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

Clément Allain
Gabriel Scherer

March 21, 2025

Context

Verification of *fine-grained concurrent* OCaml 5 programs

Saturn

Kcas

Iris

ROCQ

In search of a verification language

language	**concurrency**	**Iris**	**$\simeq$ OCaml**	**translation**	**automation**
Cameleer	☹	☹	☺	☺	☺
`coq_of_ocaml`	☹	☹	☺	☺	☹
CFML	☹	☹	☺	☺	☹
Osiris	☹	☺	☺	☺	☹
HeapLang	☺	☺	☹	☹	😐
Zoo	☺	☺	☺	☺	😐

Introduction

Zoo in practice

Zoo features

Physical equality

Structural equality

Specimen: `Kcas` (ongoing work)

Future work

Zoo in practice

OCaml

DUNE

`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

`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`:` τ `[@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

https://github.com/ocaml/ocaml/pull/13404
https://github.com/ocaml/ocaml/pull/13707

Standard library

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

- Domain
- Atomic_array
- Mutex
- Condition

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`?

OCaml's informal specification

`e1 == e2` tests for physical equality of `e1` and `e2`.

On mutable types such as references, arrays, byte sequences, records with mutable fields and objects with mutable instance variables, `e1 == e2` is true if and only if physical modification of `e1` also affects `e2`.

On non-mutable types, the behavior of `(==)` is implementation-dependent; however, it is guaranteed that `e1 == e2` implies `compare e1 e2 = 0`.

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.
```

OCaml's informal specification is too imprecise

```
type 'a t =
  'a ref 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 (
    Domain.cpu_relax () ;
    push t v
  )
```

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 *)
```

Back to Treiber stack

```
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
  )
```

Jourdan's physical equality

purposes: first, it provides a fast mechanism for comparing values using physical equality or hash equality. Second, it is easy to use hash-consing to build fast map structures using hash-consed values as keys. Finally, using such maps it is possible to implement memoization.

This assessment led us, in collaboration with Braibant and Monniaux [BJM13, BJM14], to the study of several methods to implement maximal sharing (i.e., *hash-consing*) and memoization in formally verified Coq programs. We used the case study of binary decision diagrams (BDDs), which are one of the well known uses of the hash-consing technique. We tried different approaches and compared them, as reported in the following sections. These ideas are not currently implemented in Verasco, but we believe some of them (especially the SMART and SMART+UID approaches described in Section 9.4) could be adapted to many of its data structures.

9.1. Safe Physical Equality in Coq: the PHYSEQ Approach

The obvious way of introducing physical equality in Coq is to declare it as an axiom in the development, state that physical equality implies Leibniz equality, and ask the extraction mechanism to extract it to OCaml's physical equality:

```
Parameter physEq: ∀ A:Type, A -> A -> bool.
Axiom physEq_correct: ∀ (A:Type) (x y:A), physEq x y = true -> x = y.
Extract Constant physEq => "(==)".
```

However, this appears to be unsound. Let `a` and `b` be two physically different copies of the same value. Then we have `physEq a a = true` and `a = b`, using Coq's Leibniz equality. Thus, we deduce, in Coq's logic, that `physEq a b = true`, which is wrong.

This unsoundness is of a particular kind: in fact, the axioms we postulate are not inconsistent: they can be easily instantiated by posing `physEq x y = false`. However, the OCaml term `(==)` is not a valid extraction for `physEq`, and using it would make it possible to prove properties on programs that will become false after extraction.

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
```

Unsharing

```
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!

Back to 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
```

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)
```

Specification

```
Axiom structeq_spec : ∀ `{zoo_G : !ZooG Σ} {v1 v2} footprint,
  val_traversable footprint v1 →
  val_traversable footprint v2 →
  {{{
    structeq_footprint footprint
  }}}
    v1 = v2
  {{{ b,
    RET #b;
    structeq_footprint footprint ∗
    ⌜(if b then val_structeq else val_structneq) footprint v1 v2⌝
  }}}.
```

Specification for *abstract* values

```
Lemma structeq_spec_abstract `{zoo_G : !ZooG Σ} v1 v2 :
  val_abstract v1 →
  val_abstract v2 →
  {{{
    True
  }}}
    v1 = v2
  {{{ b,
    RET #b;
    ⌜(if b then (≈) else (≉)) v1 v2⌝
  }}}
Proof.
  ...
Qed.
```

