ZOOlogy of lockfree concurrent data structures

Clément Allain

INRIA Paris

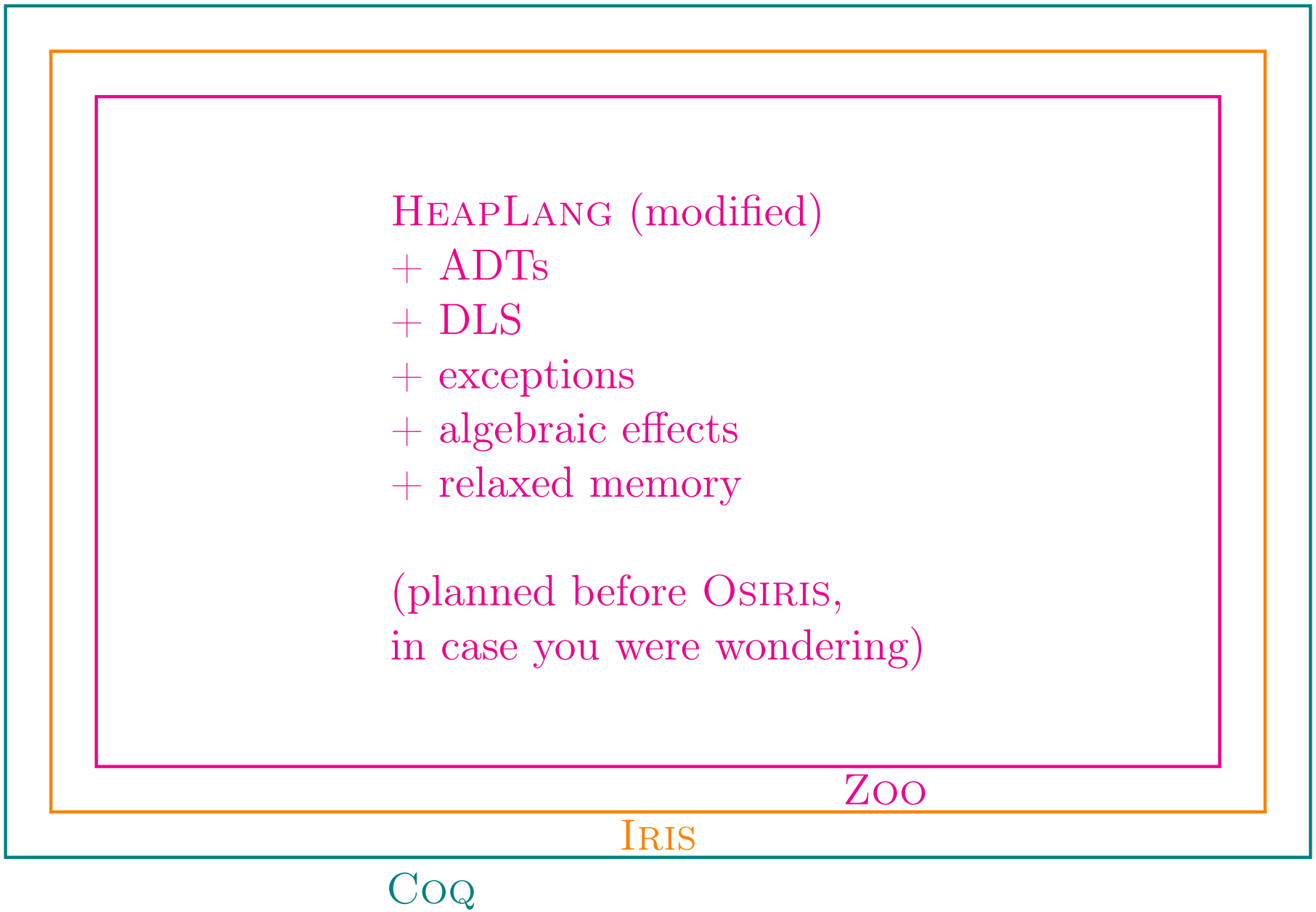
June 18, 2024

Zoology

Specimen ① : Michael-Scott queue

Specimen ② : Kcas

ZOOlogy: what is it?



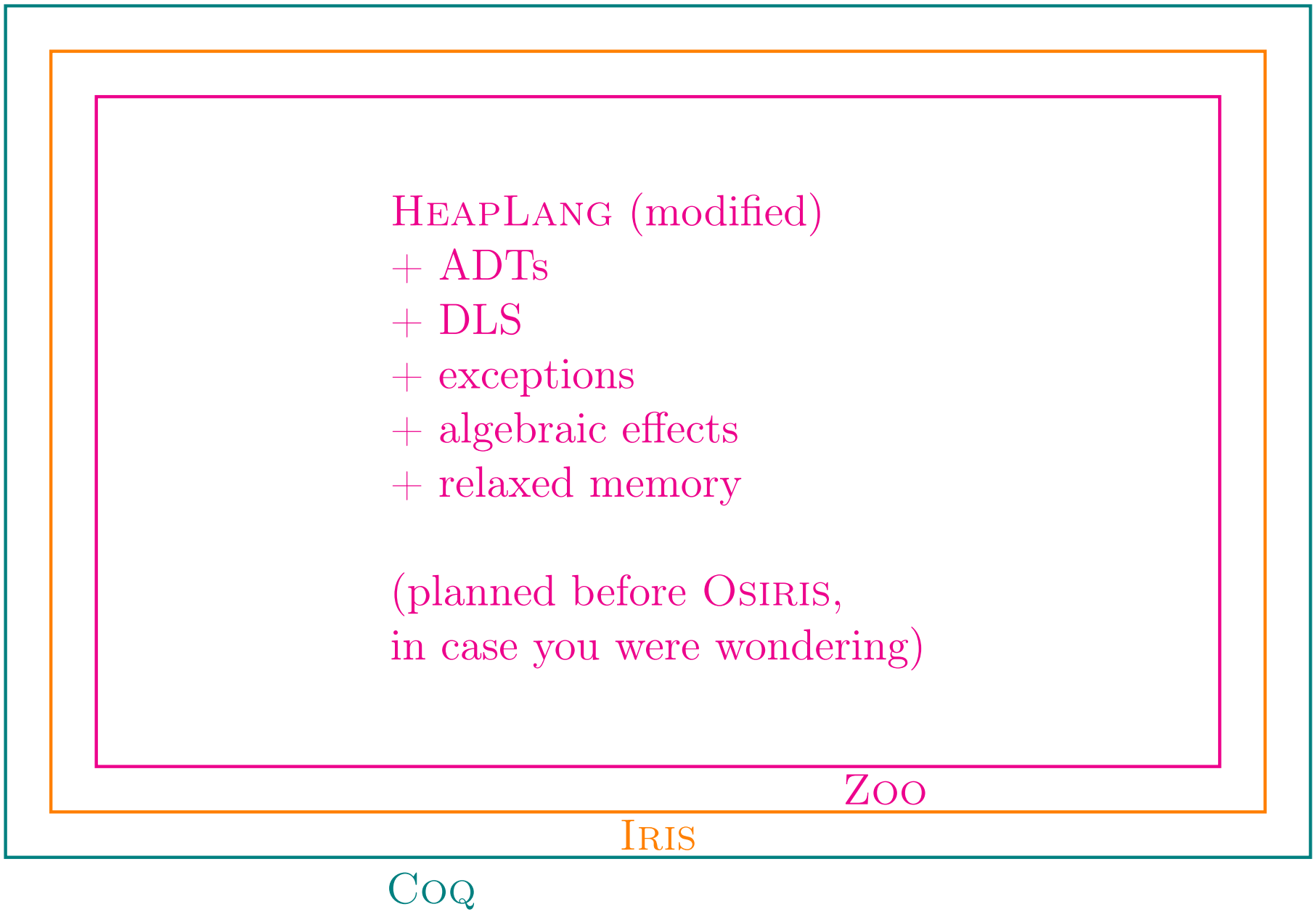
ZOOlogy: what is it?

```
MetaCoq Run (zoo_variant "list" [
  "Nil" ;
  "Cons"
]).

Definition map : val :=
  rec: "map" "fn" "xs" :=
    match: "xs" with
    | Nil =>
        §Nil
    | Cons "x" "xs" =>
        let: "y" := "fn" "x" in
        'Cons{ "y", "map" "fn" "xs" }
    end.
```

