Multi-runtime OCaml

Luca Saiu
<luca.saiu@inria.fr>

Inria, Gallium team

Conservatoire national des arts et métiers
Paris, 2013-06-24

Introducing myself

Hello, I'm Luca Saiu, Inria postdoc.

- Master's Degree at the University of Pisa;
- PhD at Université Paris 13
  - GNU epsilon (extensible programming language);
    - it includes a parallel GC, by the way
  - co-wrote Marionnet (GUI network simulator, in OCaml).

I work at Inria Saclay under the supervision of Fabrice Le Fessant, who started the multi-runtime project.

The problem

OCaml should exploit multicore machines for parallel computation.

However, the memory subsystem is a bottleneck:

- hardware: the memory wall;
- software: OCaml's GC is sequential, yet difficult to replace:
  - *very fast* and low-latency;
  - static property proofs.

The problem

OCaml should exploit multicore machines for parallel computation.

However, the memory subsystem is a bottleneck:

- hardware: the memory wall;
- software: OCaml's GC is sequential, yet difficult to replace:
  - *very fast* and low-latency;
  - static property proofs.

The problem

OCaml should exploit multicore machines for parallel computation.

However, the memory subsystem is a bottleneck:

- hardware: the memory wall;
- software: OCaml's GC is sequential, yet difficult to replace:
  - *very fast* and low-latency;
  - static property proofs.

General idea

If we have *n* cores, run *n* independent copies of the runtime, one per core:

- each individual runtime still sequential;
  - also keep non-parallel OCaml threads;
- coordination by message passing;
  - (and occasionally by shared memory)
- don't stop the world.

Alter the runtime logic as little as possible.

Threads or processes?

What's a "runtime", to the operating system? Two possibilities:

- an OS process:
  - less portable;
  - difficult to share memory;
- an OS thread:
  - portable with a simple layer;
  - easy to selectively share memory;
  - (hard to debug).

Threads or processes?

What's a "runtime", to the operating system? Two possibilities:

- an OS process:
  - less portable;
  - difficult to share memory;
- an OS thread:
  - portable with a simple layer;
  - easy to selectively share memory;
  - (hard to debug).

Threads or processes?

What's a "runtime", to the operating system? Two possibilities:

- an OS process:
  - less portable;
  - difficult to share memory;
- an OS thread:
  - portable with a simple layer;
  - easy to selectively share memory;
  - (hard to debug).

Threads or processes?

What's a "runtime", to the operating system? Two possibilities:

- an OS process:
  - less portable;
  - difficult to share memory;
- an OS thread:
  - portable with a simple layer;
  - easy to selectively share memory;
  - (hard to debug).

Required changes — 1 / 5

Very extensive (but regular) changes in the C runtime:

- global variables become fields of a large record, `struct caml_global_context`;
- pass context pointer around;
    - one more parameter to most C functions.

Required changes — 2 / 5

C changes: before...

```
void caml_process_pending_signals(void){
  int i;

  if (caml_signals_are_pending) {
    caml_signals_are_pending = 0;
    for (i = 0; i < NSIG; i++) {
      if (caml_pending_signals[i]) {
        caml_pending_signals[i] = 0;
        caml_execute_signal(i, 0);
      }
    }
  }
}
```

Required changes — 3 / 5

C changes: after...

```
void caml_process_pending_signals_r(CAML_R){
  int i;

  if (caml_signals_are_pending) {
    caml_signals_are_pending = 0;
    for (i = 0; i < NSIG; i++) {
      if (caml_pending_signals[i]) {
        caml_pending_signals[i] = 0;
        caml_execute_signal_r(ctx, i, 0);
      }
    }
  }
}
```

Required changes — 4 / 5

From context.h:

```
#define CAML_R \
  caml_global_context *ctx

#define INIT_CAML_R \
  CAML_R = caml_get_thread_local_context() /* uses TLS */

#define caml_signals_are_pending \
  ctx->caml_signals_are_pending
#define caml_pending_signals \
  ctx->caml_pending_signals
```

Required changes — 5 / 5

Some changes in the assembly part:

- reserve one register to hold the context pointer;
  - (save/restore it when needed)

Very minor changes in the compiler:

- global OCaml variables are now "contextual"
  - one dynamic array holding them all

Required changes — 5 / 5

Some changes in the assembly part:

- reserve one register to hold the context pointer;
  - (save/restore it when needed)

Very minor changes in the compiler:

- global OCaml variables are now "contextual"
  - one dynamic array holding them all

Consequences of this architecture

Pros:

- keep OCaml's GC;
- scalability: the sequential GC isn't a bottleneck;
- generalization from multicores to networks;
- link two ore more libraries with C interface using OCaml internally.

