Zoo : Un cadriciel pour la vérification de programmes OCaml 5 concurrents en logique de séparation

Clément Allain

29 janvier 2025

Motivation

Vérification de programmes OCaml 5 *concurrents*.

Saturn

`Kcas`

V

Logique de séparation Iris

- État logique personnalisable
  - Protocoles concurrents
  - Atomicité logique
  - Points de linéarisation externes
  - Points de linéarisation dépendants du futur
- Mécanisation en Rocq
- Modèle mémoire faible

Motivation

Vérification de programmes OCaml 5 *concurrents*.

Saturn

`Kcas`

Iris

ROCQ

À la recherche d'un langage de vérification

langage	**concurrence**	**Iris**	**$\simeq$ OCaml**	**traduction**	**automatisation**
Cameleer	☹	☹	☺	☺	☺
`coq_of_ocaml`	☹	☹	☺	☺	☹
CFML	☹	☹	☺	☺	☹
Osiris	☹	☺	☺	☺	☹
HeapLang	☺	☺	☹	☹	😐
Zoo	☺	☺	☺	☺	😐

Zoo, un langage pragmatique

- Fragment formalisé d'OCaml 5
  suffisamment expressif pour Saturn et `Kcas`.
- Sémantique formelle correcte vis-à-vis d'OCaml.

- Instance Iris.

- Commodités :
  - outil de traduction d'OCaml vers Zoo
  - code reconnaissable
  - automatisation minimale (Diaframe)

Zoo en pratique

OCaml

DUNE

$\xrightarrow{\texttt{ocaml2zoo}}$

Iris

Zoo

ROCQ

Zoo en pratique

```
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 en pratique

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

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

queue.ml

```
let pop t = ...
```

```
$ ocaml2zoo project theories
```

Zoo en pratique

- 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 en pratique

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

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

queue__code.v

```
Definition queue_pop : val := ...
```

```
$ ocaml2zoo project theories
```

Zoo en pratique

```
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 en pratique

```
project
├── dune-project
└── lib
    └── domainslib
        ...
        └── queue.mli
```

```
theories
└── domainslib
```

scheduler.ml

```
let get_task =
  ... Saturn.Queue.pop ...
```

```
$ ocaml2zoo project theories
```

Zoo en pratique

```
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
```

project
dune-project

scheduler__code.v

```
From saturn Require Import
  queue__code
  queue__types.

Definition scheduler_get_task : val :=
  ... queue_pop ...
```

$ ocaml2zoo project theories

Zoo en pratique

```
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 en pratique

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

$\Longrightarrow$

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

```
$ ocaml2zoo project theories
```

scheduler__code.v

```
From saturn Require Import
  queue__code
  queue__types.

Definition scheduler_get_task : val :=
  ... queue_pop ...
```

Zoo en pratique

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

$\Longrightarrow$

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

```
$ ocaml2zoo project theories
```

Zoo en pratique

```
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.
```

Types algébriques de données

```
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.
```

Enregistrements

```
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".
```

Enregistrements en place

```
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).
```

Fonctions mutuellement récursives

```
let 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.
```

Concurrence

`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
`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

Bibliothèque standard

- `Array`
- `Dynarray`
- `List`
- `Stack`
- `Queue`
- `Deque`

- `Domain`
- `Atomic_array`
- `Mutex`
- `Condition`

Égalité physique : pile de Treiber

```
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
  )
```

Conflits de représentation des valeurs

```
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 *)
```

Partage

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

Conflits + partage

```
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 *)
```

Pile de Treiber

```
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
  )
```

Égalité physique : Eio.Rcfd

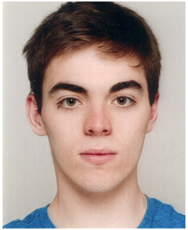
```
type state = Open of Unix.file_descr | Closing of (unit -> unit)
type t = { mutable ops: int [@atomic]; mutable state: state [@atomic]; }

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
```

Départage

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



Clément Allain
Impossible ! Identité unique.

Armaël Guéneau
Ce serait du départage.

Vincent Laviron
C'est possible !

Eio.Rcfd

```
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
```

Constructeurs génératifs

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

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

Merci de votre attention !