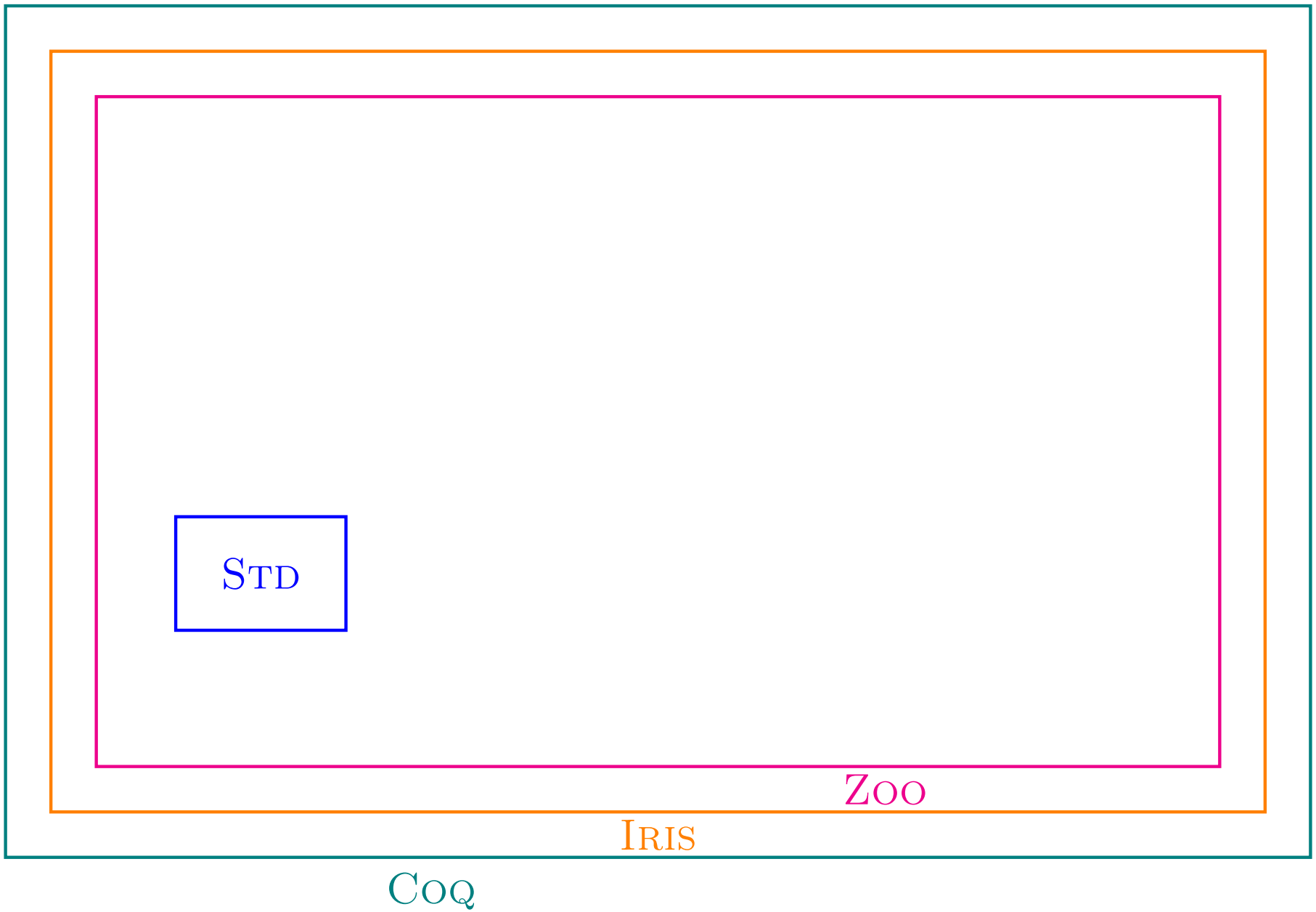
IRIS

COQ

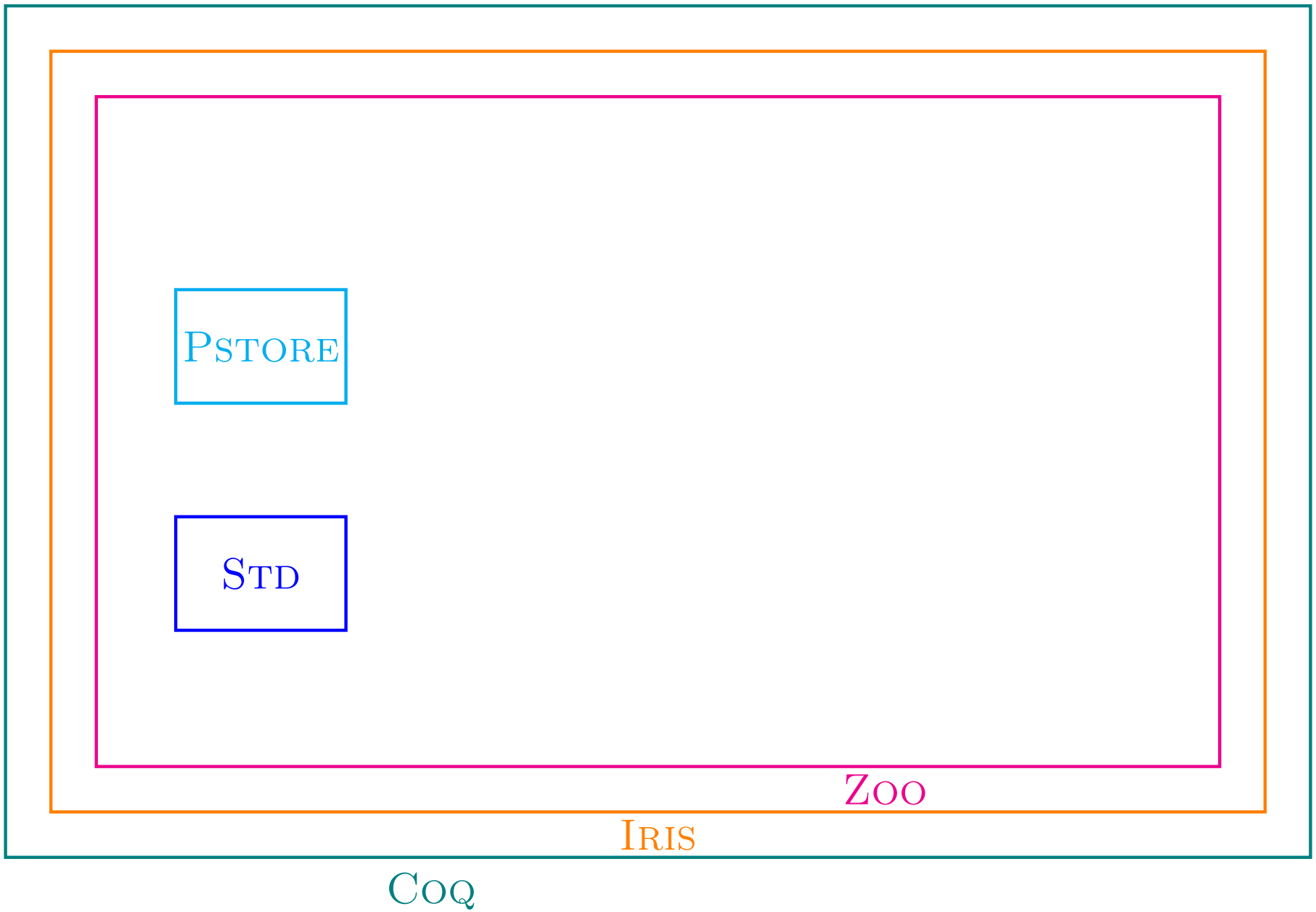
ZOOlogy: what is it?