Cons:

- difficult to debug.

Consequences of this architecture

Pros:

- keep OCaml's GC;
- scalability: the sequential GC isn't a bottleneck;
- generalization from multicores to networks;
- link two ore more libraries with C interface using OCaml internally.

Cons:

- difficult to debug.

Consequences of this architecture

Pros:

- keep OCaml's GC;
- scalability: the sequential GC isn't a bottleneck;
- generalization from multicores to networks;
- link two ore more libraries with C interface using OCaml internally.

Cons:

- difficult to debug.

Consequences of this architecture

Pros:

- keep OCaml's GC;
- scalability: the sequential GC isn't a bottleneck;
- generalization from multicores to networks;
- link two ore more libraries with C interface using OCaml internally.

Cons:

- difficult to debug.

Consequences of this architecture

Pros:

- keep OCaml's GC;
- scalability: the sequential GC isn't a bottleneck;
- generalization from multicores to networks;
- link two ore more libraries with C interface using OCaml internally.

Cons:

- difficult to debug.

History — 1 / 5

- 2011: Fabrice Le Fessant starts the project at OCamlPro:
  - global variables moved into a big `struct`;
  - pointer to the big `struct` added to runtime functions;
  - assembler changes:
    - On x86_64, `%r13` is now reserved as the context pointer;
  - everything works on one runtime
    - except threads and a couple minor `otherlibs`
  - patch file > 23,000 lines
  - ...no more time, project suspended
- Late 2012: I arrive at Inria:
  - study the code;
  - start porting Fabrice's patch to OCaml svn head...
    - lots of conflicts

History — 1 / 5

- 2011: Fabrice Le Fessant starts the project at OCamlPro:
  - global variables moved into a big `struct`;
  - pointer to the big `struct` added to runtime functions;
  - assembler changes:
    - On x86_64, `%r13` is now reserved as the context pointer;
  - everything works on one runtime
    - except threads and a couple minor `otherlibs`
  - patch file > 23,000 lines
  - ...no more time, project suspended
- Late 2012: I arrive at Inria:
  - study the code;
  - start porting Fabrice's patch to OCaml svn head...
    - lots of conflicts

History — 1 / 5

- 2011: Fabrice Le Fessant starts the project at OCamlPro:
  - global variables moved into a big `struct`;
  - pointer to the big `struct` added to runtime functions;
  - assembler changes:
    - On x86_64, `%r13` is now reserved as the context pointer;
  - everything works on one runtime
    - except threads and a couple minor `otherlibs`
  - patch file > 23,000 lines
  - ...no more time, project suspended
- Late 2012: I arrive at Inria:
  - study the code;
  - start porting Fabrice's patch to OCaml svn head...
    - lots of conflicts

amd64: Fabrice had reserved %r13 for the context pointer

From `asmcomp/amd64/proc.ml`. See any difference?

Mainline in 2011:

```
(* Conventions:
     rax - r11: OCaml function arguments
     rax: OCaml and C function results
     xmm0 - xmm9: OCaml function arguments
     xmm0: OCaml and C function results
```

Mainline in late 2012:

```
(* Conventions:
     rax - r13: OCaml function arguments
     rax: OCaml and C function results
     xmm0 - xmm9: OCaml function arguments
     xmm0: OCaml and C function results
```

They changed the register map in the mean time! (PR#5707)

amd64: Fabrice had reserved %r13 for the context pointer

From `asmcomp/amd64/proc.ml`. See any difference?

