Zoo:
A framework for the verification of concurrent OCaml 5 programs using separation logic

Clément
Allain

Gabriel
Scherer

- Introduction
- Zoo overview
- Verification contributions
- Physical equality
- Future work

OCaml 5 (2022)

Parallelism: multi-core runtime, domains, atomic references

Concurrency: algebraic effects

⇓

Nascent ecosystem of parallel & concurrent software

`Domainslib`, `Saturn`, `Eio` . . .

⇓

Formal verification

OCaml verification ecosystem

Language	**Concurrency**	**Iris**	$\approx$ **OCaml**	**Translation**	**Automation**
Cameleer	☹	☹	☺	☺	☺
`coq_of_ocaml`	☹	☹	☺	☺	☹
CFML	☹	☹	☺	☺	☹
Osiris	☹	☺	☺	☺	☹
HeapLang	☺	☺	☹	☹	😐
Zoo	☺	☺	☺	☺	😐

Introduction

Zoo overview

Verification contributions

Physical equality

Future work

Zoo

OCaml

DUNE

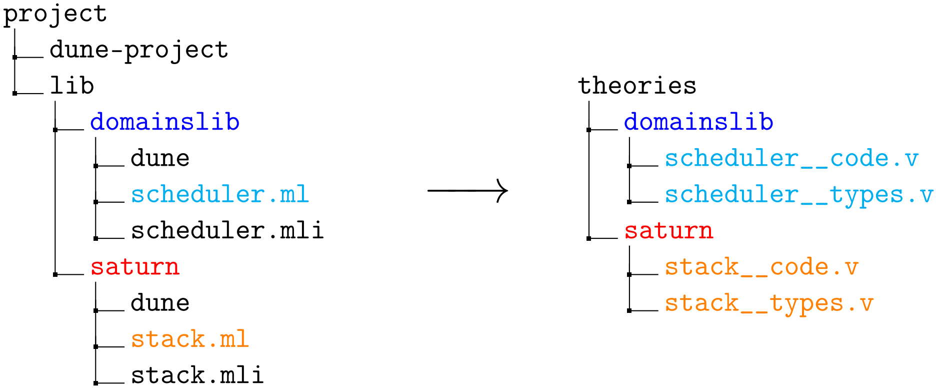
`ocaml2zoo` →

Iris

Zoo

ROCQ

Zoo in practice



- project
  - dune-project
  - lib
    - domainslib
      - dune
      - scheduler.ml
      - scheduler.mli
    - saturn
      - dune
      - stack.ml
      - stack.mli

⟶

- theories
  - domainslib
    - scheduler__code.v
    - scheduler__types.v
  - saturn
    - stack__code.v
    - stack__types.v

```
$ ocaml2zoo project theories
```

Zoo in practice

```
let rec push t v =
  let old = Atomic.get t in
  let new_ = v :: old in
  if not (Atomic.compare_and_set t old new_) then (
    Domain.cpu_relax () ;
    push t v
  )
```

```
Definition stack_push : val :=
  rec: "stack_push" "t" "v" =>
    let: "old" := !"t" in
    let: "new" := ‘Cons( "v", "old" ) in
    if: ~ CAS "t" "old" "new" then (
      domain_cpu_relax () ;;
      "stack_push" "t" "v"
    ).
```

Zoo in practice

```
Lemma stack_push_spec t ι v :
  <<<
    stack_inv t ι
  | ∀∀ vs, stack_model t vs
  >>>
    stack_push t v @ ↑ι
  <<<
    stack_model t (v :: vs)
  | RET (); True
  >>>.
Proof. ... Qed.
```

`stack_push` is *linearizable*

Zoo in practice

```
Lemma stack_push_spec t ι v :
  <<<
    stack_inv t ι
  | ∀∀ vs, stack_model t vs
  >>>
    stack_push t v @ ↑ι
  <<<
    stack_model t (v :: vs)
  | RET (); True
  >>>.
Proof. ... Qed.
```

