0 then
                   get x from (local -1)
                 else 0);;

let main() =
 if local == 0 then  f 0 else g local;;

main();;
```

Process 0 and process 1 are not inside the same wave :
⇒ no communication

Slide 16

Asynchronicity Depth

maximum distance, in wave, between the slowest and fastest process :

- same value for all processes,
- maximum frozen CE : $p + 1$,
- pipeline between processes

Spatial Recursion

- allows nested same `sync`
- stays at the same wave
- communication environment is not modified

Slide 17

Example 4

```
let rec scanf (v: int list) =
  sync( if local == 0
        then [v]
        else v::(get (scanf v) from (local-1)))
;;
```

Slide 18

Portable Implementation for Objective Caml

using :

- `camlp4` : to extend the grammar;
- `threads` library : to execute requests and current evaluation;
- `unix` library : to communicate requests and results;
- `Marshal` module : to freeze CE and to transfert structured values;

Slide 19

Implementation Design

- *nodes* program instances
- each one has at least two threads :
  - a server which can receive requests from intances
  - current evaluation
- each received request starts a new thread
- each program instance has an IP address and an unique port
- each program instance knwows all couples (IP adr, port)

Slide 20

Implementation Design

By program transformation : an Objective Caml Flight program will be rewritten to an Objective Caml program.

`sync` transforamtion :

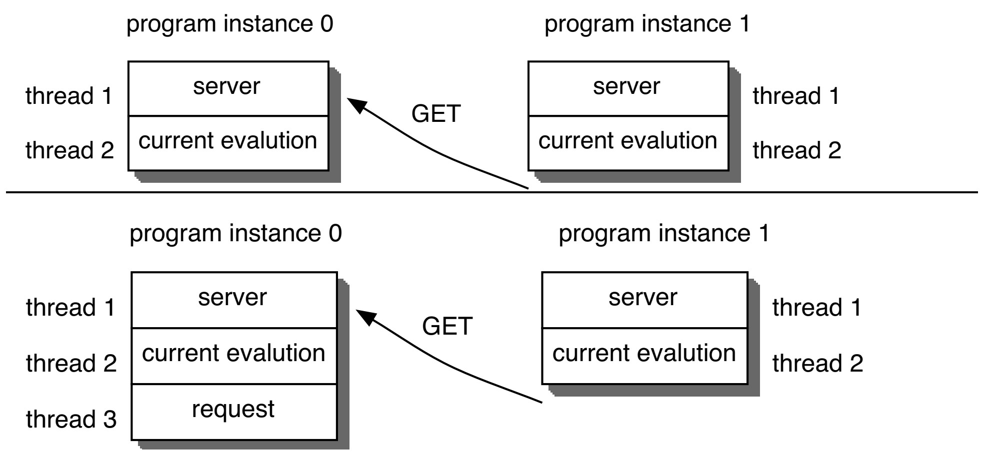
- $[sync(e)] \rightarrow begin\_sync(n, ce)$; $[e]$; $end\_sync(n)$
- $[get\ e\ from\ i \rightarrow drequest\ from\ to\ n\_get\ n\_sync\ n\_wave$

where *ce* contains closures corresponding to the `get` expressions :

```
fun () -> e
```

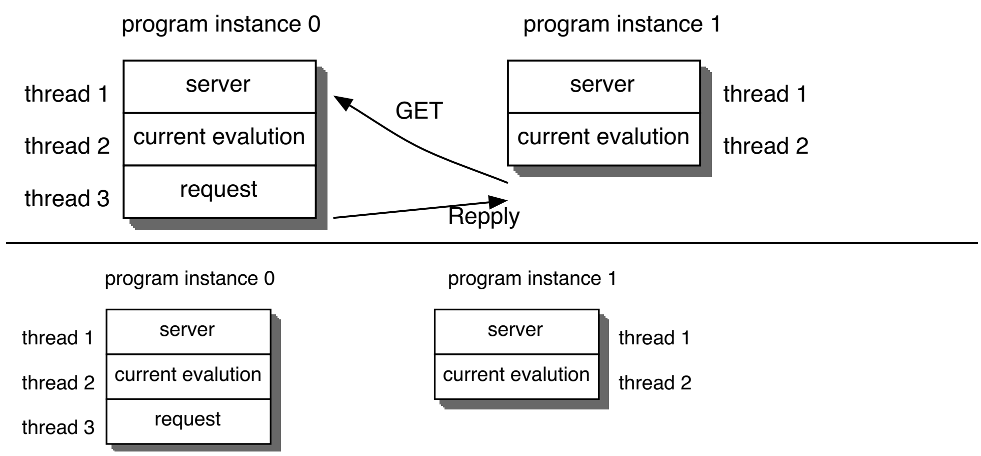
Slide 21

Sending a request



Slide 22

Sending a result



Slide 23

Flight module

- defines `local` and `nodes`
- using communications values

  `type 'a valeur = Valeur of 'a | Exn of exn;;`

  a remote exception will be raised locally.
- manages some global variables : waves, current sync number, ...
- and defines several usefull functions :
  - `close_Flight` : to wait for the end of all processes
  - `begin_sync` : is called when evaluation enters inside a `sync`
  - `save_closures` i: to freeze closures corresponding to `gets`
  - `end_sync` : when process exits from a `sync`
  - `drequest` : runned when a request is needed.

Slide 24

Syntax Extension (1)

`pa_ocf.ml`: `camlp4` parser file

