

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

Clément Allain
Gabriel Scherer

March 20, 2025

Introduction

Zoo in practice

Zoo features

Physical equality

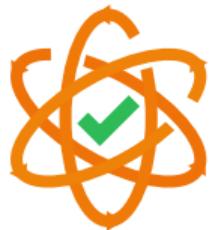
Structural equality

Specimen: Kcas (ongoing work)

Future work

Context

Verification of *fine-grained concurrent OCaml 5* programs



Saturn
Kcas



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

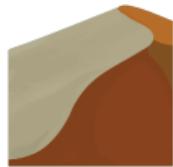
Specimen: Kcas (ongoing work)

Future work

Zoo in practice



OCaml



DUNE

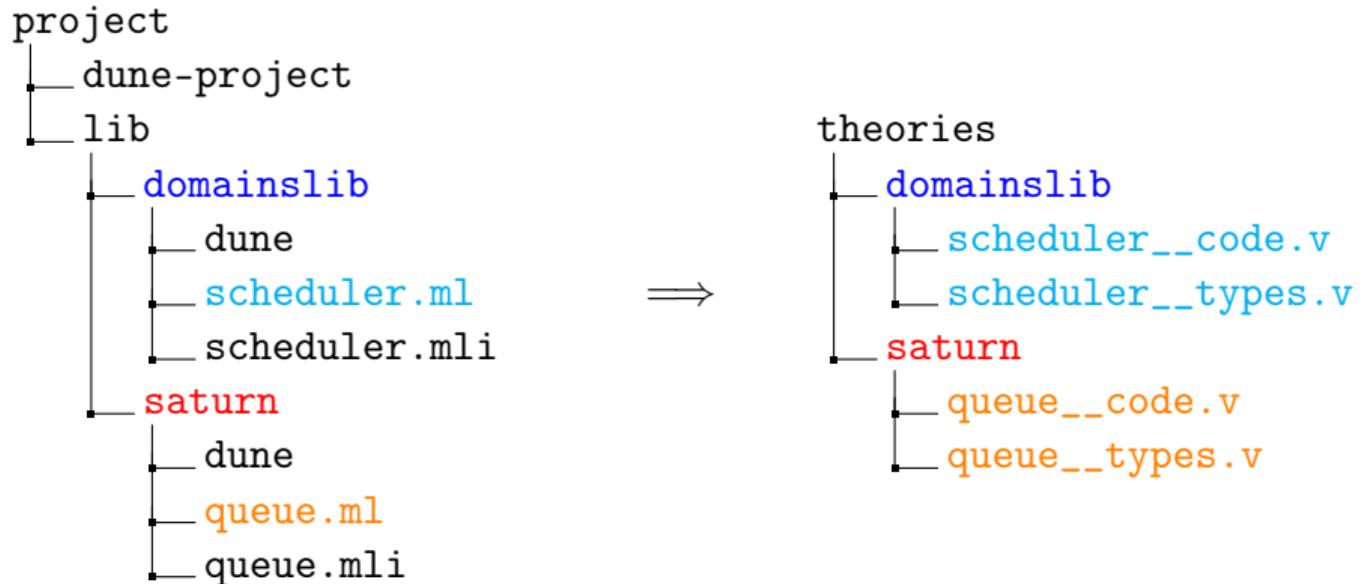
ocaml2zoo
→



Zoo

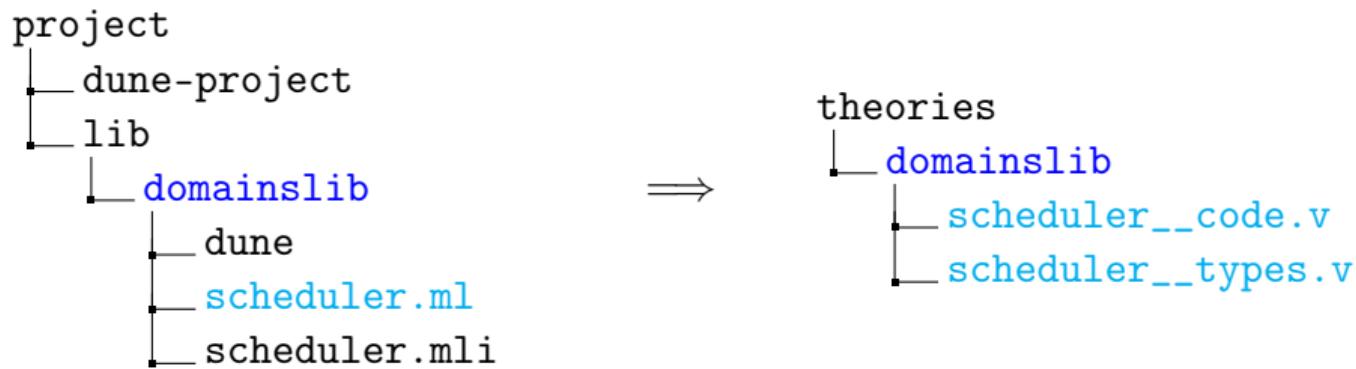


Zoo in practice



```
$ ocaml2zoo project theories
```

Zoo in practice



```
$ ocaml2zoo project theories
```

Zoo in practice

```
Lemma stack_push_spec_seq t  $\iota$  v :  
{{{  
  stack_model t vs  
}}}  
  stack_push t v  
{{{  
  RET ();  
  stack_model t (v :: vs)  
}}}.
```

Proof.

...