`stack_push` is *linearizable*

Zoo features

- Algebraic data types
- Records
- Mutually recursive functions
- Physical equality
- Structural equality

- Prophecy variables
- Diaframe (basic automation)

- Atomic references
- Atomic record fields
- Atomic arrays
- Generative constructors

Introduction

Zoo overview

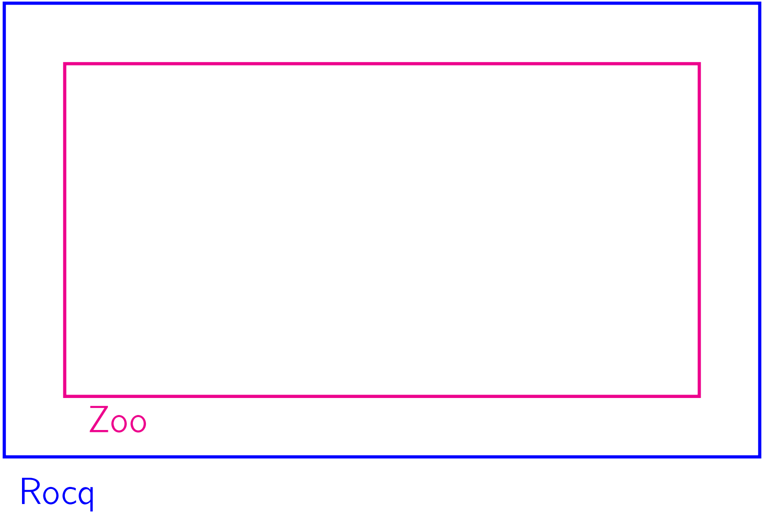
Verification contributions

Physical equality

Future work

Verification contributions

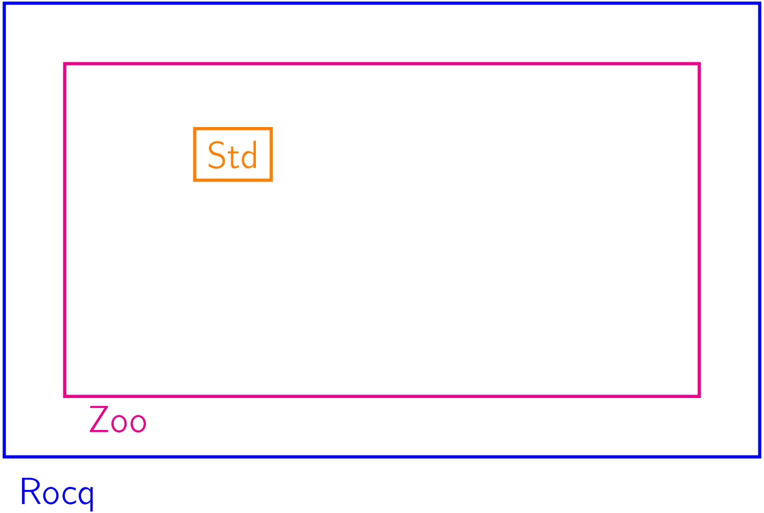
Verification contributions



Zoo

Rocq

Verification contributions



Std

Zoo

Rocq

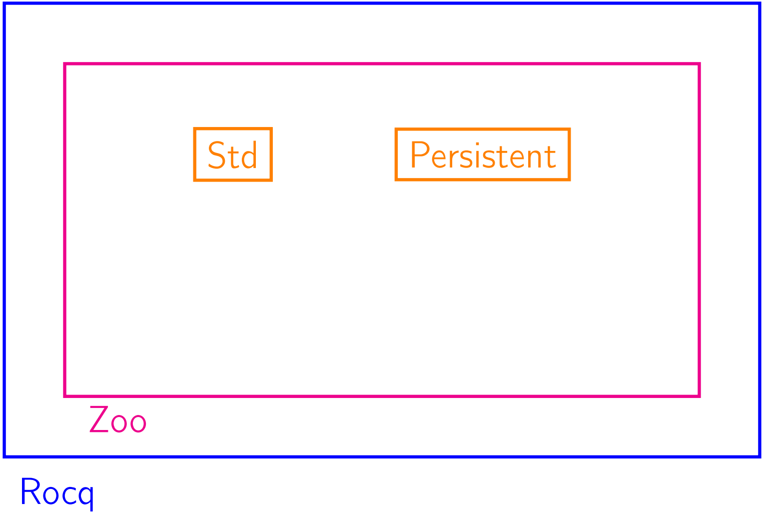
Verification contributions

Standard data structures