ZOOlogy: what is it?

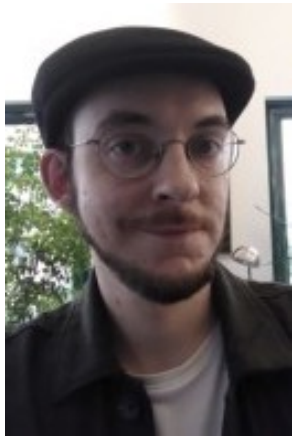


ZOOlogy: what is it?



Zoology: what is it?

Clément Allain



Basile Clément

Alexandre Moine

Gabriel Scherer

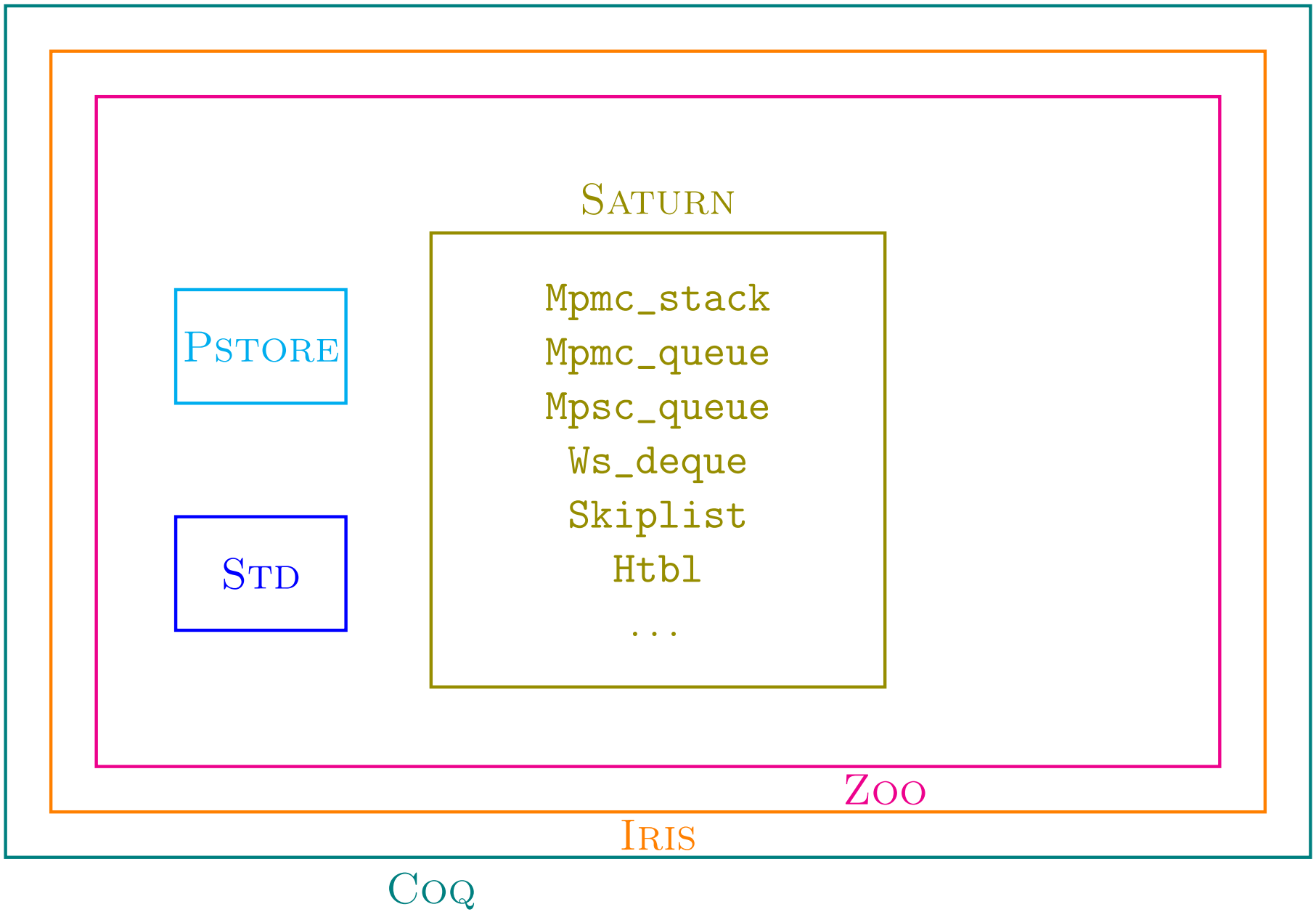
The Pstore team!

Zoo

Iris

Coq

Zoology: what is it?



ZOOlogy: what is it?

Vesa Karvonen

Carine Morel

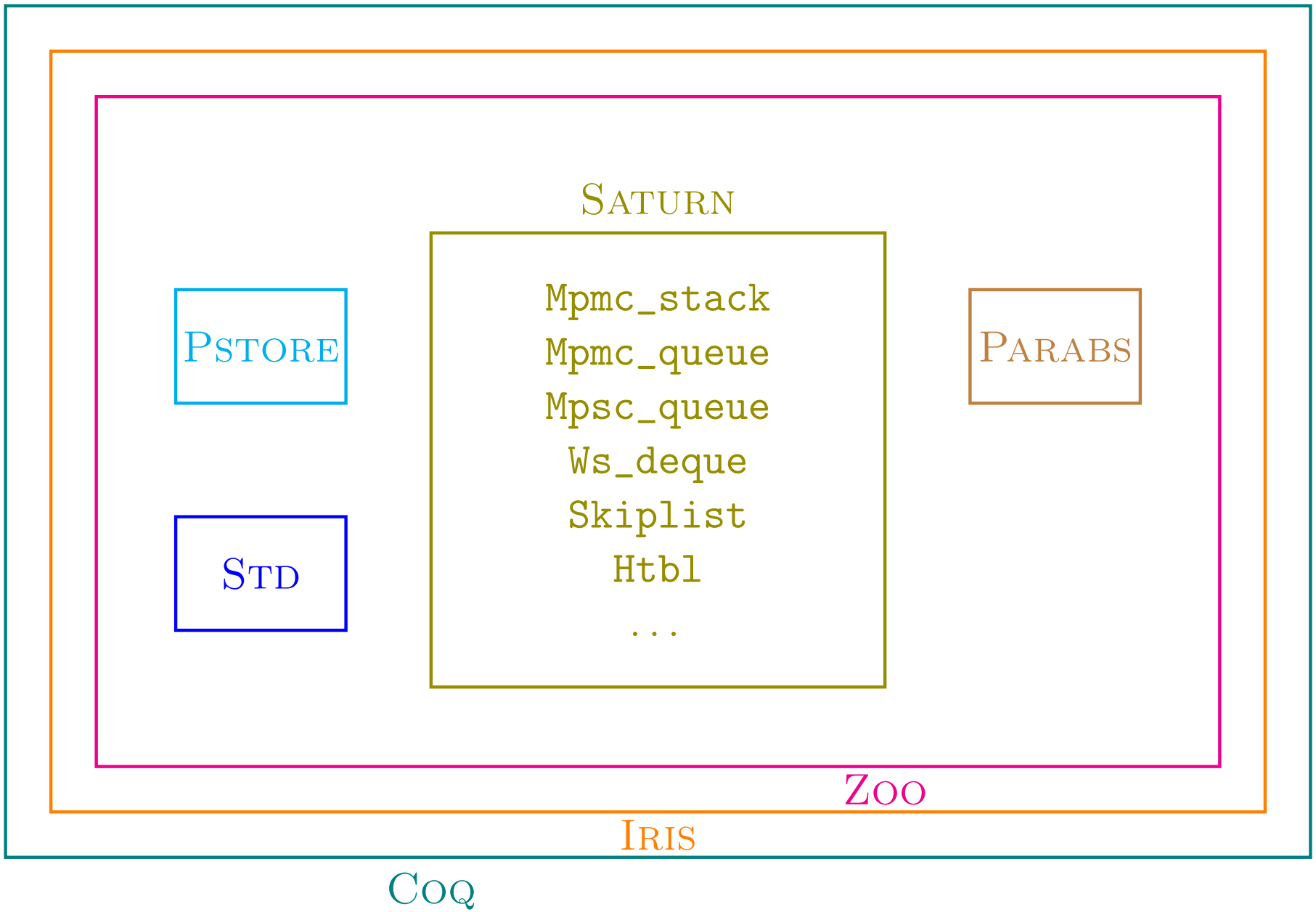
The SATURN team!