Qed.

```
Lemma stack_push_spec_atomic t  $\iota$  v :  
<<<  
  stack_inv t  $\iota$   
|  $\forall$  vs,  
  stack_model t vs  
>>>  
  stack_push t v @  $\uparrow$  $\iota$   
<<<  
  stack_model t (v :: vs)  
| RET (); True  
>>>.  
Proof.  
...  
Qed.
```

Introduction

Zoo in practice

Zoo features

Physical equality

Structural equality

Specimen: Kcas (ongoing work)

Future work

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

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

(* boilerplate *)

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

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 e1 e2`

`e1 <- e2`

`Atomic.exchange e1 e2`

`Xchg e1.[contents] e2`

`Atomic.compare_and_set e1 e2 e3`

`CAS e1.[contents] e2 e3`

`Atomic.fetch_and_add e1 e2`

`FAA e1.[contents] e2`

`type t = { ...; mutable f: τ [@atomic]; ... }`

`Atomic.Loc.exchange [%atomic.loc e1.f] e2`

`Xchg e1.[f] e2`

`Atomic.Loc.compare_and_set [%atomic.loc e1.f] e2 e3`

`CAS e1.[f] e2 e3`

`Atomic.Loc.fetch_and_add [%atomic.loc e1.f] e2`

`FAA e1.[f] e2`

<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

Introduction

Zoo in practice

Zoo features

Physical equality

Structural equality

Specimen: Kcas (ongoing work)

Future work

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  $\iota$  v :
```

```
<<<
```

```
  stack_inv t  $\iota$ 
```

```
|  $\forall \forall$  vs,
```

```
  stack_model t vs
```

```
>>>
```

```
  stack_push t v @  $\uparrow \iota$ 
```

```
<<<
```

```
  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
  )
```

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 Lviron
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)
```

Introduction

Zoo in practice

Zoo features

Physical equality

Structural equality

Specimen: Kcas (ongoing work)

Future work

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.
```

Introduction

Zoo in practice

Zoo features

Physical equality

Structural equality

Specimen: Kcas (ongoing work)

Future work

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.  
in
```



Vesa Karvonen
The main author of Kcas

```
Xt.commit { tx }
```

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

$$\frac{\frac{\frac{\left\{ \begin{array}{c} \ast_{\ell \in ls} \text{ loc-inv } \ell \ i \\ \forall vs. \ \ast_{\ell, v \in ls, vs} \ell \rightarrowtail v \end{array} \right\}}{\text{mcas } ls \text{ before } afters, \uparrow i}}{\text{if } b \text{ then } vs = \text{before} * \ast_{\ell, v \in ls, \text{afters}} \ell \rightarrowtail v \\ \text{else } \exists i. \ vs_i \neq \text{before}_i * \ast_{\ell, v \in ls, vs} \ell \rightarrowtail v}}{\{ b. \text{ True} \}}$$

MCAS specification: taking physical equality seriously

$$\frac{\frac{\frac{\left\{ \begin{array}{c} * \text{ loc-inv } \ell \ i \\ \ell \in ls \end{array} \right\}}{\left\langle \forall vs. \begin{array}{c} * \\ \ell, v \in ls, vs \end{array} \end{right\rangle} \text{ mcas } ls \text{ before } afters, \uparrow i}{\left\langle \begin{array}{c} \text{if } b \text{ then } vs \approx \text{before} * \\ \exists b. \text{ else } \exists i. vs_i \not\approx \text{before}_i * \end{array} \right. \begin{array}{c} * \\ \ell, v \in ls, afters \\ \ell \mapsto v \end{array} \right\rangle \text{ if } b \text{ True}}{\left\{ b. \text{ True} \right\}}$$

MCAS specification: read-only locations

$$\left\{ \underset{\ell \in ls}{*} \text{loc-inv } \ell \iota * \underset{\ell \in ls}{*} \text{loc-inv } \ell \iota \right\}$$

$$\left\langle \forall ws, vs. \underset{\ell, v \in ls, ws}{*} \ell \rightarrowtail v * \underset{\ell, v \in ls, vs}{*} \ell \rightarrowtail v \right\rangle$$

mcas *ls ls befores afters*, $\uparrow \iota$

$$\text{if } b \text{ then } vs \approx \text{belfores} * \underset{\ell, v \in ls, afters}{*} \ell \rightarrowtail v$$

$$\left\langle \exists b. \underset{\ell, v \in ls, ws}{*} \ell \rightarrowtail v * \text{else } (\underset{\ell s \neq []}{*} \vee \exists i. vs_i \not\approx \text{belfores}_i) * \underset{\ell, v \in ls, vs}{*} \ell \rightarrowtail v \right\rangle$$

{ *b.* True }

MCAS specification: relaxed memory

$$\begin{array}{c}
 \left\{ \exists \mathcal{W} * \underset{\ell \in \text{ls}}{*} \text{loc-inv } \ell \uparrow \iota \right\} \\
 \hline
 \left\langle \forall vs, \mathcal{V}s. \underset{\ell, v, \mathcal{V} \in \text{ls}, vs, \mathcal{V}s}{*} \ell \mapsto (v, \mathcal{V}) \right\rangle \\
 \hline
 \text{mcas } \text{ls before } \text{afters}, \uparrow \iota \\
 \hline
 \left\langle \begin{array}{l} \text{if } b \text{ then } vs \approx \text{before} * \underset{\ell, v, \mathcal{V} \in \text{ls}, \text{afters}, vs}{*} \ell \mapsto (v, \mathcal{V} \sqcup \mathcal{W}) \\ \text{else } \exists i. vs_i \not\approx \text{before}_i * \underset{\ell, v, \mathcal{V} \in \text{ls}, vs, \mathcal{V}s}{*} \ell \mapsto (v, \mathcal{V}) \end{array} \right\rangle \\
 \hline
 \left\{ b. \text{if } b \text{ then } \underset{v \in vs}{*} \exists \mathcal{V} \text{ else True} \right\}
 \end{array}$$

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>

1 INTRODUCTION

When proving correctness of a program P , it is often easier and more natural to reason *forward*—that is, to start at the beginning of P 's execution and reason about how it behaves as it executes. But

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, Burlington, MA, USA

virendra.marathe@oracle.com

Igor Zablotchi¹

EPFL, Lausanne, Switzerland

igor.zablotchi@epfl.ch

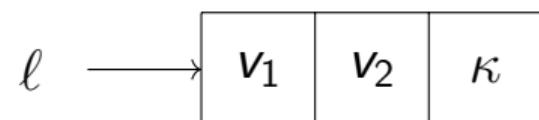
Abstract

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.