- `Array`
- `Dynarray`
- `List`
- `Stack`
- `Queue`
- `Inf_array`
- `Deque`
- `Domain`
- `Mutex`
- `Semaphore`
- `Condition`
- `Ivar`
- `Mvar`

Rocq

Verification contributions



Std

Persistent

Zoo

Rocq

Verification contrib

Basile Clément

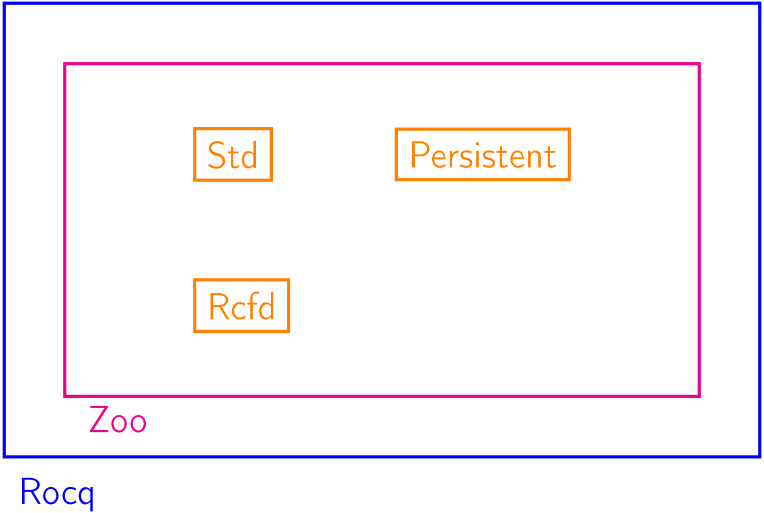
Gabriel Scherer

Persistent data structures

- Persistent array
- Persistent store
- Persistent union-find

Roc

Verification contributions



Std

Persistent

Rcfd

Zoo

Rocq

Verification contrib

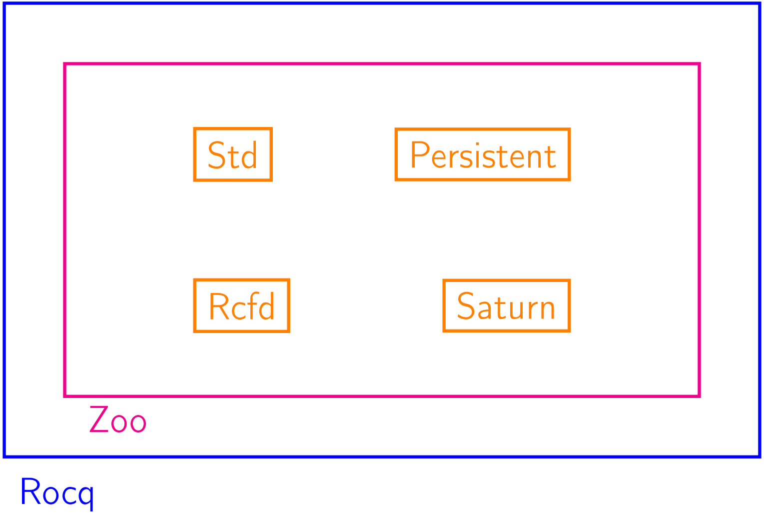
Thomas Leonard

Parallelism-safe file descriptor

- Generative constructors
- Intricate concurrent protocol
- Two ownership regimes