Mainline in 2011:

```
(* Conventions:
     rax - r11: OCaml function arguments
     rax: OCaml and C function results
     xmm0 - xmm9: OCaml function arguments
     xmm0: OCaml and C function results
```

Mainline in late 2012:

```
(* Conventions:
     rax - r13: OCaml function arguments
     rax: OCaml and C function results
     xmm0 - xmm9: OCaml function arguments
     xmm0: OCaml and C function results
```

They changed the register map in the mean time! (PR#5707)

History — 2 / 5: I ported Fabrice's work

- Lots of nontrivial conflicts in the assembly part. I fixed it very, very carefully because I couldn't even compile, let alone test;
- I fixed conflicts in the C part (many, but mostly trivial);
- I scanned the code start-to-finish, adding the context parameter to the new part;
- Figured out a way to bootstrap: adding a new `CAMLprim` isn't trivial with OCaml's build system.
- I gradually fixed compilation errors.

Incredibly, hello world ran nearly at the first attempt.

After a couple months, my branch was in the state in which Fabrice had left it, ported to the new svn head.

History — 2 / 5: I ported Fabrice's work

- Lots of nontrivial conflicts in the assembly part. I fixed it very, very carefully because I couldn't even compile, let alone test;
- I fixed conflicts in the C part (many, but mostly trivial);
- I scanned the code start-to-finish, adding the context parameter to the new part;
- Figured out a way to bootstrap: adding a new `CAMLprim` isn't trivial with OCaml's build system.
- I gradually fixed compilation errors.

Incredibly, hello world ran nearly at the first attempt.

After a couple months, my branch was in the state in which Fabrice had left it, ported to the new svn head.

History — 2 / 5: I ported Fabrice's work

- Lots of nontrivial conflicts in the assembly part. I fixed it very, very carefully because I couldn't even compile, let alone test;
- I fixed conflicts in the C part (many, but mostly trivial);
- I scanned the code start-to-finish, adding the context parameter to the new part;
- Figured out a way to bootstrap: adding a new `CAMLprim` isn't trivial with OCaml's build system.
- I gradually fixed compilation errors.

Incredibly, hello world ran nearly at the first attempt.

After a couple months, my branch was in the state in which Fabrice had left it, ported to the new svn head.

History — 2 / 5: I ported Fabrice's work

- Lots of nontrivial conflicts in the assembly part. I fixed it very, very carefully because I couldn't even compile, let alone test;
- I fixed conflicts in the C part (many, but mostly trivial);
- I scanned the code start-to-finish, adding the context parameter to the new part;
- Figured out a way to bootstrap: adding a new `CAMLprim` isn't trivial with OCaml's build system.
- I gradually fixed compilation errors.

Incredibly, hello world ran nearly at the first attempt.

After a couple months, my branch was in the state in which Fabrice had left it, ported to the new svn head.

History — 2 / 5: I ported Fabrice's work

- Lots of nontrivial conflicts in the assembly part. I fixed it very, very carefully because I couldn't even compile, let alone test;
- I fixed conflicts in the C part (many, but mostly trivial);
- I scanned the code start-to-finish, adding the context parameter to the new part;
- Figured out a way to bootstrap: adding a new `CAMLprim` isn't trivial with OCaml's build system.
- I gradually fixed compilation errors.

Incredibly, hello world ran nearly at the first attempt.

After a couple months, my branch was in the state in which Fabrice had left it, ported to the new svn head.

History — 3 / 5: I moved OCaml globals to contexts

We need a separate copy of OCaml globals per context, possibly to add dynamically, after context creation (about one month):

- C part quite easy:
  - defined an extensible buffer in C;
  - added an extensible buffer field to `struct caml_global_context`, made it a GC root;
  - general solution, for C contextual variables as well.
- `ocamlopt` changes not very hard either (but I didn't know its internals):
  - now the compiler generates just a word to be filled with a variable offset for the compilation unit (following a suggestion by Fabrice)
  - I made a couple other minor changes in ocamlopt, needed to propagate information
- `asmcomp/amd64/emit.mlp`: I added indirection to global accesses (currently two levels: optimizable to one)