ZOO

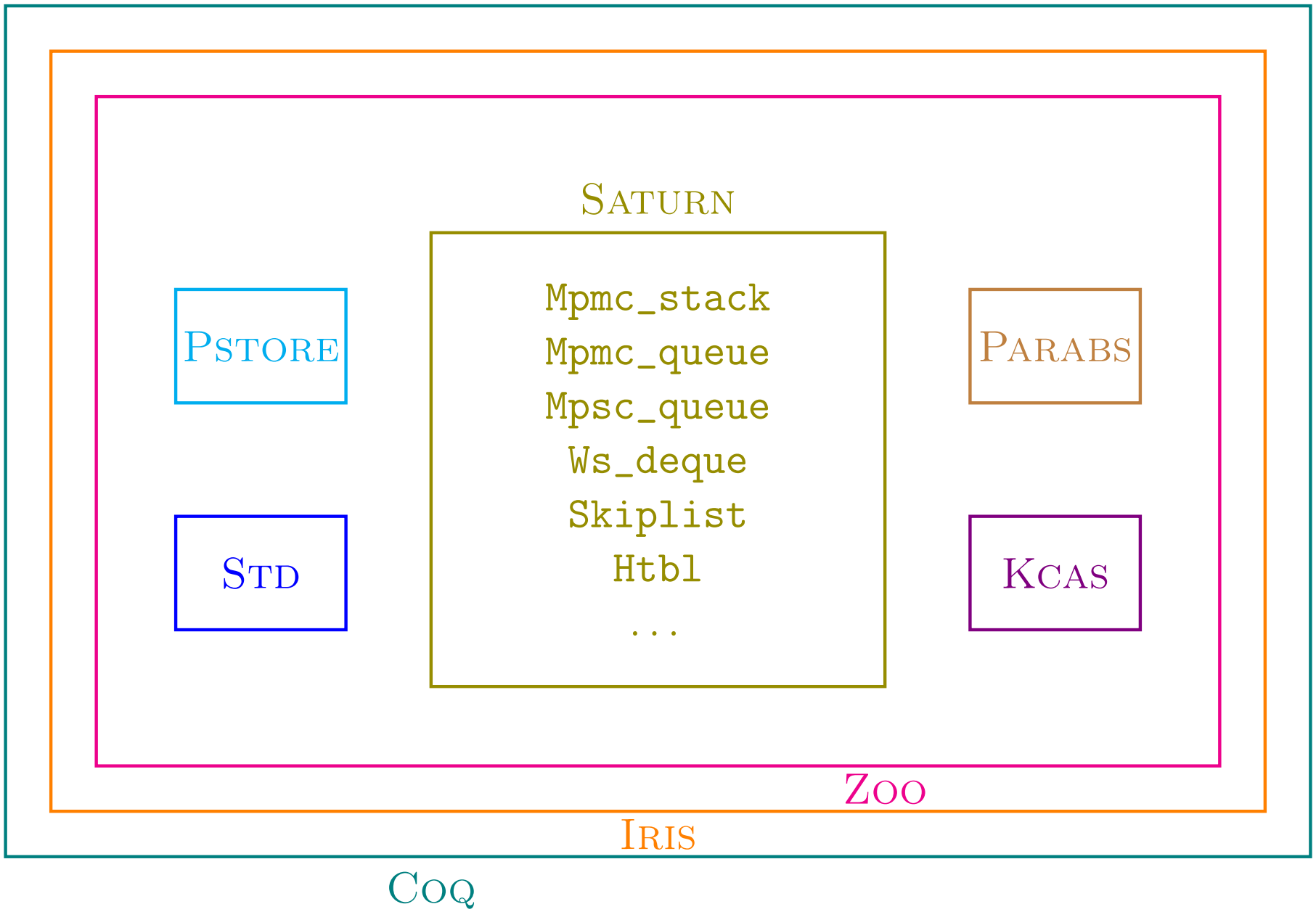
IRIS

COQ

ZOOlogy: what is it?



Zoology: what is it?



Zoology: what is it?

Vesa Karvonen

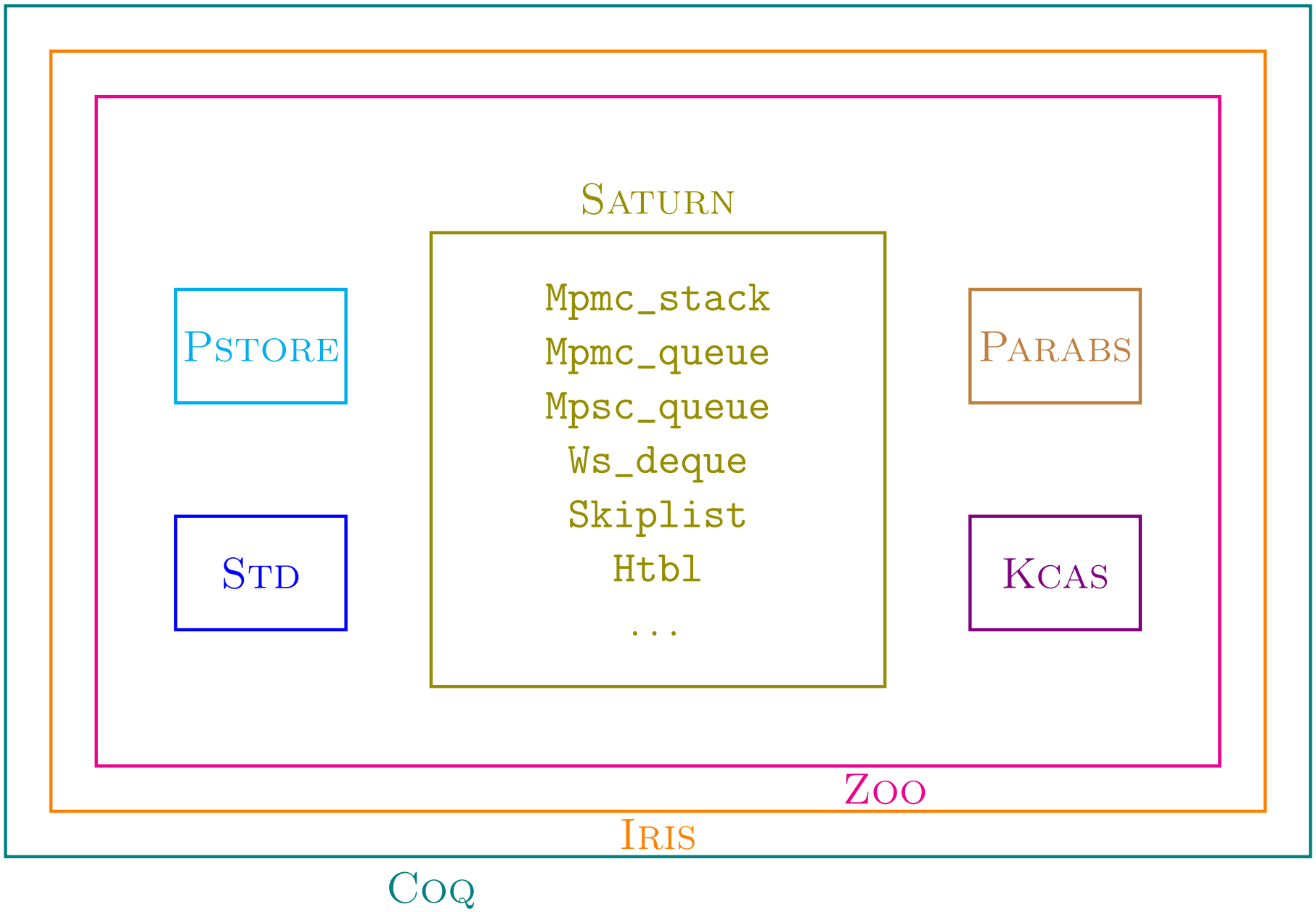
The main author of Kcas!