Roc

Verification contributions


Std

Persistent

Rcfd

Saturn

Zoo

Rocq

Verification contrib

Vesa Karvonen

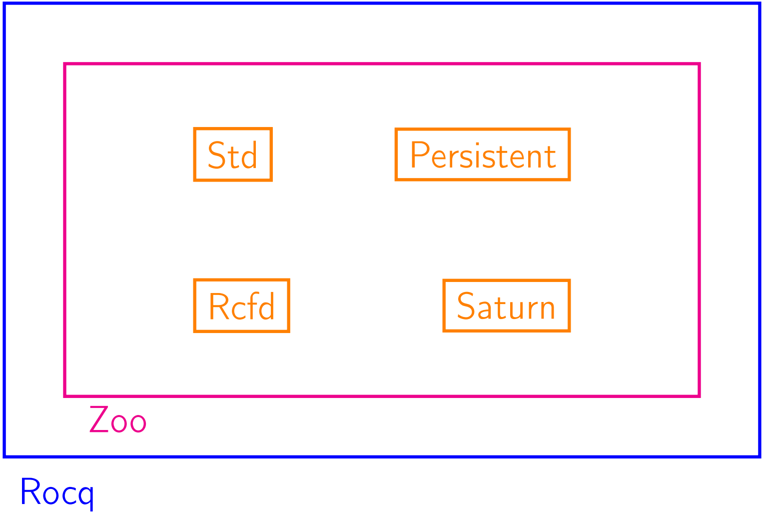
Carine Morel

Standard lock-free data structures

- Stacks
- List-based queues
- Array-based queues
- Stack-based queues

Roco

Verification contributions



Std

Persistent

Rcfd

Saturn

Zoo

Rocq

Introduction

Zoo overview

Verification contributions

Physical equality

Future work

Physical equality in *fine-grained* concurrent programs

```
type 'a t =
  'a list Atomic.t

let create () =
  Atomic.make []

let rec push t v =
  let old = Atomic.get t in
  let new = v :: old in
  if Atomic.compare_and_set t old new then
    ()
  else
    push t v
```

Physical equality in *fine-grained* concurrent programs

```
type 'a t =
  'a list Atomic.t

let create () =
  Atomic.make []

let rec push t v =
  let old = Atomic.get t in
  let new = v :: old in
  if Atomic.compare_and_set t old new then
    ()
  else
    push t v
```

Physical comparison (==)

Physical equality in *fine-grained* concurrent programs

```
type 'a t =
  'a list Atomic.t

let create () =
  Atomic.make []

let rec push t v =
  let old = Atomic.get t in
  let new = v :: old in
  if Atomic.compare_and_set t old new then
    ()
  else
    push t v
```

Physical comparison (==)

OCaml: under-specified

Physical equality in *fine-grained* concurrent programs

```
type 'a t =
  'a list Atomic.t

let create () =
  Atomic.make []

let rec push t v =
  let old = Atomic.get t in
  let new = v :: old in
  if Atomic.compare_and_set t old new then
    ()
  else
    push t v
```

Physical comparison (==)

OCaml: under-specified

HeapLang: too restrictive
incompatible w/ OCaml

When physical equality returns `true`

```
let rec push t v =
  let old = Atomic.get t in
  let new = v :: old in

  if Atomic.compare_and_set t old new then


    ()
  else
    push t v
```

When physical equality returns `true`

```
let rec push t v =
  let old = Atomic.get t in
  let new = v :: old in
  ⟨ vs. stack-model t vs ∗ . . . ⟩
  if Atomic.compare_and_set t old new then


    ()
  else
    push t v
```

When physical equality returns `true`

```
let rec push t v =
  let old = Atomic.get t in
  let new = v :: old in
  $\langle\ vs.\ \text{stack-model}\ t\ vs * \ldots\ \rangle$
  if Atomic.compare_and_set t old new then
    $\langle\ \text{stack-model}\ t\ (v :: old) * vs \overset{\text{phys}}{\approx} old\ \rangle$

    ()
  else
    push t v
```