History — 3 / 5: I moved OCaml globals to contexts

We need a separate copy of OCaml globals per context, possibly to add dynamically, after context creation (about one month):

- C part quite easy:
  - defined an extensible buffer in C;
  - added an extensible buffer field to struct `caml_global_context`, made it a GC root;
  - general solution, for C contextual variables as well.
- `ocamlopt` changes not very hard either (but I didn't know its internals):
  - now the compiler generates just a word to be filled with a variable offset for the compilation unit (following a suggestion by Fabrice)
  - I made a couple other minor changes in ocamlopt, needed to propagate information
- `asmcomp/amd64/emit.mlp`: I added indirection to global accesses (currently two levels: optimizable to one)

History — 3 / 5: I moved OCaml globals to contexts

We need a separate copy of OCaml globals per context, possibly to add dynamically, after context creation (about one month):

- C part quite easy:
  - defined an extensible buffer in C;
  - added an extensible buffer field to `struct caml_global_context`, made it a GC root;
  - general solution, for C contextual variables as well.
- `ocamlopt` changes not very hard either (but I didn't know its internals):
  - now the compiler generates just a word to be filled with a variable offset for the compilation unit (following a suggestion by Fabrice)
  - I made a couple other minor changes in ocamlopt, needed to propagate information
- `asmcomp/amd64/emit.mlp`: I added indirection to global accesses (currently two levels: optimizable to one)

How to access a global

An access to the second global (offset 8) of the module Q on amd64 GNU/Linux (PIC):

- before:

```
movq camlQ@GOTPCREL(%rip), %rax # load Q's address
movq 8(%rax), %rax  # dereference the second word
```

- after:

```
movq camlQ@GOTPCREL(%rip), %rax # load Q's offset address
movq (%rax), %rax # load Q's offset
addq 56(%r13), %rax # globals + offset = Q's address
movq 8(%rax), %rax # dereference the second word
```

How to access a global

An access to the second global (offset 8) of the module Q on amd64 GNU/Linux (PIC):

- before:

```
movq camlQ@GOTPCREL(%rip), %rax # load Q's address
movq 8(%rax), %rax  # dereference the second word
```

- after:

```
movq camlQ@GOTPCREL(%rip), %rax # load Q's offset address
movq (%rax), %rax # load Q's offset
addq 56(%r13), %rax # globals + offset = Q's address
movq 8(%rax), %rax # dereference the second word
```

History — 4 / 5: context split

I implemented the context split operator.

Somewhat inspired by my PhD thesis work ("unexec", §3). From C:

- make a Caml tuple containing all globals plus the function to run
- serialize it into a buffer (which preserves sharing)
- in each new thread:
  - deserialize the buffer
  - set the global variables
  - run the function

Generalized marshalling to serialize channels (at least `stdin`, `stdout` and `stderr`!).

Hard to get right. I was also guilty of naïve premature optimization in a couple cases.

History — 4 / 5: context split

I implemented the context split operator.

Somewhat inspired by my PhD thesis work ("unexec", §3). From C:

- make a Caml tuple containing all globals plus the function to run
- serialize it into a buffer (which preserves sharing)
- in each new thread:
  - deserialize the buffer
  - set the global variables
  - run the function

Generalized marshalling to serialize channels (at least `stdin`, `stdout` and `stderr`!).

Hard to get right. I was also guilty of naïve premature optimization in a couple cases.

History — 4 / 5: context split

I implemented the context split operator.

Somewhat inspired by my PhD thesis work ("unexec", §3). From C:

- make a Caml tuple containing all globals plus the function to run
- serialize it into a buffer (which preserves sharing)
- in each new thread:
  - deserialize the buffer
  - set the global variables
  - run the function

Generalized marshalling to serialize channels (at least `stdin`, `stdout` and `stderr`!).

Hard to get right. I was also guilty of naïve premature optimization in a couple cases.

History — 5 / 5

- Implemented communication operators
- Made `otherlibs/systhreads` support the multi-runtime: multi-runtime + (non-parallel)multi-thread
  - still chasing the last bugs
- Fixed context splitting on bytecode

In all such cases:

- relatively little coding time;
- debugging is very difficult.