Zoo

Iris

Coq

Zoology: what is it?



ZOOlogy: why is it fun?

Lockfree algorithms typically exhibit complex behaviors:

- physical state $\neq$ logical state,
- external linearization points,
- future-dependent linearization points.

IRIS is a good match for verifying them thanks to advanced mechanisms:

- invariants to enforce protocols,
- atomic updates to materialize linearization points,
- prophecy variables to reason about the future.

Zoology

Specimen ① : Michael-Scott queue

Specimen ② : Kcas

Original algorithm (1996)

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*

Maged M. Michael Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{michael,scott}@cs.rochester.edu

Abstract

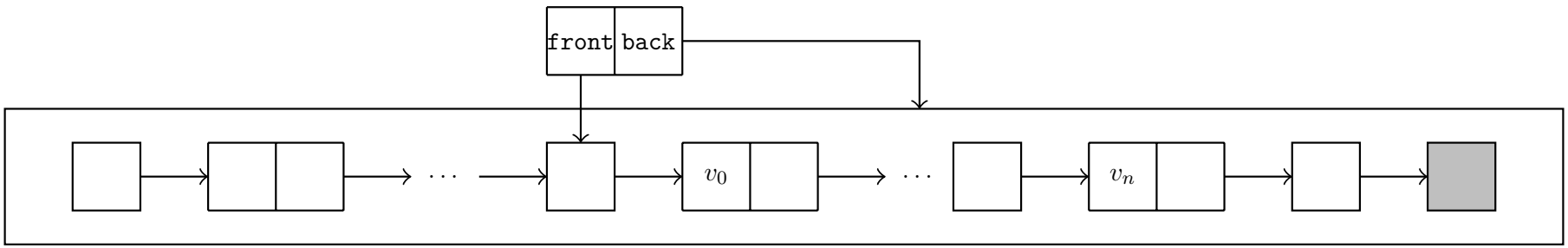
Drawing ideas from previous authors, we present a new non-blocking concurrent queue algorithm and a new two-lock queue algorithm in which one enqueue and one dequeue can proceed concurrently. Both algorithms are simple, fast, and practical; we were surprised not to find them in the literature. Experiments on a 12-node SGI Challenge multiprocessor indicate that the new non-blocking queue consistently outperforms the best known alternatives; it is the clear algorithm of choice for machines that provide a universal atomic primitive (e.g. `compare_and_swap` or `load_linked/store_conditional`). The two-lock concurrent queue outperforms a single lock when several processes are competing simultaneously for access; it appears to be the algorithm of choice for busy queues on machines with non-universal atomic primitives (e.g. `test_and_set`). Since much of the motivation for non-blocking algorithms is rooted in their immunity to large, unpredictable delays in process execution, we report experimental results both for systems with dedicated processors and for

1 Introduction

Concurrent FIFO queues are widely used in parallel applications and operating systems. To ensure correctness, concurrent access to shared queues has to be synchronized. Generally, algorithms for concurrent data structures, including FIFO queues, fall into two categories: *blocking* and *non-blocking*. Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. Non-blocking algorithms guarantee that if there are one or more active processes trying to perform operations on a shared data structure, some operation will complete within a finite number of time steps. On asynchronous (especially multiprogrammed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Possible sources of delay include processor scheduling preemption, page faults, and cache misses. Non-blocking algorithms are more robust in the face of these events.

Many researchers have proposed lock-free algorithms for concurrent FIFO queues. Hwang and Briggs [7],

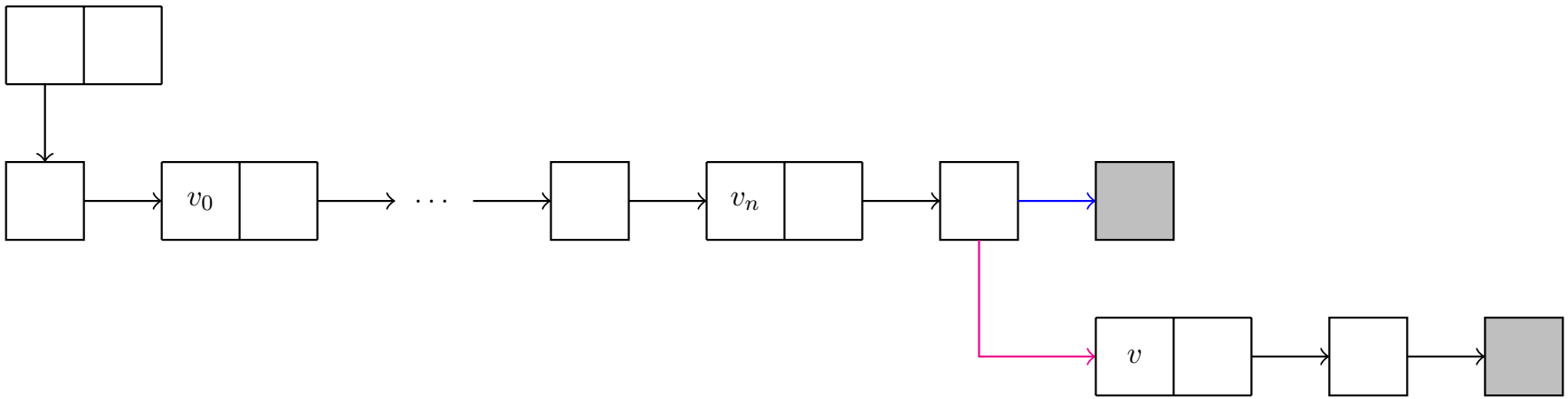
Basic implementation



singly-linked list

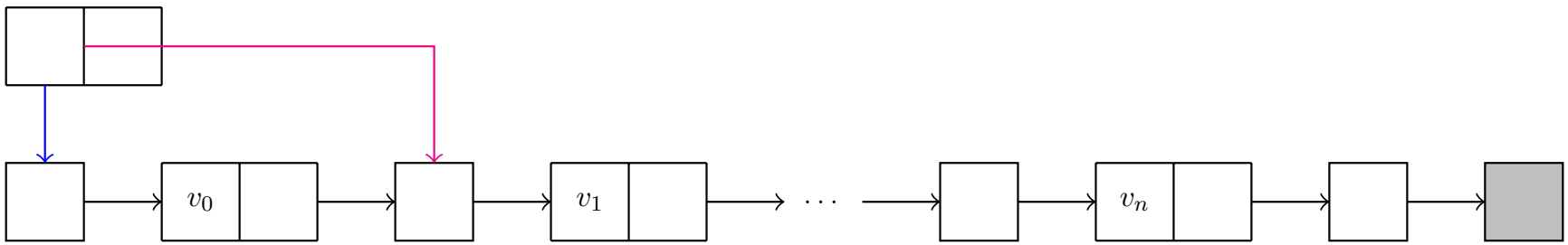
- The queue contains $[v_0, \cdots, v_n]$.
- `front` points to the first component of the active part of the list.
- `back` points somewhere in the list, usually closer to the end.

