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

Clément Allain

Gabriel Scherer

# Context

Verification of *fine-grained concurrent* OCaml 5 programs

`Saturn`

`Kcas`

`Parabs`

Iris

ROCQ

# Iris artillery

- higher-order ghost state
- user-defined ghost state
- invariants
- atomic updates
- prophecy variables

# HeapLang, the canonical Iris language

- **simple & expressive**, *but*
- **lacks basic abstractions**
  - tuples, records
  - algebraic data types (ADTs)
  - mutually recursive functions
- **lacks a standard library**
- **physical equality is problematic**
  - restricted to "unboxed" values
  - incompatible with OCaml

# The big picture

OCaml

DUNE

$\xrightarrow{\texttt{ocaml2zoo}}$

Iris

Zoo

ROCQ

# Zoo in practice

```
project
├── dune-project
└── lib
    ├── domainslib
    │   ├── dune
    │   ├── scheduler.ml
    │   └── scheduler.mli
    └── saturn
        ├── dune
        ├── queue.ml
        └── queue.mli
```

$\Longrightarrow$

```
theories
├── domainslib
│   ├── scheduler__code.v
│   └── scheduler__types.v
└── saturn
    ├── queue__code.v
    └── queue__types.v
```

```
$ ocaml2zoo project theories
```

# Zoo in practice

```
project
├── dune-project
└── lib
    └── domainslib
        ├── dune
        ├── scheduler.ml
        └── scheduler.mli
```

$\Longrightarrow$

```
theories
└── domainslib
    ├── scheduler__code.v
    └── scheduler__types.v
```

```
$ ocaml2zoo project theories
```

# Zoo in practice

```
Lemma stack_push_spec_seq t ι v :
  {{{
    stack_model t vs
  }}}
    stack_push t v
  {{{
    RET ();
    stack_model t (v :: vs)
  }}}.
Proof.
  ...
Qed.
```

```
Lemma stack_push_spec_atomic 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.
```

# Algebraic data types

```
type 'a t =
  | Nil
  | Cons of 'a * 'a t

let rec map fn t =
  match t with
  | Nil -> Nil
  | Cons (x, t) ->
      let y = fn x in
      Cons (y, map fn t)
```

```
Notation "'Nil'"  := (
  in_type "t" 0
)(in custom zoo_tag).
Notation "'Cons'" := (
  in_type "t" 1
)(in custom zoo_tag).

Definition map : val :=
  rec: "map" "fn" "t" =>
    match: "t" with
    | Nil => §Nil
    | Cons "x" "t" =>
        let: "y" := "fn" "x" in
        ‘Cons( "y", "map" "fn" "t" )
    end.
```

# Records

```
type 'a t =
  { mutable f1: 'a;
    mutable f2: 'a;
  }

let swap t =
  let f1 = t.f1 in
  t.f1 <- t.f2 ;
  t.f2 <- f1
```

```
Notation "'f1'" := (
  in_type "t" 0
)(in custom zoo_field).
Notation "'f2'" := (
  in_type "t" 1
)(in custom zoo_field).

Definition swap : val :=
  fun: "t" =>
    let: "f1" := "t".{f1} in
    "t" <-{f1} "t".{f2} ;;
    "t" <-{f2} "f1".
```

# Inline records

```
type 'a node =
  | Null
  | Node of
    { mutable next: 'a node;
      mutable data: 'a;
    }
```

```
Notation "'Null'" := (
  in_type "node" 0
)(in custom zoo_tag).
Notation "'Node'" := (
  in_type "node" 1
)(in custom zoo_tag).

Notation "'next'" := (
  in_type "node.Node" 0
)(in custom zoo_field).
Notation "'data'" := (
  in_type "node.Node" 1
)(in custom zoo_field).
```

# Mutually recursive functions

```
let rec f x = g x
and g x = f x
```

```
Definition f_g := (
  recs: "f" "x" => "g" "x"
  and:  "g" "x" => "f" "x"
)%zoo_recs.

(* boilerplate *)

Definition f := ValRecs 0 f_g.
Definition g := ValRecs 1 f_g.

Instance : AsValRecs' f 0 f_g [f;g].
Proof. done. Qed.
Instance : AsValRecs' g 1 f_g [f;g].
Proof. done. Qed.
```

# Concurrency

| OCaml | Model |
|---|---|
| `Atomic.set` $e_1$ $e_2$ | $e_1$ `<-` $e_2$ |
| `Atomic.exchange` $e_1$ $e_2$ | `Xchg` $e_1$`.[contents]` $e_2$ |
| `Atomic.compare_and_set` $e_1$ $e_2$ $e_3$ | `CAS` $e_1$`.[contents]` $e_2$ $e_3$ |
| `Atomic.fetch_and_add` $e_1$ $e_2$ | `FAA` $e_1$`.[contents]` $e_2$ |

`type t = { ...; mutable` $f$`:` $\tau$ `[@atomic]; ... }`

| | |
|---|---|
| `Atomic.Loc.exchange [%atomic.loc` $e_1$`.`$f$`]` $e_2$ | `Xchg` $e_1$`.[`$f$`]` $e_2$ |
| `Atomic.Loc.compare_and_set [%atomic.loc` $e_1$`.`$f$`]` $e_2$ $e_3$ | `CAS` $e_1$`.[`$f$`]` $e_2$ $e_3$ |
| `Atomic.Loc.fetch_and_add [%atomic.loc` $e_1$`.`$f$`]` $e_2$ | `FAA` $e_1$`.[`$f$`]` $e_2$ |