History — 5 / 5

- Implemented communication operators
- Made `otherlibs/systhreads` support the multi-runtime: multi-runtime + (non-parallel)multi-thread
  - still chasing the last bugs
- Fixed context splitting on bytecode

In all such cases:

- relatively little coding time;
- debugging is very difficult.

Challenges — 1 / 2

Since the GC is moving, heaps must be disjoint:

No Caml pointers from one context to another context heap!

If I violate this condition by mistake, I get problems which are extremely painful to debug.

- pointed data changes for apparently no reason;
- crash when following an invalid pointer (if I'm lucky);
  - very non-deterministic;
  - crashes usually far from their cause, in space and time:
    - patiently debug with prints, gdb, `valgrind` and deductive reasoning.

I sprinkle my code with forced collections, to intentionally cause such crashes by stressing the system.

Challenges — 1 / 2

Since the GC is moving, heaps must be disjoint:

No Caml pointers from one context to another context heap!

If I violate this condition by mistake, I get problems which are extremely painful to debug.

- pointed data changes for apparently no reason;
- crash when following an invalid pointer (if I'm lucky);
  - very non-deterministic;
  - crashes usually far from their cause, in space and time:
    - patiently debug with prints, `gdb`, `valgrind` and deductive reasoning.

I sprinkle my code with forced collections, to intentionally cause such crashes by stressing the system.

Challenges — 1 / 2

Since the GC is moving, heaps must be disjoint:

No Caml pointers from one context to another context heap!

If I violate this condition by mistake, I get problems which are extremely painful to debug.

- pointed data changes for apparently no reason;
- crash when following an invalid pointer (if I'm lucky);
  - very non-deterministic;
  - crashes usually far from their cause, in space and time:
    - patiently debug with prints, `gdb`, `valgrind` and deductive reasoning.

I sprinkle my code with forced collections, to intentionally cause such crashes by stressing the system.

Challenges — 2 / 2

Debugging is difficult:

- I occasionally forgot to protect local C variables of type `value` from the GC, with CAMLparam$X$/CAMLlocal$X$/CAMLreturn$X$;
  - same disaster as above.
- once I spent several weeks chasing a single bug in the assembly code, which trashed the context pointer when returning from Caml code to C code:
  - the cause: a missing conversion from words to bytes in an assembly macro

```
/* Load global [srclabel] in register [dstreg]. */
#define POP_VAR(dstlabel) \
        popq dstlabel*8(%r13)
```

Challenges — 2 / 2

Debugging is difficult:

- I occasionally forgot to protect local C variables of type `value` from the GC, with CAMLparam$X$/CAMLlocal$X$/CAMLreturn$X$;
  - same disaster as above.
- once I spent several weeks chasing a single bug in the assembly code, which trashed the context pointer when returning from Caml code to C code:
  - the cause: a missing conversion from words to bytes in an assembly macro

```
/* Load global [srclabel] in register [dstreg]. */
#define POP_VAR(dstlabel) \
        popq dstlabel*8(%r13)
```

OCaml interface — split

The main primitive from `stdlib/context.mli`:

```
type t  (* Abstract *)

val split_into_context_array :
  int -> (int -> unit)
  -> (t array)
```

Communication model

Fabrice suggested me to look at Erlang and Scala for inspiration...

...I wasn't too impressed:

[To do: somebody reminded me that my solution doesn't treat *failure* in the same elaborate way. That's correct: here I'm just speaking about the general communication model.]

- a "process" also serves as a mailbox;
- as a common idiom they dispatch on a message field, in practice simulating multiple mailboxes.

In my mind repeating code patterns $\equiv$ insufficient abstraction.

I extended the model (a simple idea: probably it occurred to somebody else as well).

Mailboxes:

- are a separate type;
- can be sent as messages, à-la-π-calculus.

Communication model

Fabrice suggested me to look at Erlang and Scala for inspiration...

...I wasn't too impressed:

[To do: somebody reminded me that my solution doesn't treat *failure* in the same elaborate way. That's correct: here I'm just speaking about the general communication model.]