`push` linearization



$$\text{queue-model } t\ [v_0, \cdots, v_n]$$
$$\Downarrow$$
$$\text{queue-model } t\ [v_0, \cdots, v_n, v]$$

pop linearization



$$\text{queue-model } t\ [v_0, \cdots, v_n]$$
$$\Downarrow$$
$$\text{queue-model } t\ [v_1, \cdots, v_n]$$

Basic OCaml 5 implementation

```
type 'a node =
  | Nil
  | Next of 'a * 'a node Atomic.t

type 'a t = {
  front: 'a node Atomic.t Atomic.t ;
  back: 'a node Atomic.t Atomic.t ;
}
```

Specification

$$\frac{\{\,\text{queue-inv}\ t\ \iota\,\}}{\langle\,\forall\, vs.\,\text{queue-model}\ t\ vs\,\rangle}\;\texttt{queue\_push}\ t\ v,\ \uparrow\iota\;\frac{\langle\,\text{queue-model}\ t\ (vs +\!\!+ [v])\,\rangle}{\{\,()\,.\,\text{True}\,\}}$$

$$\frac{\{\,\text{queue-inv}\ t\ \iota\,\}}{\langle\,\forall\, vs.\,\text{queue-model}\ t\ vs\,\rangle}\;\texttt{queue\_pop}\ t,\ \uparrow\iota\;\frac{\langle\,\text{queue-model}\ t\ (\text{tail}\ vs)\,\rangle}{\{\,\text{head}\ vs.\,\text{True}\,\}}$$

Contextual Refinement of the Michael-Scott Queue (Proof Pearl)

Simon Friis Vindum
vindum@cs.au.dk
Aarhus University
Aarhus, Denmark

Lars Birkedal
birkedal@cs.au.dk
Aarhus University
Aarhus, Denmark

Abstract

The Michael-Scott queue (MS-queue) is a concurrent non-blocking queue. In an earlier pen-and-paper proof it was shown that a simplified variant of the MS-queue contextually refines a coarse-grained queue. Here we use the Iris and ReLoC logics to show, for the first time, that the original MS-queue contextually refines a coarse-grained queue. We make crucial use of the recently introduced prophecy variables of Iris and ReLoC. Our proof uses a fairly simple invariant that relies on encoding which nodes in the MS-queue can reach other nodes. To further simplify the proof, we extend separation logic with a generally applicable *persistent points-to predicate* for representing immutable pointers. This relies on a generalization of the well-known algebra of fractional permissions into one of *discardable* fractional permissions. We define the persistent points-to predicate entirely inside the base logic of Iris (thus getting soundness "for free").

We use the same approach to prove refinement for a variant of the MS-queue resembling the one used in the `java.util.concurrent` library.

We have mechanized our proofs in Coq using the formalizations of ReLoC and Iris in Coq.

CCS Concepts: • **Theory of computation → Separation logic**; **Concurrent algorithms**.

Keywords: separation logic, concurrency, Iris, Coq

1 Introduction

The Michael-Scott queue (MS-queue) is a fast and practical fine-grained concurrent queue [14]. We prove that the MS-queue is a *contextual refinement* of a coarse-grained concurrent queue. The coarse-grained queue, shown in Figure 1, is implemented as a reference to a functional list and uses a lock to sequentialize concurrent accesses to the queue. We thus prove that in any program we may replace uses of the coarse-grained, but obviously correct, concurrent queue with the faster, but more intricate, MS-queue, without changing the observable behaviour of the program. We recall that, formally, an expression e contextually refines another expression e', denoted $\Delta; \Gamma \vdash e \precsim_{ctx} e' : \tau$, if for all contexts K, of ground type, whenever $K[e]$ terminates with a value there exists an execution of $K[e']$ that terminates with the same value. One should think of e as the *implementation* (in our case the MS-queue), e' as the *specification* (in our case the coarse-grained queue), and K as a *client* of a queue implementation.

Note that the contextual refinement implies that the internal states of the two queues are *encapsulated* and hidden from clients who could otherwise tell the difference between the two implementations. Contextual refinement is also related to *linearizability*, a popular correctness criterion considered for concurrent data structures. Linearizability has

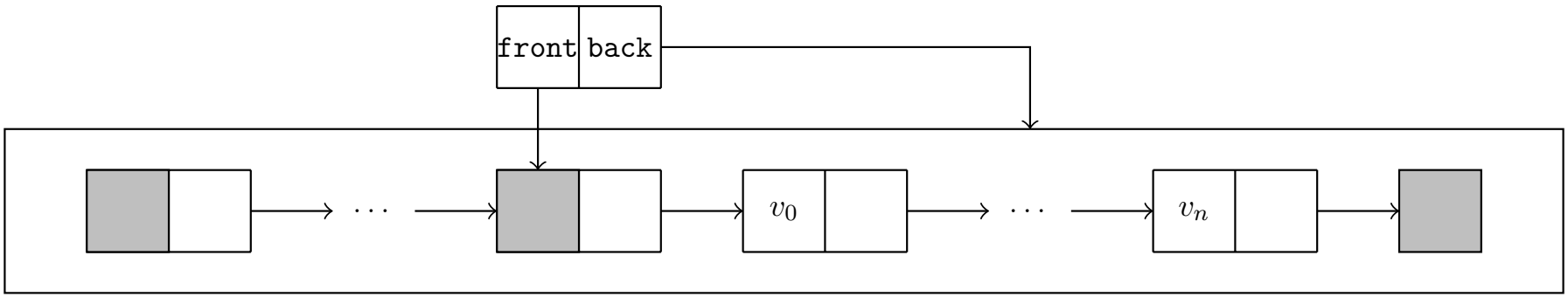
Verification by Vindum & Birkedal (2021)

Contextual Refinement of the Michael-Scott Queue (Proof Pearl)

A key insight of our approach is how the invariants that the MS-queue maintains can be expressed in terms of which nodes are *reachable* from other nodes. Reachability is expressed with an inductive predicate:

$$\ell_n \rightsquigarrow \ell_m \triangleq \exists \ell_{n\to}, v.\, \ell_n \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}(v, \ell_{n\to}) * (\ell_n = \ell_m \vee \exists \ell_p.\, \ell_{n\to} \hookrightarrow_{\mathsf{i}}^{\square} \ell_p * \ell_p \rightsquigarrow \ell_m)$$