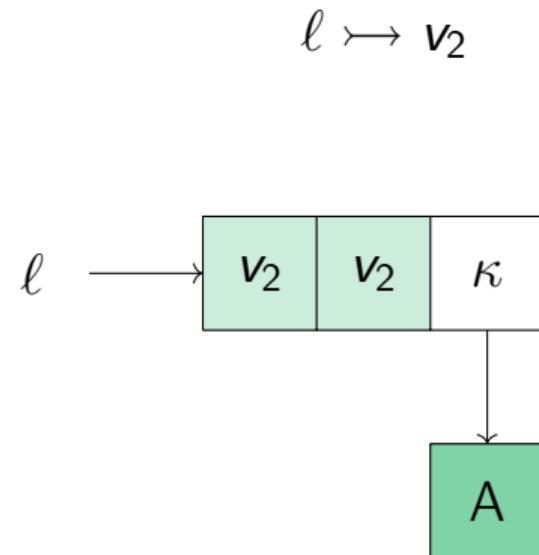
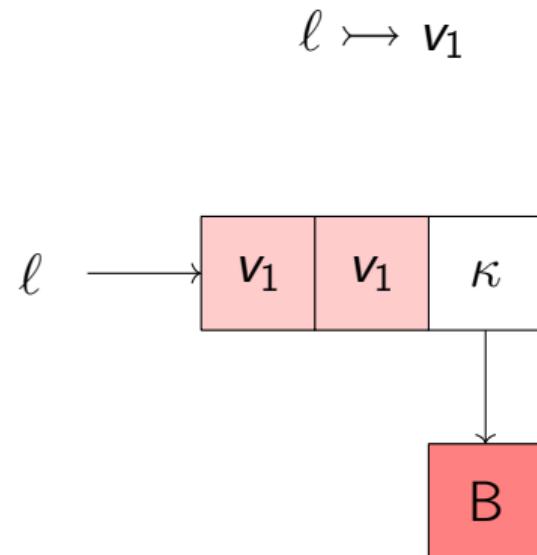
We first prove in this paper that it is impossible to “pack” the information required to perform a k -word CAS (k -CAS) in less than k locations to be CASed. Then we present the first algorithm that requires $k + 1$ CASes per call to k -CAS in the common uncontended case. We implement our algorithm and 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.