- a "process" also serves as a mailbox;
- as a common idiom they dispatch on a message field, in practice simulating multiple mailboxes.

In my mind repeating code patterns ≡ insufficient abstraction.

I extended the model (a simple idea: probably it occurred to somebody else as well).

Mailboxes:

- are a separate type;
- can be sent as messages, à-la-π-calculus.

Communication model

Fabrice suggested me to look at Erlang and Scala for inspiration...

...I wasn't too impressed:

[To do: somebody reminded me that my solution doesn't treat *failure* in the same elaborate way. That's correct: here I'm just speaking about the general communication model.]

- a "process" also serves as a mailbox;
- as a common idiom they dispatch on a message field, in practice simulating multiple mailboxes.

In my mind repeating code patterns $\equiv$ insufficient abstraction.

I extended the model (a simple idea: probably it occurred to somebody else as well).

Mailboxes:

- are a separate type;
- can be sent as messages, à-la-π-calculus.

Communication model

Fabrice suggested me to look at Erlang and Scala for inspiration...

...I wasn't too impressed:

[To do: somebody reminded me that my solution doesn't treat *failure* in the same elaborate way. That's correct: here I'm just speaking about the general communication model.]

- a "process" also serves as a mailbox;
- as a common idiom they dispatch on a message field, in practice simulating multiple mailboxes.

In my mind repeating code patterns $\equiv$ insufficient abstraction.

I extended the model (a simple idea: probably it occurred to somebody else as well).

Mailboxes:

- are a separate type;
- can be sent as messages, à-la-π-calculus.

Low-level OCaml interface — mailboxes

From `stdlib/context.mli`:

```
type mailbox  (* abstract *)
val make_mailbox : unit -> mailbox

val send : mailbox -> 'a -> unit
val receive : mailbox -> 'a  (* unsafe *)
```

Restricting a mailbox to a single type would be very constraining.

Relatively easy to implement, with marshalling and synchronization; LIFO.

Just **a good simple layer on which to build higher-level interfaces**.

A more comfortable split

Friendlier split functions using mailboxes, implemented in OCaml:

```
val split1 :
  (mailbox -> unit)
  -> (*new context mailbox*)mailbox

val split_into_mailbox_list :
  int -> (int -> mailbox -> unit)
  -> (*mailboxes to new contexts*)(mailbox list)
```

"Conservative" high-level interface: actors

We can have a simple safe layer implemented on top of the low-level interface.

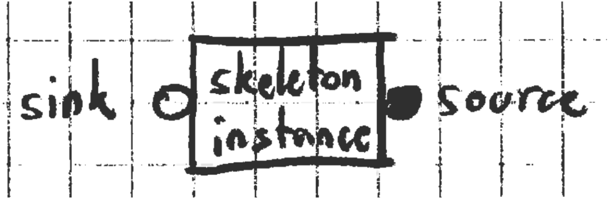
```
type message =
| Int of int
| String of string
| Float of float
| Pair of message * message
| Mailbox of mailbox
| ...

val send : mailbox -> message -> unit
val receive : mailbox -> message
```

Pro: trivial to generalize to communication over sockets.
Not implemented yet, but easy.

Algorithmic skeletons — 1 / 3

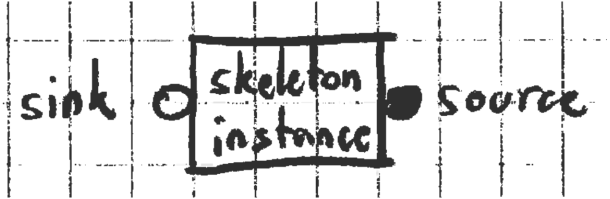
A skeleton instance consumes objects, computes something and produces results:




```
type 'a sink = 'a -> unit
type 'a source = unit -> 'a
type ('a, 'b) instantiated_skeleton =
  ('a sink) * ('b source)
```