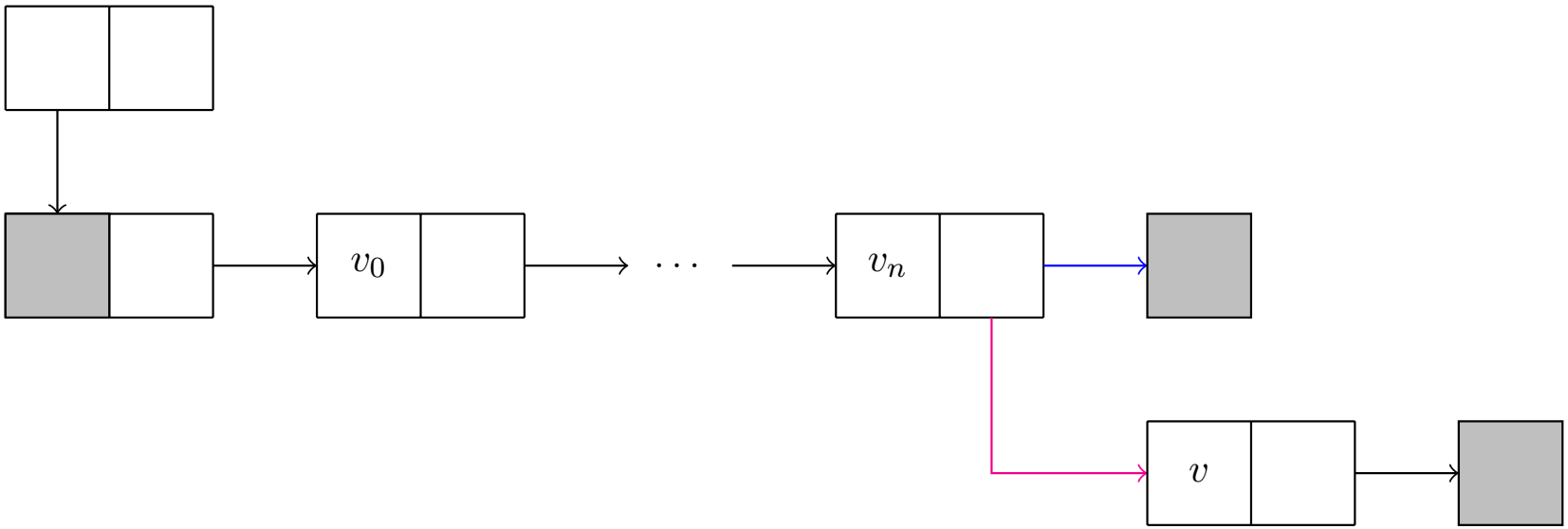
Optimized implementation



singly-linked list

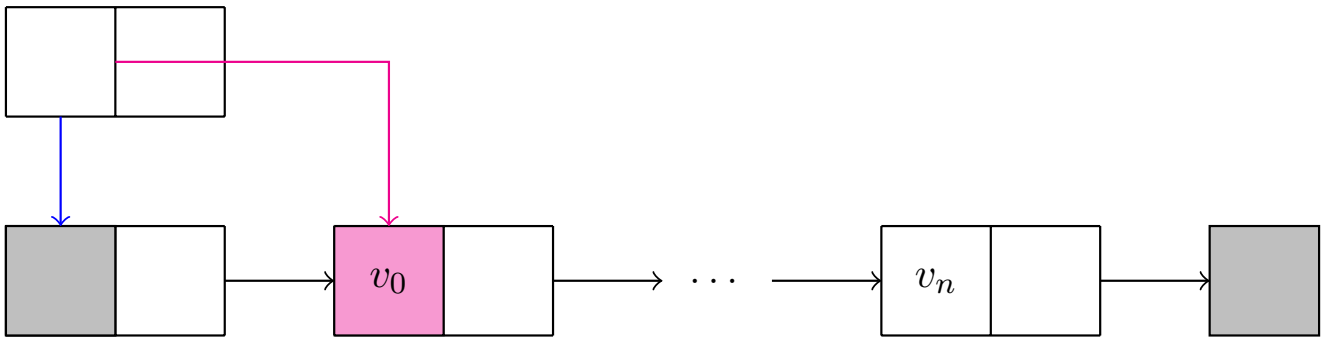
- The queue contains $[v_0, \cdots, v_n]$.
- `front` points to the first component of the active part of the list.
- `back` points somewhere in the list, usually closer to the end.

`push` linearization



$$\text{queue-model } t\ [v_0, \cdots, v_n]$$
$$\Downarrow$$
$$\text{queue-model } t\ [v_0, \cdots, v_n, v]$$

pop linearization

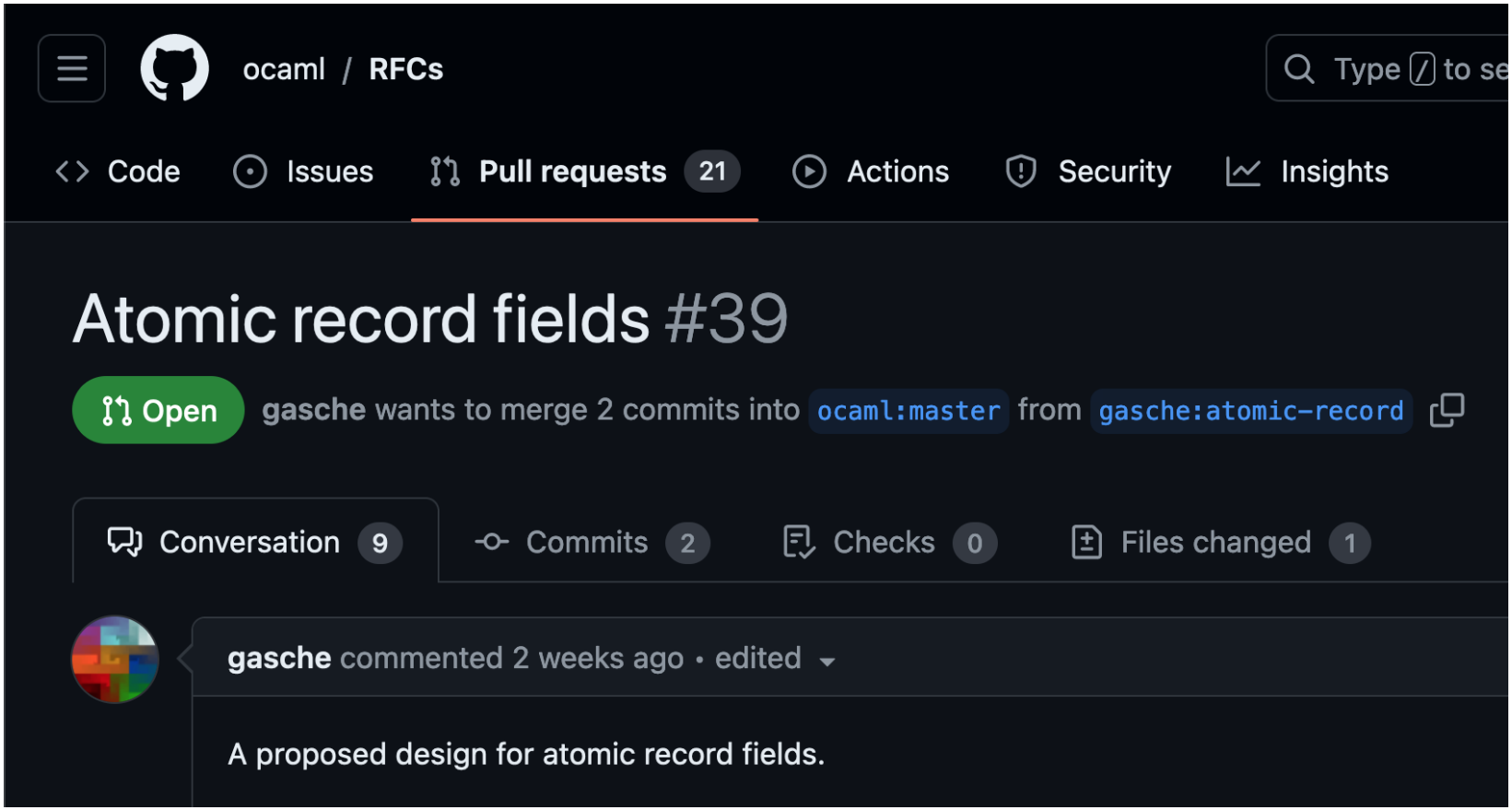


$$\text{queue-model } t\ [v_0, \cdots, v_n]$$
$$\Downarrow$$
$$\text{queue-model } t\ [v_1, \cdots, v_n]$$

Optimized OCaml 5 implementation — first try

```
type ('a, _) node =
  | Nil : ('a, [> `Nil]) node
  | Next : {
      mutable next: ('a, [`Nil | `Next]) node ;
      mutable value: 'a ;
    } ->
    ('a, [> `Next]) node

external node_as_atomic :
  ('a, [`Next]) node ->
  ('a, [`Nil | `Next]) node Atomic.t
= "%identity"

let push t v =
  ...
  if Atomic.compare_and_set (node_as_atomic back) Nil node
  then ...
  else ...
```

A new hope



(no modification of the OCaml 5 memory model)

Optimized OCaml 5 implementation — second try

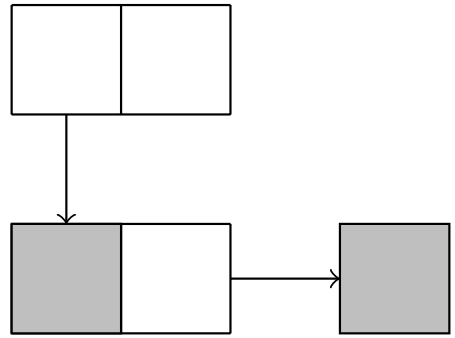
```
type ('a, _) node =
  | Nil : ('a, [> `Nil]) node
  | Next of {
      mutable next: ('a, [`Nil | `Next]) node [@atomic] ;
      mutable value: 'a ;
    } ->
    ('a, [> `Next]) node

let push t v =
  ...
  if Atomic.Loc.compare_and_set
     [%atomic.field back.next]
     Nil node
  then ...
  else ...
```