When physical equality returns `true`

```
let rec push t v =
  let old = Atomic.get t in
  let new = v :: old in
  $\langle\ vs.\ \text{stack-model}\ t\ vs * \ldots\ \rangle$
  if Atomic.compare_and_set t old new then
    $\langle\ \text{stack-model}\ t\ (v :: old) * vs \overset{\text{phys}}{\approx} old\ \rangle$
    $\langle\ \text{stack-model}\ t\ (v :: vs)\ \rangle$
    ()
  else
    push t v
```

When physical equality returns `true`

```
let rec push t v =
  l
  l
  ⟨
  i


  e
    push t v
```

Sharing of immutable blocks

```
let test1 = 1 :: [] == 1 :: []   (* maybe true *)
let test2 = 1 :: [] == 1 :: []   (* maybe false *)
```

$$v_1 \overset{\text{rocq}}{=} v_2 \quad \not\Longrightarrow \quad v_1 \overset{\text{phys}}{\approx} v_2$$

When physical equality returns true

```
let
  l
  l
  ⟨
  i

  e
```

Value representation conflicts

```
type any = Any : 'a -> any
let test1 = Any false == Any 0   (* maybe true *)
let test2 = Any None  == Any 0   (* maybe true *)
let test3 = Any []    == Any 0   (* maybe true *)
```

$$v_1 \overset{\text{phys}}{\approx} v_2 \quad \not\Longrightarrow \quad v_1 \overset{\text{rocq}}{=} v_2$$

When physical equality returns `true`

```
let rec push t v =
  let old = Atomic.get t in
  let new = v :: old in
  $\langle\ vs.\ \textsf{stack-model}\ t\ vs * \ldots\ \rangle$
  if Atomic.compare_and_set t old new then
    $\langle\ \textsf{stack-model}\ t\ (v :: old) * vs \overset{\textsf{phys}}{\approx} old\ \rangle$
    ~~$\langle\ \textsf{stack-model}\ t\ (v :: vs)\ \rangle$~~
    ()
  else
    push t v
```

When physical equality returns `false`

```
type state = Open of Unix.file_descr | Closing of (unit -> unit)
type t = { mutable state: state [@atomic]; ... }

let make fd = { state= Open fd; ... }
let close t = match t.state with
  | Closing _ -> false
  | Open fd as old ->
      let close () = Unix.close fd in
      let new = Closing close in

      if Atomic.Loc.compare_and_set [%atomic.loc t.state] old new
      then ... else



        false
```

When physical equality returns `false`

```
type state = Open of Unix.file_descr | Closing of (unit -> unit)
type t = { mutable state: state [@atomic]; ... }

let make fd = { state= Open fd; ... }

let close t = match t.state with
  | Closing _ -> false
  | Open fd as old ->
      let close () = Unix.close fd in
      let new = Closing close in

      if Atomic.Loc.compare_and_set [%atomic.loc t.state] old new
      then ... else


        false
```

When physical equality returns `false`

```
type state = Open of Unix.file_descr | Closing of (unit -> unit)
type t = { mutable state: state [@atomic]; ... }

let make fd = { state= Open fd; ... }

let close t = match t.state with
  | Closing _ -> false
  | Open fd as old ->
      let close () = Unix.close fd in
      let new = Closing close in

      if Atomic.Loc.compare_and_set [%atomic.loc t.state] old new
      then ... else


        false
```

When physical equality returns `false`

```
type state = Open of Unix.file_descr | Closing of (unit -> unit)
type t = { mutable state: state [@atomic]; ... }

let make fd = { state= Open fd; ... }

let close t = match t.state with
  | Closing _ -> false
  | Open fd as old ->
      let close () = Unix.close fd in
      let new = Closing close in

      if Atomic.Loc.compare_and_set [%atomic.loc t.state] old new
      then ... else



        false
```