Algorithmic skeletons — 1 / 3

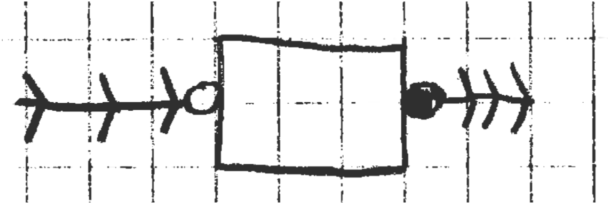
A skeleton instance consumes objects, computes something and produces results:




```
type 'a sink = 'a -> unit
type 'a source = unit -> 'a
type ('a, 'b) instantiated_skeleton =
  ('a sink) * ('b source)
```

Algorithmic skeletons — 2 / 3

The opportunity for parallelization comes from streams:



Results exit in the same *order* as the arguments entered, but (hopefully) at a faster rate.

Algorithmic skeletons — 3 / 3

A skeleton is a (potentially) parallel computation not yet allocated on cores.

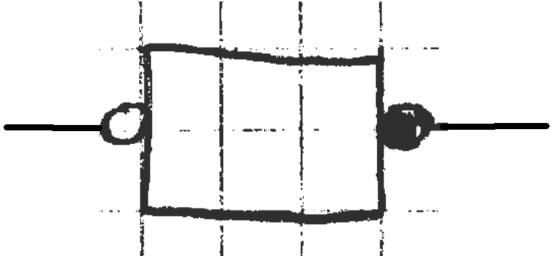
```
type ('input, 'output) skeleton  (* abstract *)
```

Instantiate a skeleton on the available cores:

```
val instantiate : (('a, 'b) skeleton) ->
  (('a, 'b) instantiated_skeleton)
```

Trivial skeleton

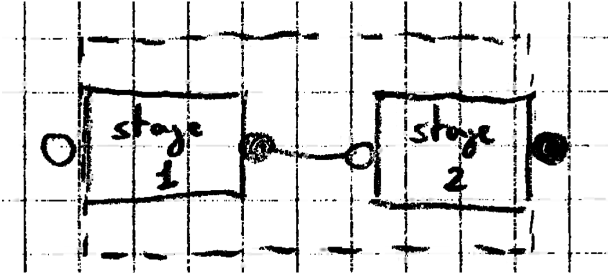
Turn an OCaml function into a (non-parallel) skeleton:



```
val trivial :
  ('a -> 'b)
  -> (('a, 'b) skeleton)
```

Pipeline skeleton

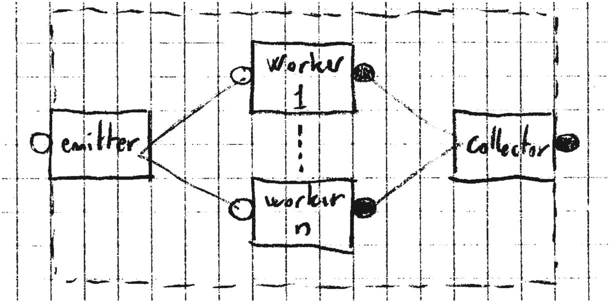
Compose two skeletons into one pipeline skeleton:




```
val pipeline :
  (('a, 'b) skeleton) -> (('b, 'c) skeleton)
  -> (('a, 'c) skeleton)
```

Taskfarm skeleton

Compose n skeletons into one taskfarm skeleton:

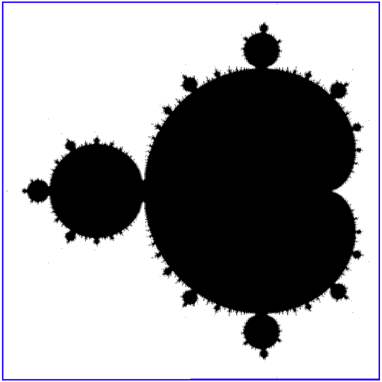



```
val task_farm :
   int -> (('a, 'b) skeleton)
   -> (('a, 'b) skeleton)
```

Tentative high-level interface example — 1 / 2

The *Computer Language Shootout* at `benchmarksgame.alioth.debian.org` contains an OCaml program to generate a Mandelbrot set approximation as a PNM file.

The problem: parallelize it for multicores.