## Standard library

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

# Diaframe (Ike Mulder *et al.*) support

```
Proof.
  ...

  iInv "Hinv" as "(:inv_inner =1)".
  ...
  iSplitR ... { iFrame. iSteps. }

  iInv "Hinv" as "(:inv_inner =2)".
  ...
  iSplitR ... { iFrame. iSteps. }

  ...
Qed.
```

# Restriction on physical comparison

```
Definition val_is_unboxed v :=
  match v with
  | LitV lit =>
      lit_is_unboxed lit
  | InjLV (LitV lit) =>
      lit_is_unboxed lit
  | InjRV (LitV lit) =>
      lit_is_unboxed lit
  | _ =>
      False
  end.

Definition vals_compare_safe v1 v2 :=
  val_is_unboxed v1 ∨ val_is_unboxed v2.
```

# Treiber stack

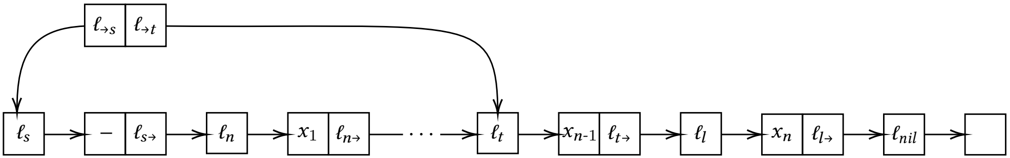
```
type 'a t =
  'a list Atomic.t

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
    push t v
```

---

```
Definition push : val :=
  rec: "push" "t" "v" :=
    let: "old" := ! "t" in
    let: "new" := SOME (ref ("v", "old")) in
    if: CAS "t" "old" "new" then #() else
      "push" "t" "v".
```

# Michael-Scott queue

```
type ('a, _) node =
  | Null : ('a, [> `Null]) node
  | Node : { mutable next: ('a, [`Null | `Node]) node [@atomic];
             mutable data: 'a; }
           -> ('a, [> `Node]) node
type 'a t =
  { mutable front: ('a, [`Node]) node [@atomic];
    mutable back: ('a, [`Node]) node [@atomic]; }
```



# RDCSS

```
NewRDCSS(n) ≜ ref(inl(n));

rec Get(ℓ_n) ≜
  match !ℓ_n with
    inl(n)         ⇒ n
  | inr(ℓ_descr)   ⇒ Complete(ℓ_descr, ℓ_n); Get(ℓ_n)
  end;

Complete(ℓ_descr, ℓ_n) ≜
let (ℓ_m, m_1, n_1, n_2, p) = !ℓ_descr;
let tid = NewGhostId;
let m = !ℓ_m;
let n_new = if m = m_1 then n_2 else n_1;
Resolve(CmpX(ℓ_n, inr(ℓ_descr), inl(n_new)), p, tid);
()
```

```
RDCSS(ℓ_m, ℓ_n, m_1, n_1, n_2) ≜
let p = NewProph;
let ℓ_descr = ref(ℓ_m, m_1, n_1, n_2, p);
rec rdcss_inner() =
  let (v, b) = CmpX(ℓ_n, inl(n_1), inr(ℓ_descr));
  match v with
    inl(n)      ⇒
      if b then
        Complete(ℓ_descr, ℓ_n); n_1
      else  n
  | inr(ℓ'_descr) ⇒
      Complete(ℓ'_descr, ℓ_n); rdcss_inner()
  end;
rdcss_inner()
```

# Physical comparison decides Rocq equality

```
Definition bin_op_eval op v1 v2 :=
  if decide (op = EqOp) then
    if decide (vals_compare_safe v1 v2) then
      Some $ LitV $ LitBool $ bool_decide (v1 = v2)
    else
      None
  else
    ...
```

# Classification of Zoo values

- boolean
- integer
- mutable block (pointer)
- immutable block (tag and fields)
- function

## Non-deterministic semantics

```
let x1 = Some ()
let x2 = Some ()
let test1 = x1 == x1 (* true *)
let test2 = x1 == x2 (* false *)
```

What *guarantees* when physical equality (1) returns `true`,
(2) returns `false`?

# Sharing

```
let test1 = Some 0 == Some 0 (* true *)
let test2 = [0;1]  == [0;1]  (* true *)
```

# Value representation conflicts

```
let test1 = Obj.repr false == Obj.repr 0 (* true *)
let test2 = Obj.repr None  == Obj.repr 0 (* true *)
let test3 = Obj.repr []    == Obj.repr 0 (* true *)
```

# Sharing + conflicts

```
type any =
  Any : 'a -> any

let test1 = Any false == Any 0 (* true *)
let test2 = Any None  == Any 0 (* true *)
let test3 = Any []    == Any 0 (* true *)
```

# Treiber stack

```
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 not @@ Atomic.compare_and_set t old new_ then (
    Domain.cpu_relax () ;
    push t v
  )
```

# Treiber stack specification

```
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.
```

# Unsharing

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

**Clément Allain**
Impossible! Unique identity.

**Armaël Guéneau**
This would be *unsharing*.

**Vincent Laviron**
It's possible!

# Eio.Rcfd

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

let make fd = { ops= 0; state= Open fd }

let closed = Closing (fun () -> ())
let close t =
  match t.state with
  | Closing _ -> false
  | Open fd as prev ->
      let close () = Unix.close fd in
      let next = Closing close in
      if Atomic.Loc.compare_and_set [%atomic.loc t.state] prev next then
        ...
      else
        false
```

# Generative constructors

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

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

Context

Zoo in practice

Zoo features

Physical equality in HeapLang

Physical equality in OCaml

Future work

# Future work

- exceptions
- algebraic effects
- modules & functors
- weak memory
- coupling with semi-automated verification

Thank you for your attention!