MCAS location

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

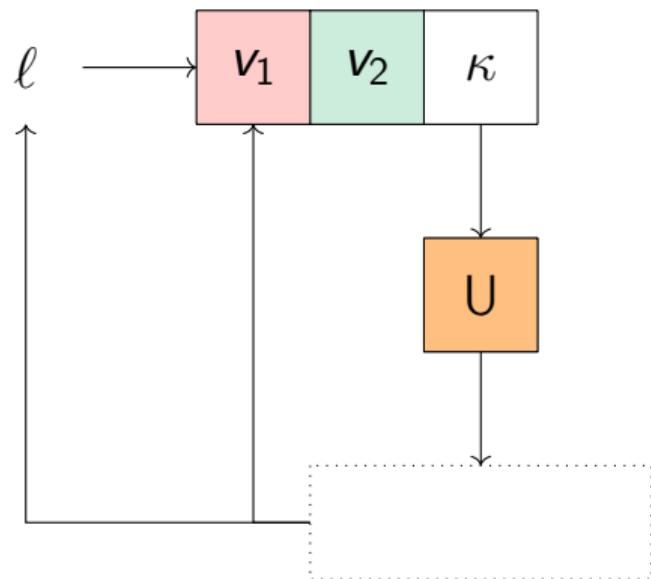


Finished MCAS

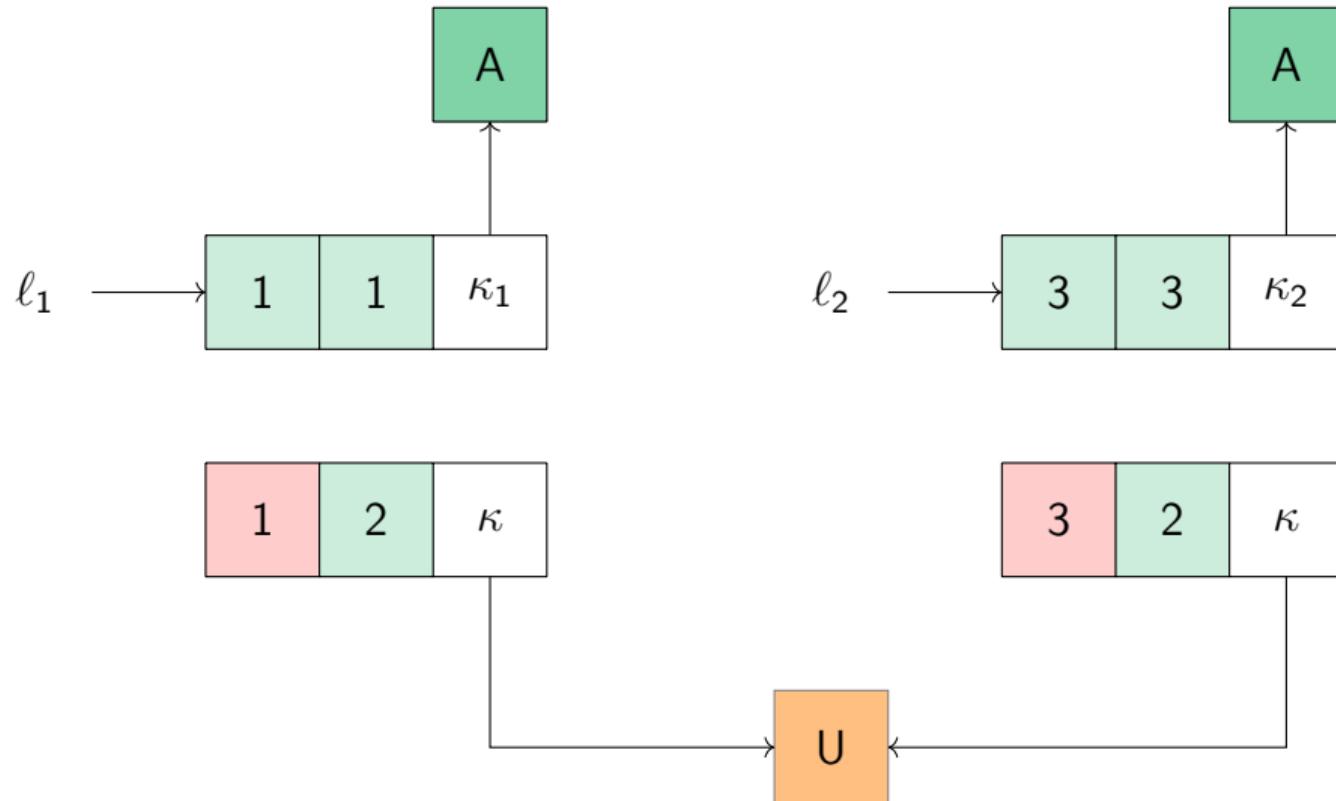


Undetermined MCAS