Kcas: software transactional memory for OCaml

```
let a = Loc.make 10 in
let b = Loc.make 52 in
let x = Loc.make 0 in

let tx ~xt =
  let a = Xt.get ~xt a in
  let b = Xt.get ~xt b in
  Xt.set ~xt x (b - a)
in

Xt.commit { tx }
```

Kcas: software transactional memory for OCaml

```
let a = Loc.make 10 in
let b
let x

let tx
  let
  let
  Xt.s
in

Xt.commit { tx }
```

Vesa Karvonen

The main author of Kcas

Kcas: software transactional memory for OCaml

```
let a = Loc.make 10 in
let b = Loc.make 52 in
let x = Loc.make 0 in

let tx ~xt =
  let a = Xt.get ~xt a in
  let b = Xt.get ~xt b in
  Xt.set ~xt x (b - a)
in

Xt.commit { tx }
```

Kcas: software transactional memory for OCaml

```
type ('k, 'v) cache =
  { space: int Loc.t;
    table: ('k, 'k Dllist.Xt.node * 'v) Hashtbl.Xt.t;
    order: 'k Dllist.Xt.t;
  }
```

MCAS

```
let a = Loc.make 10 in
let b = Loc.make 52 in
let x = Loc.make 0 in

let a = Xt.get ~xt a in        CAS (a, 10, 10)
let b = Xt.get ~xt b in        CAS (b, 52, 52)
Xt.set ~xt x (b - a)           CAS (x, 0, 42)
```

MCAS specification

$$\frac{\left\{ \mathop{\Huge *}_{\ell \in \ell s} \text{loc-inv } \ell \; \iota \right\}}{\frac{\left\langle \forall vs. \mathop{\Huge *}_{\ell, v \in \ell s, vs} \ell \rightarrowtail v \right\rangle}{\frac{\texttt{mcas } \ell s \; befores \; afters, \uparrow \iota}{\frac{\left\langle \exists b. \begin{array}{l} \textbf{if } b \textbf{ then } vs = befores * \mathop{\Huge *}_{\ell, v \in \ell s, afters} \ell \rightarrowtail v \\ \textbf{else } \exists i.\; vs_i \neq befores_i * \mathop{\Huge *}_{\ell, v \in \ell s, vs} \ell \rightarrowtail v \end{array} \right\rangle}{\{ b. \text{True} \}}}}}$$

MCAS specification: taking physical equality seriously

$$
\dfrac{
\left\{ \mathop{\Huge\ast}_{\ell \in \ell s} \text{loc-inv } \ell\ \iota \right\}
}{
\dfrac{
\left\langle \forall vs.\ \mathop{\Huge\ast}_{\ell, v \in \ell s, vs} \ell \rightarrowtail v \right\rangle
}{
\dfrac{
\texttt{mcas}\ \ell s\ \mathit{befores}\ \mathit{afters},\ \uparrow \iota
}{
\dfrac{
\left\langle \exists b.\ \begin{array}{l} \textbf{if } b \textbf{ then } vs \approx \mathit{befores} * \mathop{\Huge\ast}_{\ell, v \in \ell s, \mathit{afters}} \ell \rightarrowtail v \\ \textbf{else } \exists i.\ vs_i \not\approx \mathit{befores}_i * \mathop{\Huge\ast}_{\ell, v \in \ell s, vs} \ell \rightarrowtail v \end{array} \right\rangle
}{
\{ b.\ \text{True} \}
}
}
}
}
$$

MCAS specification: read-only locations

$$\frac{\left\{ \mathop{\Huge *}_{\ell \in \ell s} \text{loc-inv } \ell\ \iota * \mathop{\Huge *}_{\ell \in \ell s} \text{loc-inv } \ell\ \iota \right\}}{\dfrac{\left\langle \forall ws, vs.\ \mathop{\Huge *}_{\ell, v \in \ell s, ws} \ell \rightarrowtail v * \mathop{\Huge *}_{\ell, v \in \ell s, vs} \ell \rightarrowtail v \right\rangle}{\dfrac{\texttt{mcas}\ \ell s\ \ell s\ \mathit{befores}\ \mathit{afters},\ \uparrow \iota}{\dfrac{\left\langle \exists b.\ \mathop{\Huge *}_{\ell, v \in \ell s, ws} \ell \rightarrowtail v * \begin{array}{l} \textbf{if } b \textbf{ then } vs \approx \mathit{befores} * \mathop{\Huge *}_{\ell, v \in \ell s, \mathit{afters}} \ell \rightarrowtail v \\ \textbf{else } (\ell s \neq [\,] \lor \exists i.\ vs_i \not\approx \mathit{befores}_i) * \mathop{\Huge *}_{\ell, v \in \ell s, vs} \ell \rightarrowtail v \end{array} \right\rangle}{\{\ b.\ \text{True}\ \}}}}}$$