```
expr :
  [ [ "sync"; "("; c = sync_expr; ")" ->
      incr num_sync;
      num_get := 0; ...
```

where "sync_expr" are :

- basic operators, apply
- control structure (sequence, if)
- structured values (records, arrays) for access
- and `get`

Slide 25

Syntax Extension (2)

```
[ "get"; e = sync_expr; "from";  n = sync_expr ->
     begin
       incr num_get;
       let ne = <:expr< Flight.save_closure
         (Marshal.to_string (fun() -> $e$) [Marshal.Closures]) >>
       in
         env := !env @ [ne];
         <:expr< if $n$ == Flight.local then $e$ else
         Flight.drequest $n$ $int:string_of_int !num_get$
         >>
     end
```

Slide 26

Translation Example

```
let f x = Flight.env_buffer := []; begin
    Flight.save_closure
      (Marshal.to_string (fun () -> x + 1) [Marshal.Closures]);
    Flight.save_closure
      (Marshal.to_string (fun () -> x + 2) [Marshal.Closures]);
    let ___fun_sync () =
      if Flight.local == 0 then 0
      else
        (if Flight.local - 1 == Flight.local then x + 1
         else Flight.drequest (Flight.local - 1) 1) +
          (if Flight.local - 1 == Flight.local then x + 2
           else Flight.drequest (Flight.local - 1) 2)
    in
    match Flight.sync_type.(0), Flight.sync_type.(1) with
      0, 0 ->
        Flight.begin_sync 1;
        let ___res_sync =
          try ___fun_sync () with
            Flight.RemoteExn e -> raise e
          | e -> raise e
        in
        Flight.end_sync (); ___res_sync
    | ___s, ___g ->
        if ___s == 1 && ___g == 0 || ___g == 1 then ___fun_sync ()
        else raise Flight.NestedSync
  end
```

```
open Flight;;
let f x =
  sync (
    if local == 0 then 0
    else (get x + 1
           from (local - 1))
       + (get x + 2
           from (local - 1))
  );;

(f local);;
close_Flight();;
```

Slide 27

Monitor

```
% ./mon
Usage: mon filename port depth machine_1 ... machine_n
```

- `mon` sends on `machine_i` a copy of `filename` with its own port number : $port + 1 + i$
- each new process answers to `mon` when it's ready
- when all are ready, `mon` indicates it to all processes and parallel program can start.

Each procces has a unique couple (IP address, port number) :
⇒ different copies can run on the same machine.

Slide 28

Limitations

- monomorphic `get` : unsafe
- no nested sync : dynamic checking
- no new variables introduction inside a `sync`

But this new version allows side effects inside CE

⇒ and using arrays as data structures.

Slide 29

Benchmarks

Intel	fib_l	wc_seek	life
Linux	1 wave	1 wave	30 waves
Objective Caml byte-code	6.7	8.8	12.8
1/1/0	9.8 **0.7**	9.4 **0.93**	13.4 **0.95**
2/2/0	7.2 **0.93**	6 **1.4**	7.6 **1.7**
3/3/0	7 **0.96**	5.2 **1.7**	5.8 **2.2**
4/4/0	6 **1.12**	4.8 **1.8**	5.1 **2.5**
5/5/0	5.7 **1.18**	4.7 **1.9**	4.7 **2.7**

speedup factor depends of :

- sequential computation / number of distant requests
- communications / number of waves

Slide 30

Map template

Working on arrays: to simplify we suppose `len mod nodes = 0`

```
let check a =
  let la = Array.length a in
    la mod nodes == 0;;

let map_seq f a e =
  let r = Array.create (Array.length a) e in
  for i=0 + (local*nodes) to (local+1)*nodes -1 do
    r.(i) <- f (a.(i))
  done;
  r;;

let map f a e1 all scan  =
  if check a then
    let v = map_seq f a e1 in
    let v2 = scan v all in v2
  else failwith "map";;
```

Slide 31

Different scans

1. spatial recursion : 1 wave, *nodes* communications
2. naive scan : *nodes* waves, *nodes* communications
3. optimized scan : $log_2(nodes)$ waves, *nodes* communications

Slide 32

Naive scan : *nodes* waves

```
let naive_scan a all  =
  let len = Array.length a in
  let shift = len / nodes in
  let r =  Array.create len  a.(0) in
  let step = ref 1 in
   while (!step < nodes) do
     sync (
       if local == 0 then
    Array.blit  (get a from !step) (nodes*shift) r (nodes*shift) shift
     );
   incr step
   done;
   if all then sync( if local <> 0 then get r from 0 else r)
   else r;;
```

Slide 33

An optimized version

$nodes = 2^n$

```
let scan2 a all =
  let len = Array.length a in
  let shift =  (len / nodes) in
  let npass = int_of_float (sqrt (float_of_int nodes)) in
  let r =  Array.create len  a.(0) in
  let step = ref 1 in
   while (!step < nodes) do
     sync (
       if local mod ( 2 * !step ) = 0 then
          Array.blit  (get a from (local + !step)) (local * !step *shift)
                      r (local * !step *shift)  (!step * shift)
     );
   step := !step * 2
   done;
   if all then sync( if local <> 0 then get r from 0 else r)
   else r;;
```

Slide 34

Conclusion

- portable implementation
- pedagogical tool

Future work

- to optimize communications
- to define some templates (as map)
- to integrate classes and functors
- to write real applications using these templates