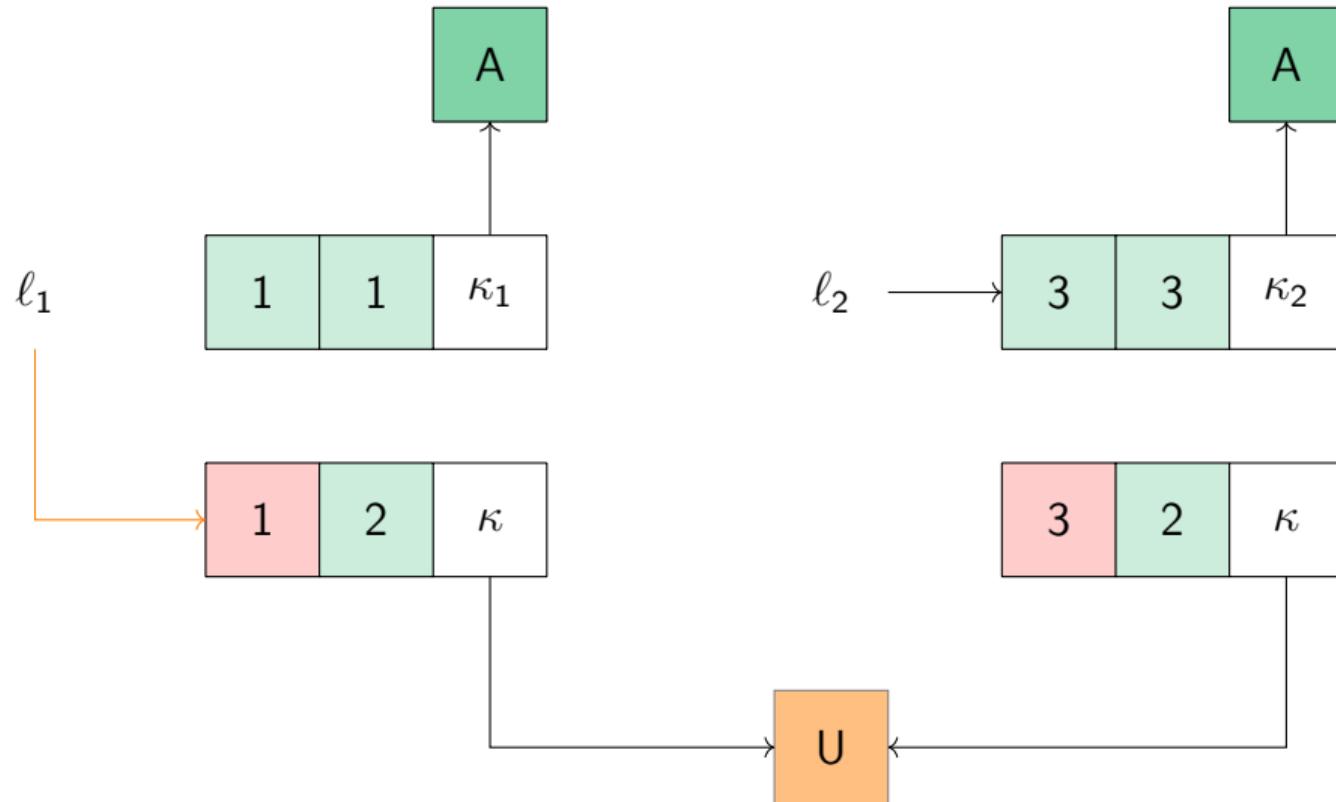
$$\ell \rightarrowtail v_1$$



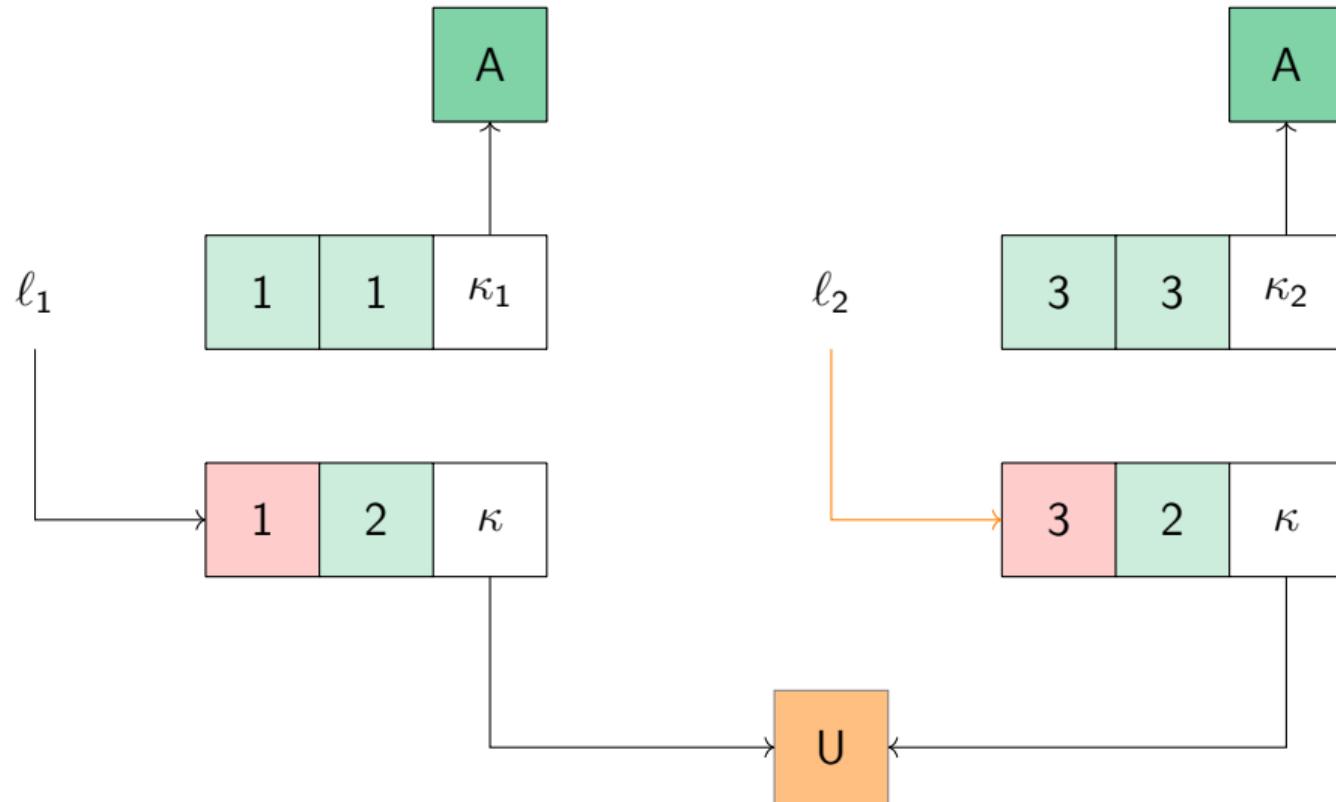
MCAS algorithm



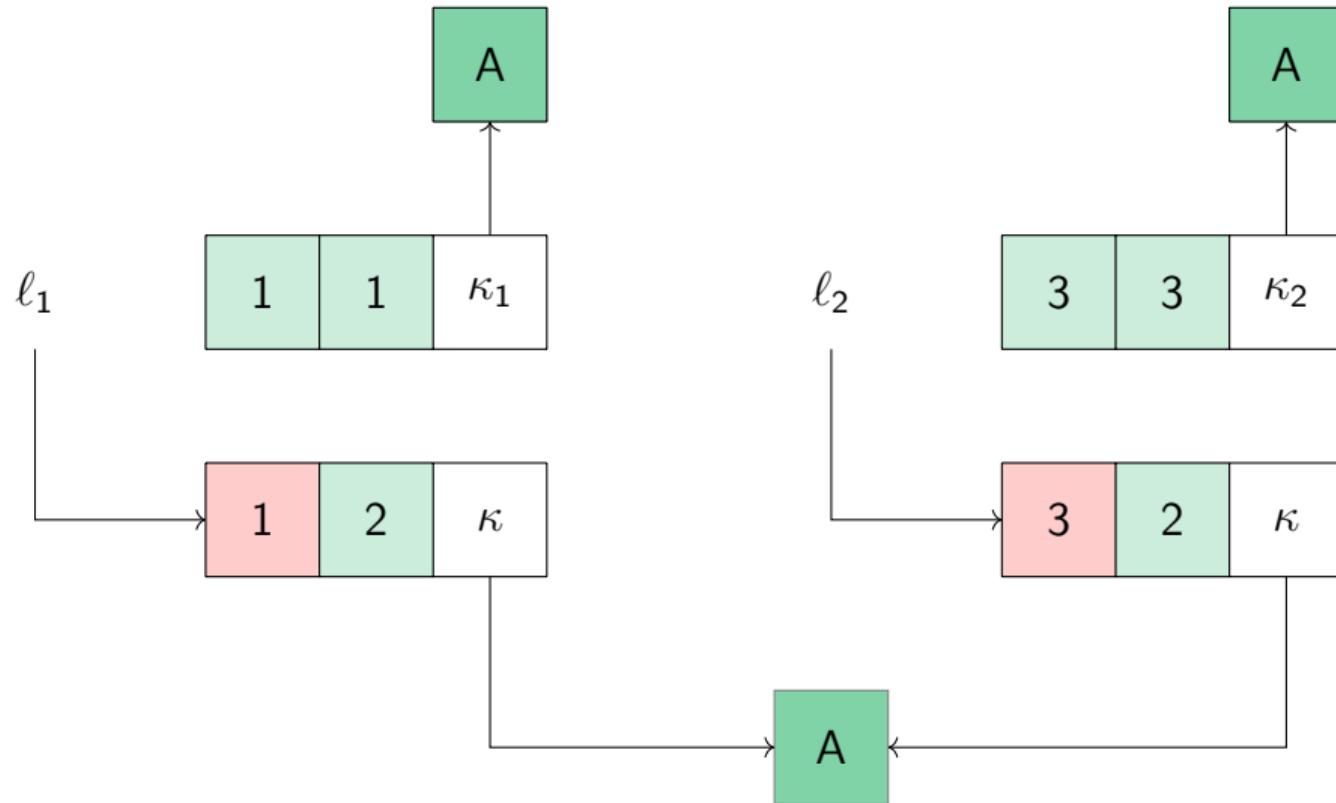
MCAS algorithm



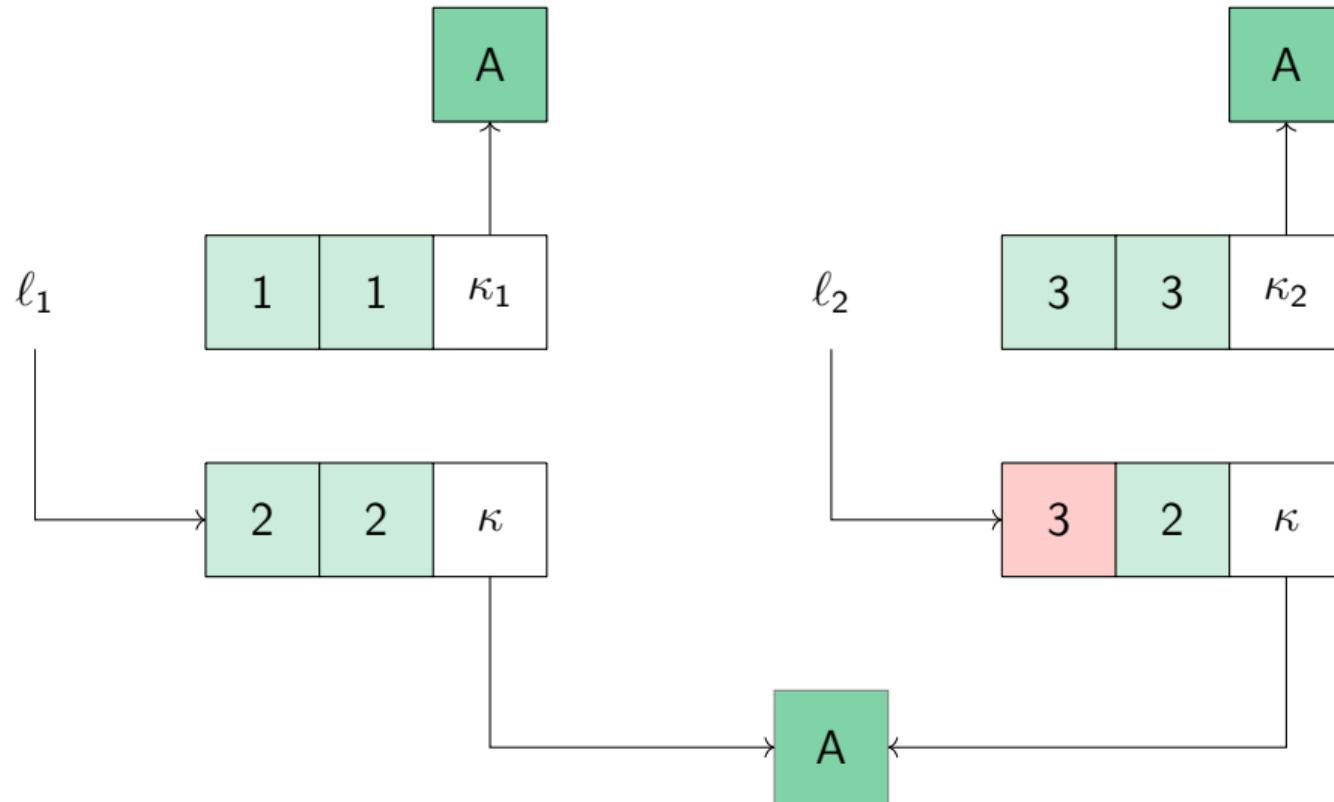