When physical equality returns false

```
type state = Open of Unix.file_descr | Closing of (unit -> unit)
type t = { mutable state: state [@atomic]; ... }

let make fd = { state= Open fd; ... }

let close t = match t.state with
  | Closing _ -> false
  | Open fd as old ->
      let close () = Unix.close fd in
      let new = Closing close in
      ⟨ state. t.state ↦ state ∗ ... ⟩
      if Atomic.Loc.compare_and_set [%atomic.loc t.state] old new
      then ... else



        false
```

When physical equality returns `false`

```
type state = Open of Unix.file_descr | Closing of (unit -> unit)
type t = { mutable state: state [@atomic]; ... }

let make fd = { state= Open fd; ... }

let close t = match t.state with
  | Closing _ -> false
  | Open fd as old ->
      let close () = Unix.close fd in
      let new = Closing close in
```

$$\langle\ state.\ t.state \mapsto state * \ldots\ \rangle$$

```
      if Atomic.Loc.compare_and_set [%atomic.loc t.state] old new
      then ... else
```

$$\langle\ t.state \mapsto state * state \overset{\text{phys}}{\not\approx} old * \ldots\ \rangle$$

```
        false
```

When physical equality returns `false`

```
type state = Open of Unix.file_descr | Closing of (unit -> unit)
type t = { mutable state: state [@atomic]; ... }

let make fd = { state= Open fd; ... }

let close t = match t.state with
  | Closing _ -> false
  | Open fd as old ->
      let close () = Unix.close fd in
      let new = Closing close in
```

$$\langle\ \mathit{state}.\ t.\mathsf{state} \mapsto \mathit{state} * \ldots\ \rangle$$

```
      if Atomic.Loc.compare_and_set [%atomic.loc t.state] old new
      then ... else
```

$$\langle\ t.\mathsf{state} \mapsto \mathit{state} * \mathit{state} \overset{\mathsf{phys}}{\not\approx} \mathit{old} * \ldots\ \rangle$$

$$\langle\ t.\mathsf{state} \mapsto \mathsf{Closing}\ - * \ldots\ \rangle$$

```
        false
```

When physical

```
type state =                                              t)
type t = { m

let make fd

let close t
  | Closing
  | Open fd
      let cl
      let ne
      ⟨ state
      if Ato                                              new
      then .
        ⟨ t.s
        ⟨ t.s
        false
```

Unsharing of immutable blocks

```ocaml
let x = Some 0
let test = x == x   (* maybe false *)
```

$$v_1^{id_1} \overset{\text{phys}}{\not\approx} v_2^{id_2} \quad \not\Longrightarrow \quad id_1 \overset{\text{rocq}}{\neq} id_2$$

Clément Allain
Impossible! Unique identity.

Armaël Guéneau
This would be *unsharing*.

Vincent Laviron
It's possible!

When physical equality returns `false`

```
type state = Open of Unix.file_descr | Closing of (unit -> unit)
type t = { mutable state: state [@atomic]; ... }

let make fd = { state= Open fd; ... }

let close t = match t.state with
  | Closing _ -> false
  | Open fd as old ->
      let close () = Unix.close fd in
      let new = Closing close in
```

$\langle\, \mathit{state}.\ t.\mathsf{state} \mapsto \mathit{state} * \ldots \,\rangle$

```
      if Atomic.Loc.compare_and_set [%atomic.loc t.state] old new
      then ... else
```

$\langle\, t.\mathsf{state} \mapsto \mathit{state} * \mathit{state} \overset{\text{phys}}{\not\approx} \mathit{old} * \ldots \,\rangle$

~~$\langle\, t.\mathsf{state} \mapsto \mathsf{Closing}\ \_ * \ldots \,\rangle$~~

```
        false
```

Generative constructors

```
type 'a glist =
  | Nil
  | Cons of 'a * 'a glist [@generative]

type state =
  | Open of Unix.file_descr [@generative] [@zoo.reveal]
  | Closing of (unit -> unit)
```

Introduction

Zoo overview

Verification contributions

Physical equality

Future work

Future work

- **Language features**
    - Exceptions
    - Algebraic effects
    - Modules & functors
- **Coupling with semi-automated verification**
- **Relaxed memory**

Thank you for your attention!