What about `is_empty`?

```
let is_empty t =
  let Next front = t.front in
  front.next == Nil
```

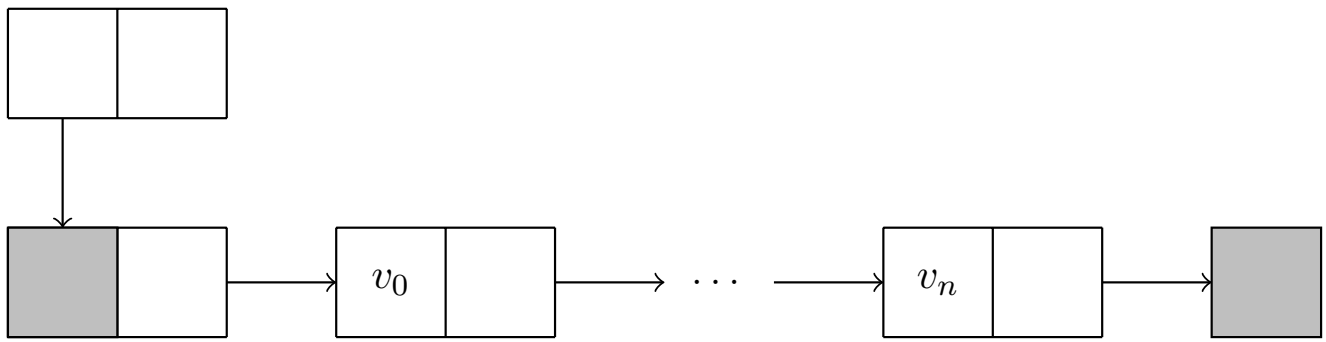
$$\frac{\frac{\frac{\frac{\{\,\text{queue-inv}\ t\ \iota\,\}}{\langle\,\forall\, vs.\,\text{queue-model}\ t\ vs\,\rangle}}{\texttt{queue\_is\_empty}\ t,\ \uparrow \iota}}{\langle\,\text{queue-model}\ t\ vs\,\rangle}}{\left\{\, vs \stackrel{?}{=} [\,].\,\text{True}\,\right\}}$$

`is_empty` (potentially external) linearization



$$[v_0, \cdots, v_n] = []$$

$$[v_0, \cdots, v_n] \neq []$$



Iris invariant

```
Definition queue_inv l γ ι : iProp Σ :=
  ∃ hist past front nodes back vs waiters,

  ⌜hist = past ++ front :: nodes⌝ ∗
  ⌜back ∈ hist⌝ ∗

  l.[front] ↦ #front ∗
  l.[back] ↦ #back ∗

  queue_front_auth γ (length past) ∗

  queue_history_auth γ hist ∗
  xchain_model hist () ∗

  queue_model₂ γ vs ∗
  ([∗ list] node; v ∈ nodes; vs, node.[xchain_data] ↦ v) ∗

  queue_waiters_auth γ waiters ∗
  ([∗ map] waiter ↦ i ∈ waiters, queue_waiter γ ι past waiter i).
```

What more?

- Multi-producer *single-consumer* queue.
- Multi-producer multi-consumer *bounded* queue.
- More advanced queues derived from Michael-Scott?
- Space complexity by Alexandre Moine.

Zoology

Specimen ① : Michael-Scott queue

Specimen ② : Kcas

Harris, Fraser & Pratt algorithm (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

`CASN` is an operation for shared-memory systems that reads the contents of a series of locations, compares these against specified values and, if they all match, updates the locations with a further set of values. All this is performed atomically

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 sometimes strictly forward reasoning is not good enough: when reasoning about a program step s_0, it may be necessary to "peek into the future" and know ahead of time what will happen at some

Guerraoui, Kogan, Marathe & Zablotchi algorithm

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.

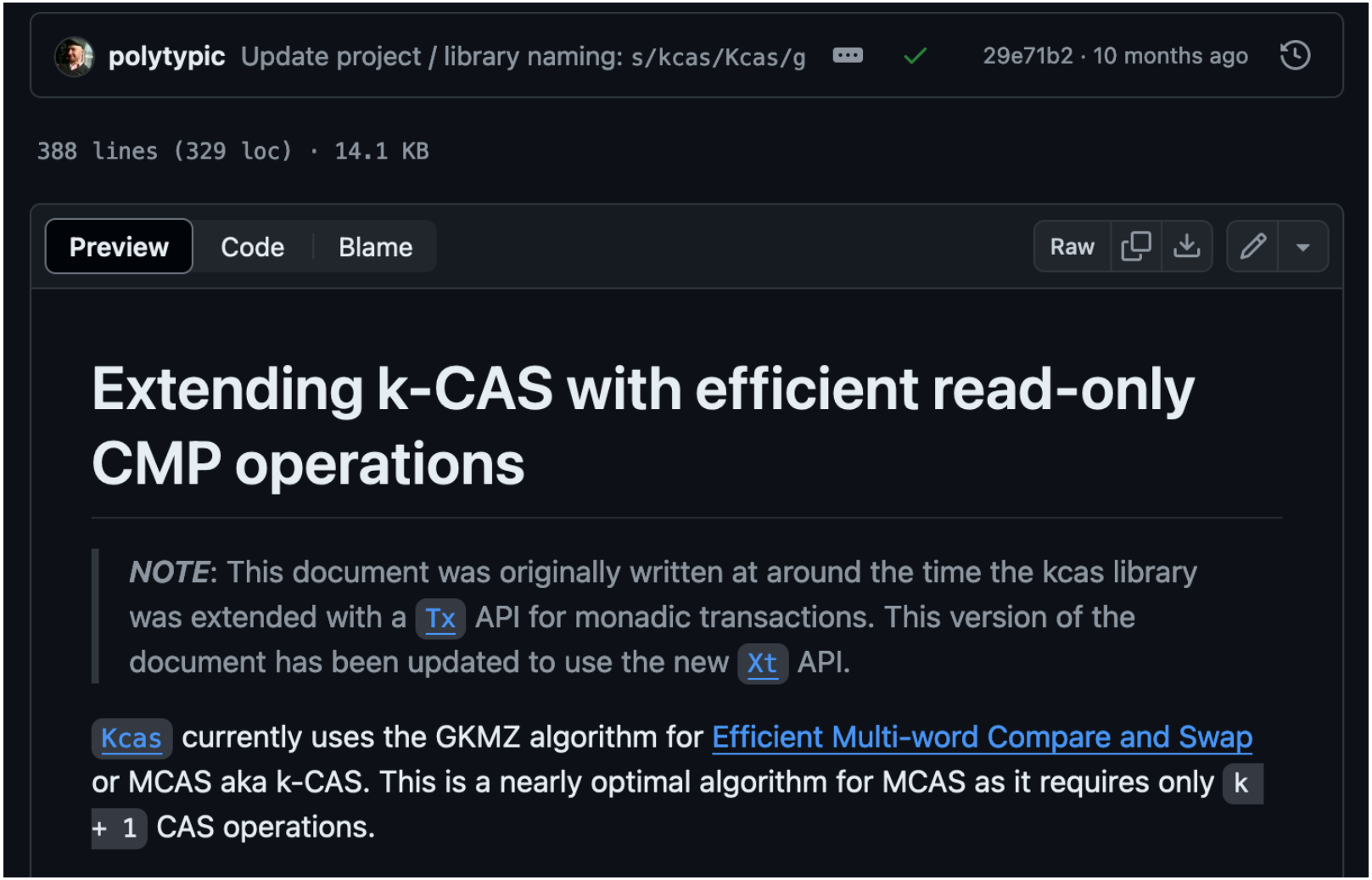
We first prove in this paper that it is impossible to "pack" the information required to perform a k-word CAS (k-CAS) in less than k locations to be CASed. Then we present the first algorithm that requires $k + 1$ CASes per call to k-CAS in the common uncontended case. We implement our algorithm and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

Keywords and phrases lock-free, multi-word compare-and-swap, persistent memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.4

Extension by Vesa Karvonen



Thank you for your attention!