MCAS algorithm



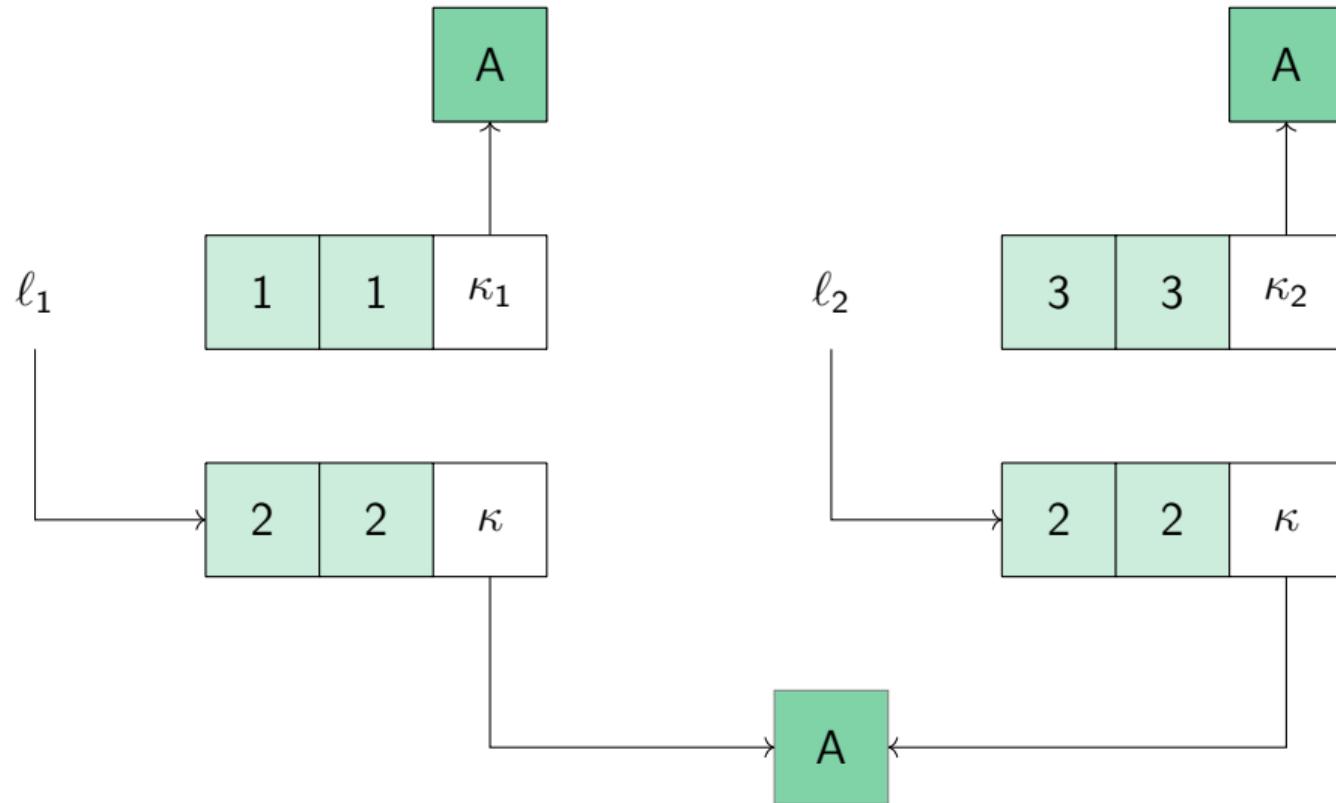
MCAS algorithm



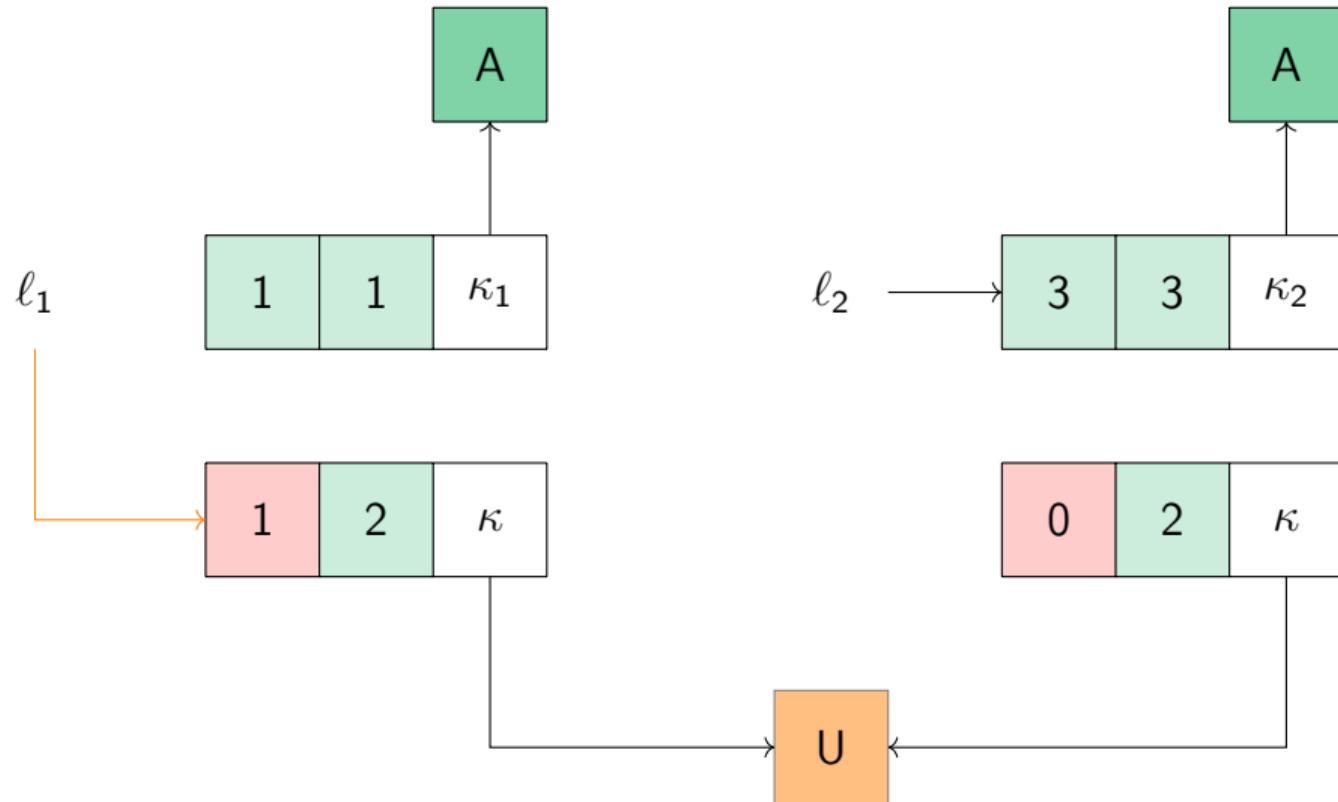
MCAS algorithm



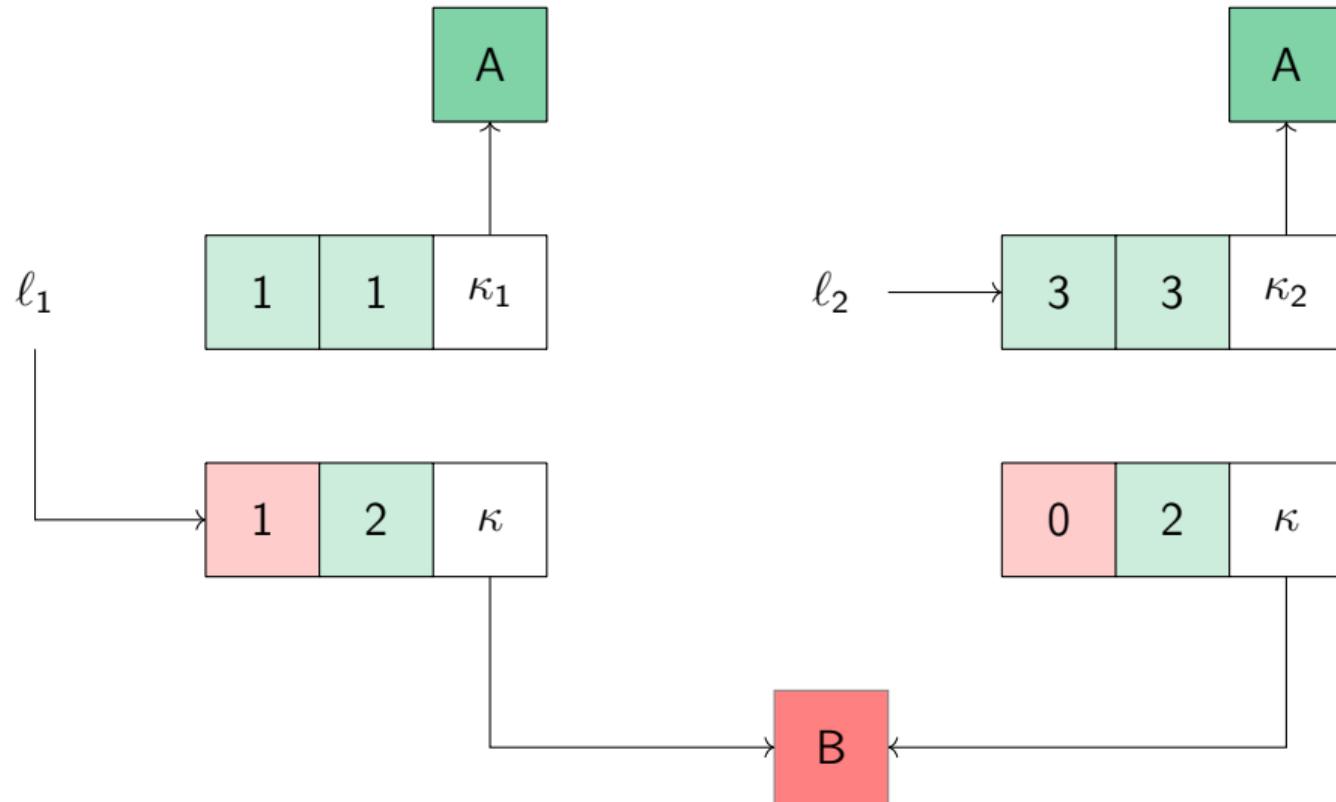
MCAS algorithm