MCAS specification: relaxed memory

$$\left\{ \sqsupseteq \mathcal{W} * \mathop{\ast}_{\ell \in \ell s} \text{loc-inv } \ell\ \iota \right\}$$

$$\left\langle \forall vs, \mathcal{V}s.\ \mathop{\ast}_{\ell, v, \mathcal{V} \in \ell s, vs, \mathcal{V}s} \ell \rightarrowtail (v, \mathcal{V}) \right\rangle$$

$$\texttt{mcas}\ \ell s\ \mathit{befores}\ \mathit{afters}, \uparrow \iota$$

$$\left\langle \exists b.\ \begin{array}{l} \textbf{if } b \textbf{ then } vs \approx \mathit{befores} * \mathop{\ast}_{\ell, v, \mathcal{V} \in \ell s, \mathit{afters}, \mathcal{V}s} \ell \rightarrowtail (v, \mathcal{V} \sqcup \mathcal{W}) \\ \textbf{else } \exists i.\ vs_i \not\approx \mathit{befores}_i * \mathop{\ast}_{\ell, v, \mathcal{V} \in \ell s, vs, \mathcal{V}s} \ell \rightarrowtail (v, \mathcal{V}) \end{array} \right\rangle$$

$$\left\{ b.\ \textbf{if } b \textbf{ then } \mathop{\ast}_{\mathcal{V} \in \mathcal{V}s} \sqsupseteq \mathcal{V} \textbf{ else } \text{True} \right\}$$

MCAS algorithm: Harris, Fraser & Pratt (2002)

A Practical Multi-Word Compare-and-Swap Operation

Timothy L. Harris, Keir Fraser and Ian A. Pratt

University of Cambridge Computer Laboratory, Cambridge, UK
{tim.harris,keir.fraser,ian.pratt}@cl.cam.ac.uk

Abstract. Work on non-blocking data structures has proposed extending processor designs with a compare-and-swap primitive, `CAS2`, which acts on two arbitrary memory locations. Experience suggested that current operations, typically single-word compare-and-swap (`CAS1`), are not expressive enough to be used alone in an efficient manner. In this paper we build `CAS2` from `CAS1` and, in fact, build an arbitrary multi-word compare-and-swap (`CASN`). Our design requires only the primitives available on contemporary systems, reserves a small and constant amount of space in each word updated (either 0 or 2 bits) and permits non-overlapping updates to occur concurrently. This provides compelling evidence that current primitives are not only universal in the theoretical sense introduced by Herlihy, but are also universal in their use as foundations for practical algorithms. This provides a straightforward mechanism for deploying many of the interesting non-blocking data structures presented in the literature that have previously required `CAS2`.

1 Introduction

Verified RDCSS by Jung *et al.*

The Future is Ours: Prophecy Variables in Separation Logic

RALF JUNG, MPI-SWS, Germany
RODOLPHE LEPIGRE, MPI-SWS, Germany
GAURAV PARTHASARATHY, ETH Zurich, Switzerland and MPI-SWS, Germany
MARIANNA RAPOPORT, University of Waterloo, Canada and MPI-SWS, Germany
AMIN TIMANY, imec-DistriNet, KU Leuven, Belgium
DEREK DREYER, MPI-SWS, Germany
BART JACOBS, imec-DistriNet, KU Leuven, Belgium

Early in the development of Hoare logic, Owicki and Gries introduced *auxiliary variables* as a way of encoding information about the *history* of a program's execution that is useful for verifying its correctness. Over a decade later, Abadi and Lamport observed that it is sometimes also necessary to know in advance what a program will do in the *future*. To address this need, they proposed *prophecy variables*, originally as a proof technique for refinement mappings between state machines. However, despite the fact that prophecy variables are a clearly useful reasoning mechanism, there is (surprisingly) almost no work that attempts to integrate them into Hoare logic. In this paper, we present the first account of prophecy variables in a Hoare-style program logic that is flexible enough to verify *logical atomicity* (a relative of linearizability) for classic examples from the concurrency literature like RDCSS and the Herlihy-Wing queue. Our account is formalized in the Iris framework for separation logic in Coq. It makes essential use of *ownership* to encode the exclusive right to resolve a prophecy, which in turn lets us enforce soundness of prophecies with a very simple set of proof rules.