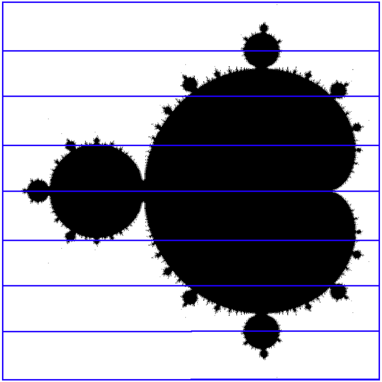
Tentative high-level interface example — 2 / 2

Embarrassingly parallel:

- generate different *horizontal stripes* in parallel

Yet not completely trivial: looks:

- "black" areas are slower to fill: we need auto-balancing
  - taskfarm skeleton



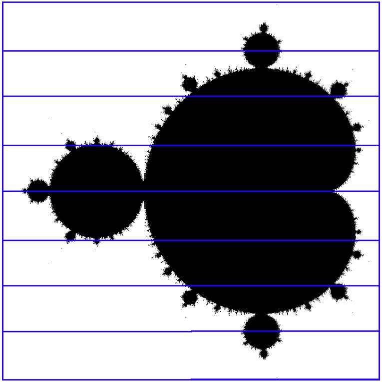
Tentative high-level interface example — 2 / 2

Embarrassingly parallel:

- generate different *horizontal stripes* in parallel

Yet not completely trivial: looks:

- “black” areas are slower to fill: we need auto-balancing
  - taskfarm skeleton



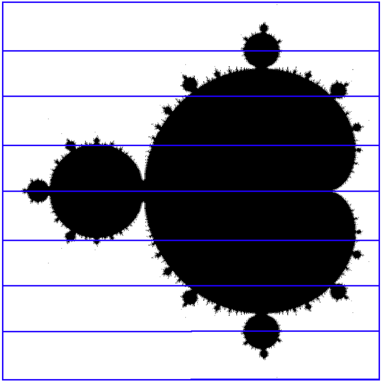
Tentative high-level interface example — 2 / 2

Embarrassingly parallel:

- generate different *horizontal stripes* in parallel

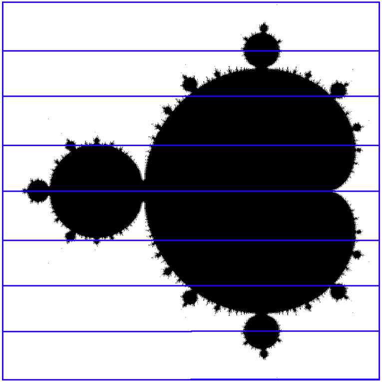
Yet not completely trivial: looks:

- "black" areas are slower to fill: we need auto-balancing
  - taskfarm skeleton



Live demo

[Live demo]



Status

The code is available on `https://github.com/lucasaiu/ocaml`.

- It's polluted with my debug prints everywhere; after solving the last crashes, I'll clean it up.
- It works reliably when not using OCaml threads, with bytecode and with native code on amd64 GNU/Linux.
- Sequential performance is good: no more than $5 \sim 10\%$ overhead.

Future developments

In the short term I will:

- fix the last multi-context + multi-thread bugs;
- restore C API compatibility using C preprocessor macros;
- support C libraries;
- update the configuration system to support (with one context only) all the other architectures;

In the longer term we'd like to:

- add an "ancient" generation for data to share among runtimes (either read-only and immortal, or hand-managed);
- decide on a high-level communication interfaces;
- we'd love to see this integrated into the mainline;
- port the multi-runtime system to the other architectures where performance is important.

Future developments

In the short term I will:

- fix the last multi-context + multi-thread bugs;
- restore C API compatibility using C preprocessor macros;
- support C libraries;
- update the configuration system to support (with one context only) all the other architectures;

In the longer term we'd like to:

- add an "ancient" generation for data to share among runtimes (either read-only and immortal, or hand-managed);
- decide on a high-level communication interfaces;
- we'd love to see this integrated into the mainline;
- port the multi-runtime system to the other architectures where performance is important.

Thanks

Thanks!

https://github.com/lucasaiu/ocaml