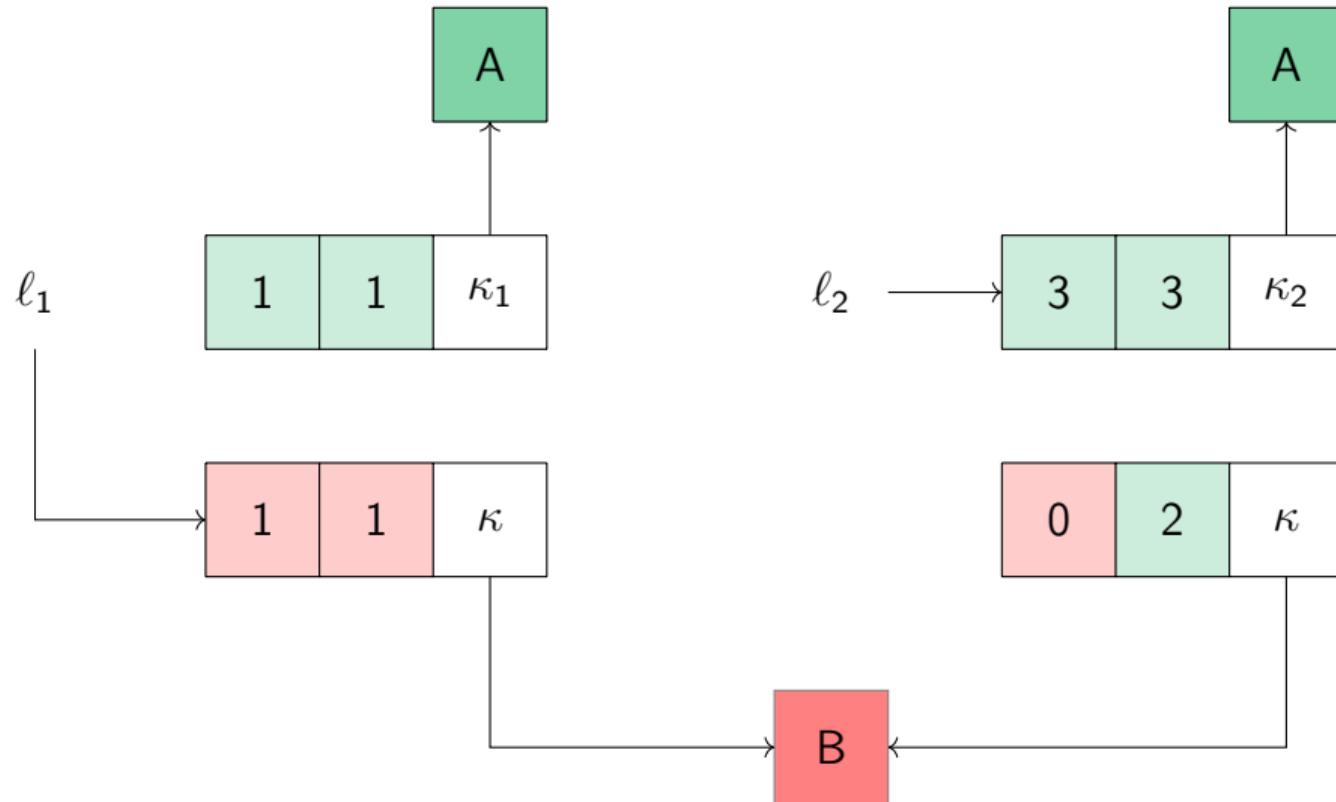
MCAS algorithm



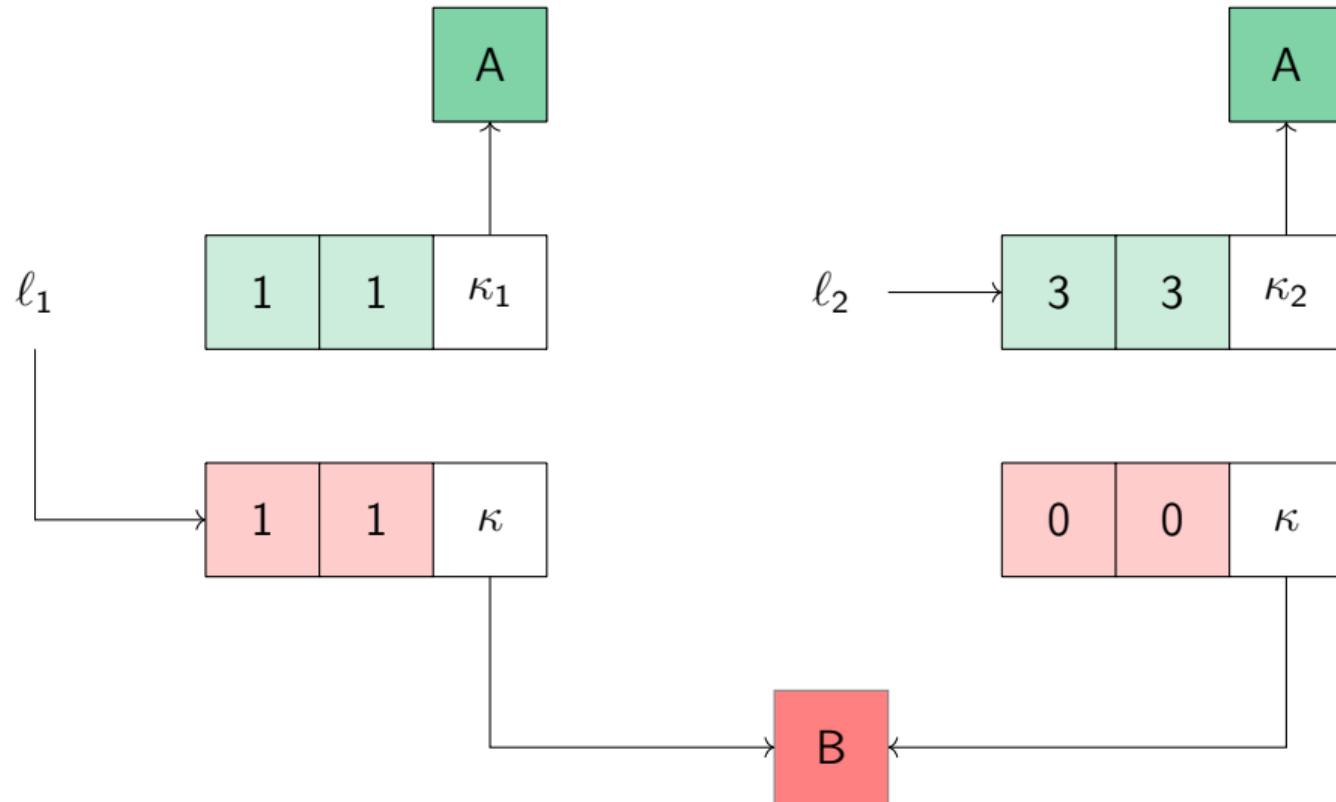
MCAS algorithm



MCAS algorithm



MCAS algorithm



Introduction

Zoo in practice

Zoo features

Physical equality

Structural equality

Specimen: Kcas (ongoing work)

Future work

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 Pereira⁴

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!