CCS Concepts: • **Theory of computation** → **Separation logic**; *Programming logic*; Operational semantics.

Additional Key Words and Phrases: Prophecy variables, separation logic, logical atomicity, linearizability, Iris

ACM Reference Format:
Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The Future is Ours: Prophecy Variables in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 45 (January 2020), 32 pages. https://doi.org/10.1145/3371113

45

1 INTRODUCTION

When proving correctness of a program *P*, it is often easier and more natural to reason *forward*—that

MCAS algorithm: Guerraoui, Kogan, Marathe & Zablotchi (2020)

Efficient Multi-Word Compare and Swap

Rachid Guerraoui
EPFL, Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Alex Kogan
Oracle Labs, Burlington, MA, USA
alex.kogan@oracle.com

Virendra J. Marathe
Oracle Labs, Burlinton, MA, USA
virendra.marathe@oracle.com

Igor Zablotchi[1]
EPFL, Lausanne, Switzerland
igor.zablotchi@epfl.ch

Abstract

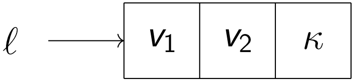
Atomic lock-free multi-word compare-and-swap (MCAS) is a powerful tool for designing concurrent algorithms. Yet, its widespread usage has been limited because lock-free implementations of MCAS make heavy use of expensive compare-and-swap (CAS) instructions. Existing MCAS implementations indeed use at least $2k+1$ CASes per k-CAS. This leads to the natural desire to minimize the number of CASes required to implement MCAS.

We first prove in this paper that it is impossible to "pack" the information required to perform a k-word CAS (k-CAS) in less than k locations to be CASed. Then we present the first algorithm that requires $k+1$ CASes per call to k-CAS in the common uncontended case. We implement our algorithm and show that it outperforms a state-of-the-art baseline in a variety of benchmarks in most considered workloads. We also present a durably linearizable (persistent memory friendly) version of our MCAS algorithm using only 2 persistence fences per call, while still only requiring $k+1$ CASes per k-CAS.

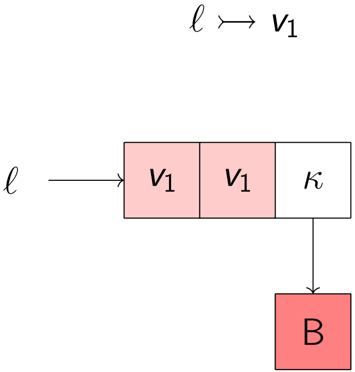
2012 ACM Subject Classification Theory of computation → Concurrent algorithms

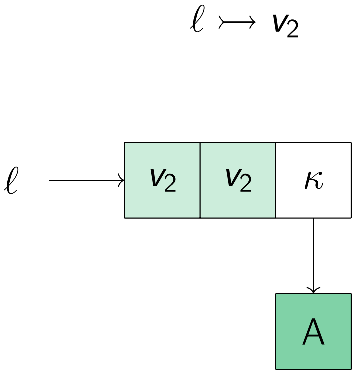
MCAS location

$$\ell \rightarrowtail v_1 \text{ or } v_2$$



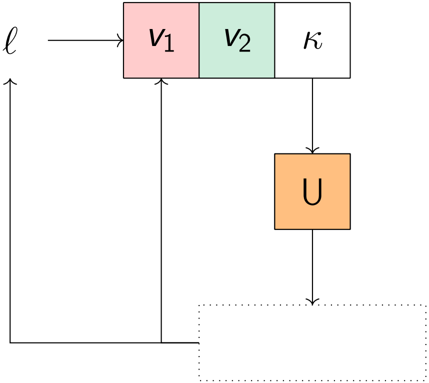
Finished MCAS



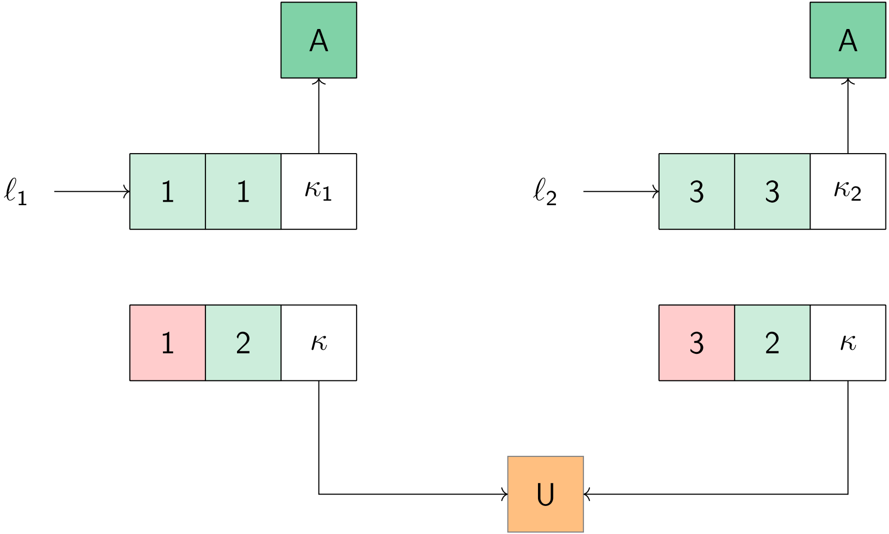


Undetermined MCAS

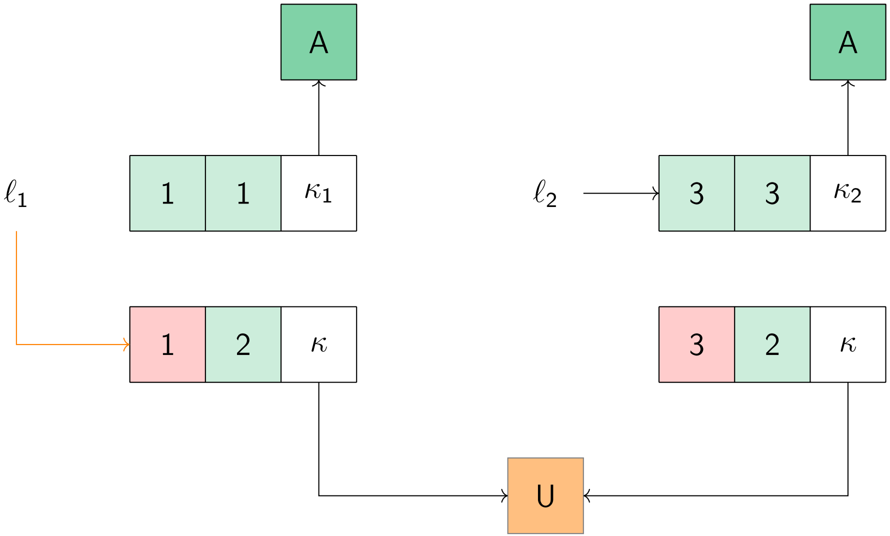
$$\ell \rightarrowtail v_1$$



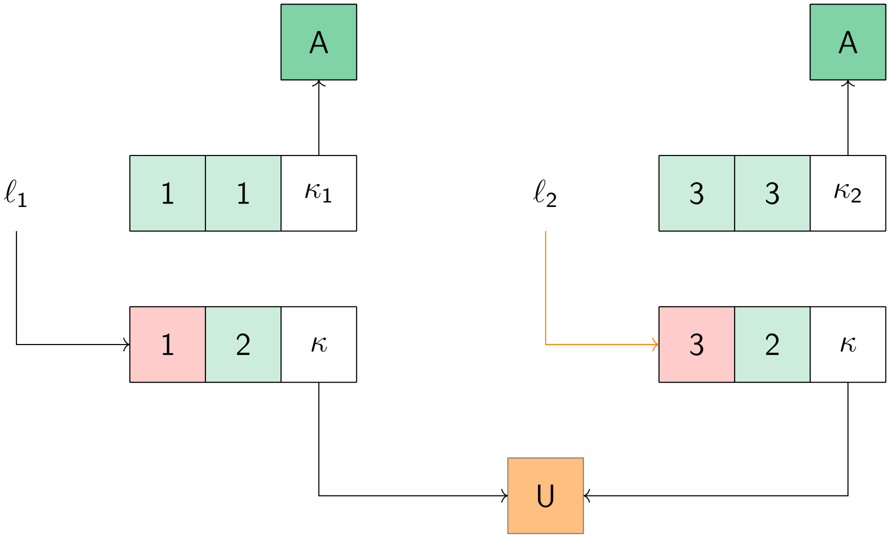
MCAS algorithm



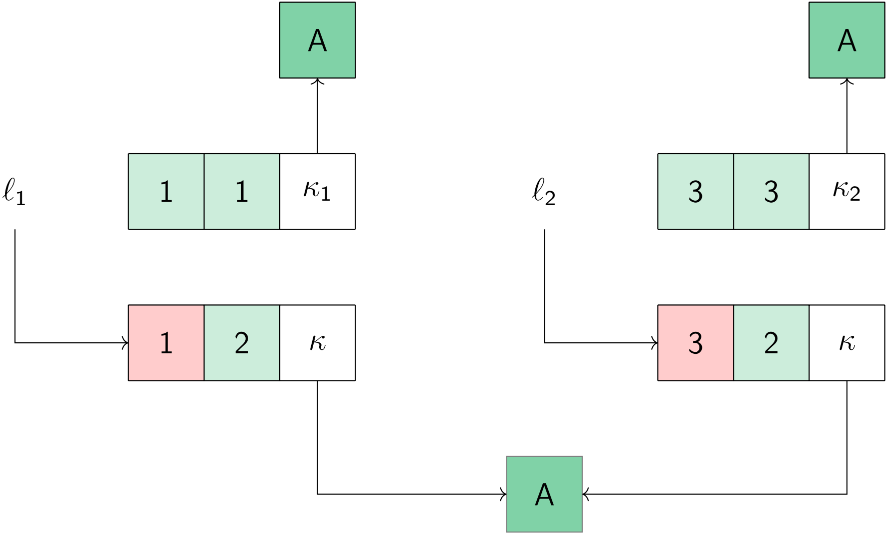
MCAS algorithm



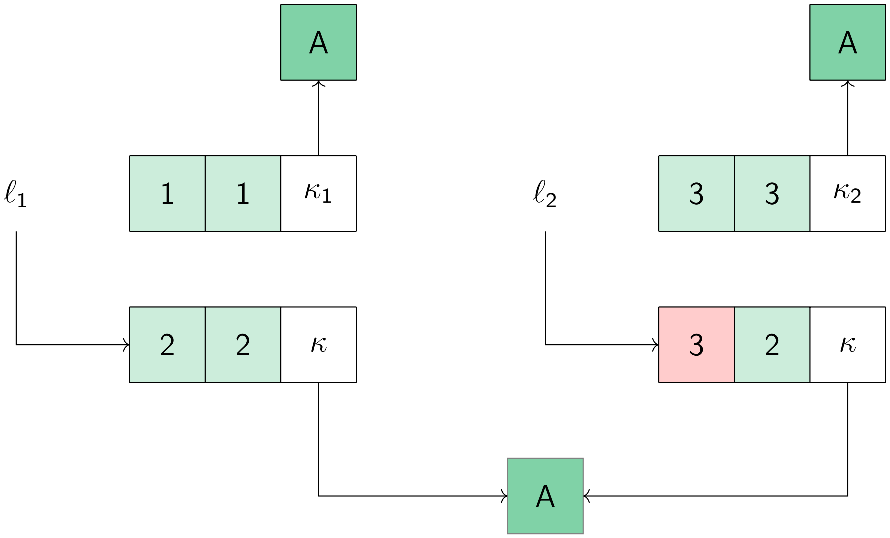
MCAS algorithm



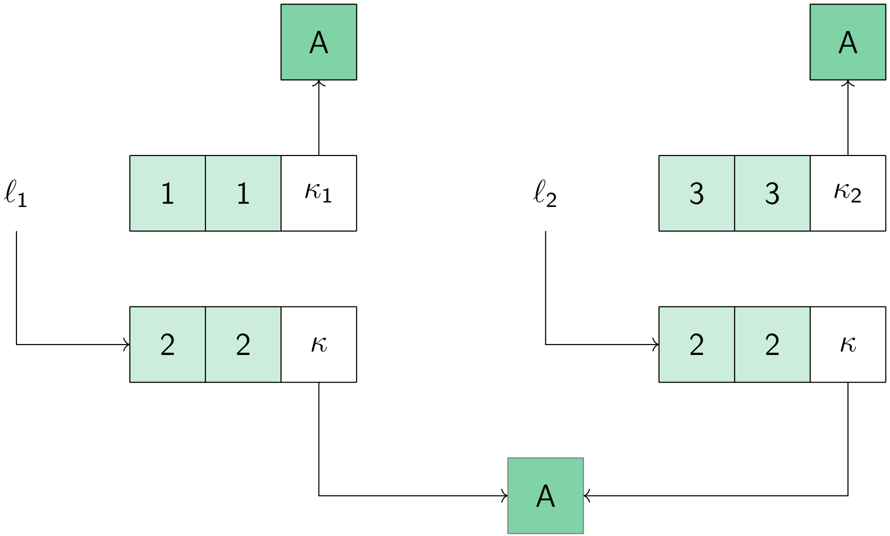
MCAS algorithm



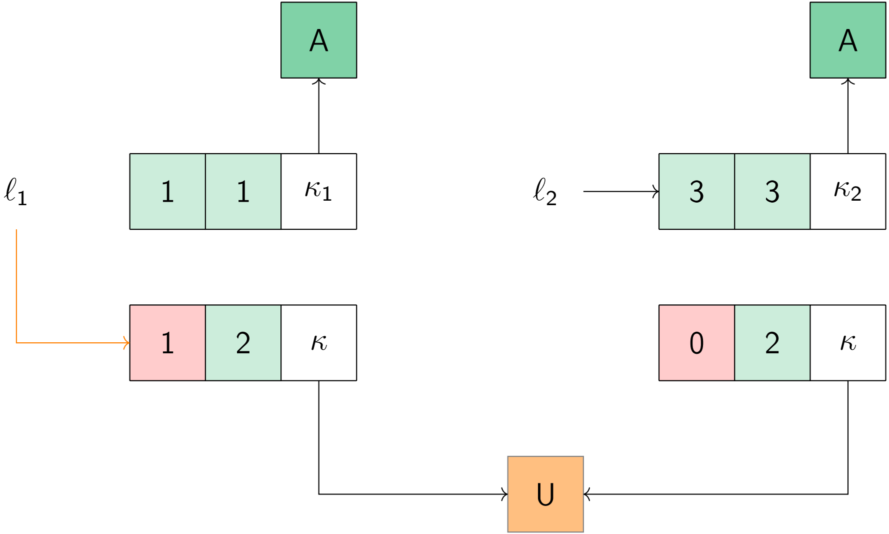
MCAS algorithm



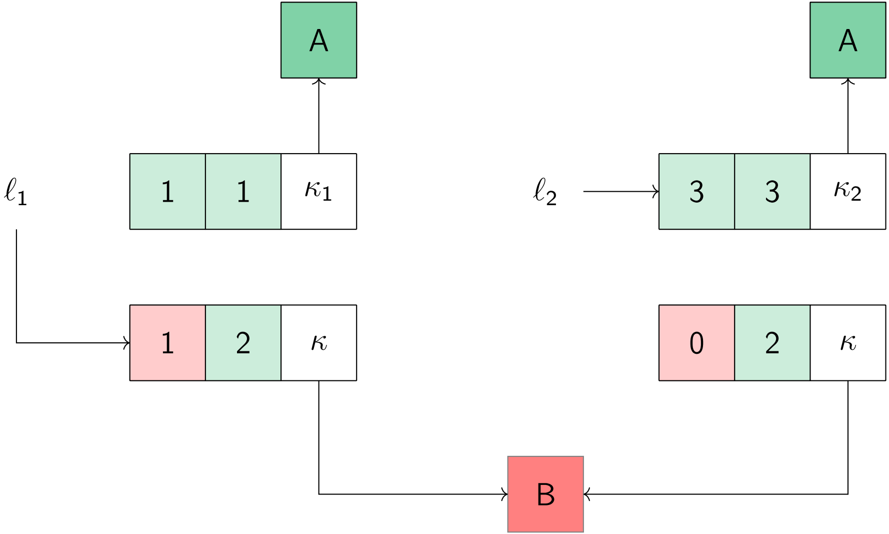
MCAS algorithm



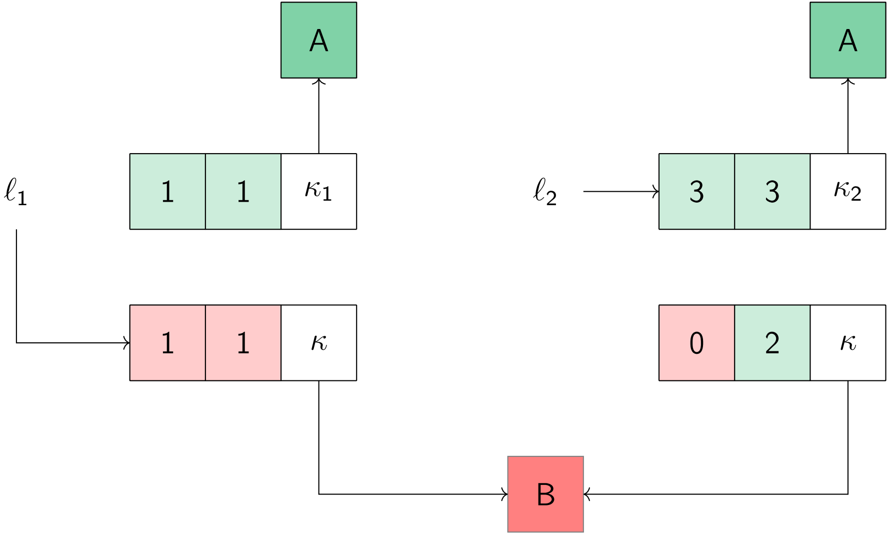
MCAS algorithm



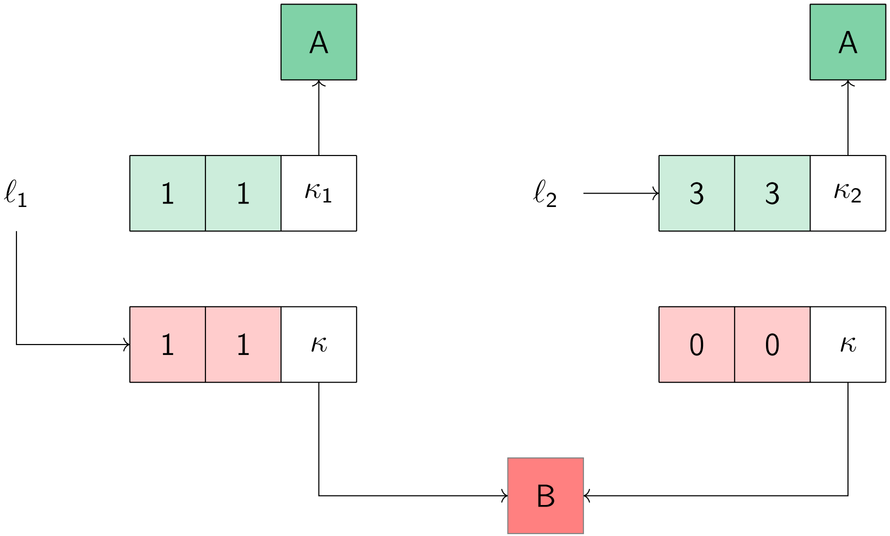
MCAS algorithm



MCAS algorithm



MCAS algorithm



Coupling with semi-automated verification (Gospel)

GOSPEL — Providing OCaml with a Formal Specification Language

Arthur Charguéraud[1,2], Jean-Christophe Filliâtre[3,1], Cláudio Lourenço[3,1], and Mário Pereira[4]

[1] Inria
[2] Université de Strasbourg, CNRS, ICube
[3] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[4] NOVA LINCS & DI, FCT, Universidade Nova de Lisboa, Portugal

Abstract. This paper introduces GOSPEL, a behavioral specification language for OCaml. It is designed to enable modular verification of data structures and algorithms. GOSPEL is a contract-based, strongly typed language, with a formal semantics defined by means of translation into Separation Logic. Compared with writing specifications directly in Separation Logic, GOSPEL provides a high-level syntax that greatly improves conciseness and makes it accessible to programmers with no familiarity with Separation Logic. Although GOSPEL has been developed for specifying OCaml code, we believe that many aspects of its design could apply to other programming languages. This paper presents the design and semantics of GOSPEL, and reports on its application for the development

Thank you for your attention!