ZOOlogy of lockfree concurrent data structures

Clément Allain

INRIA Paris

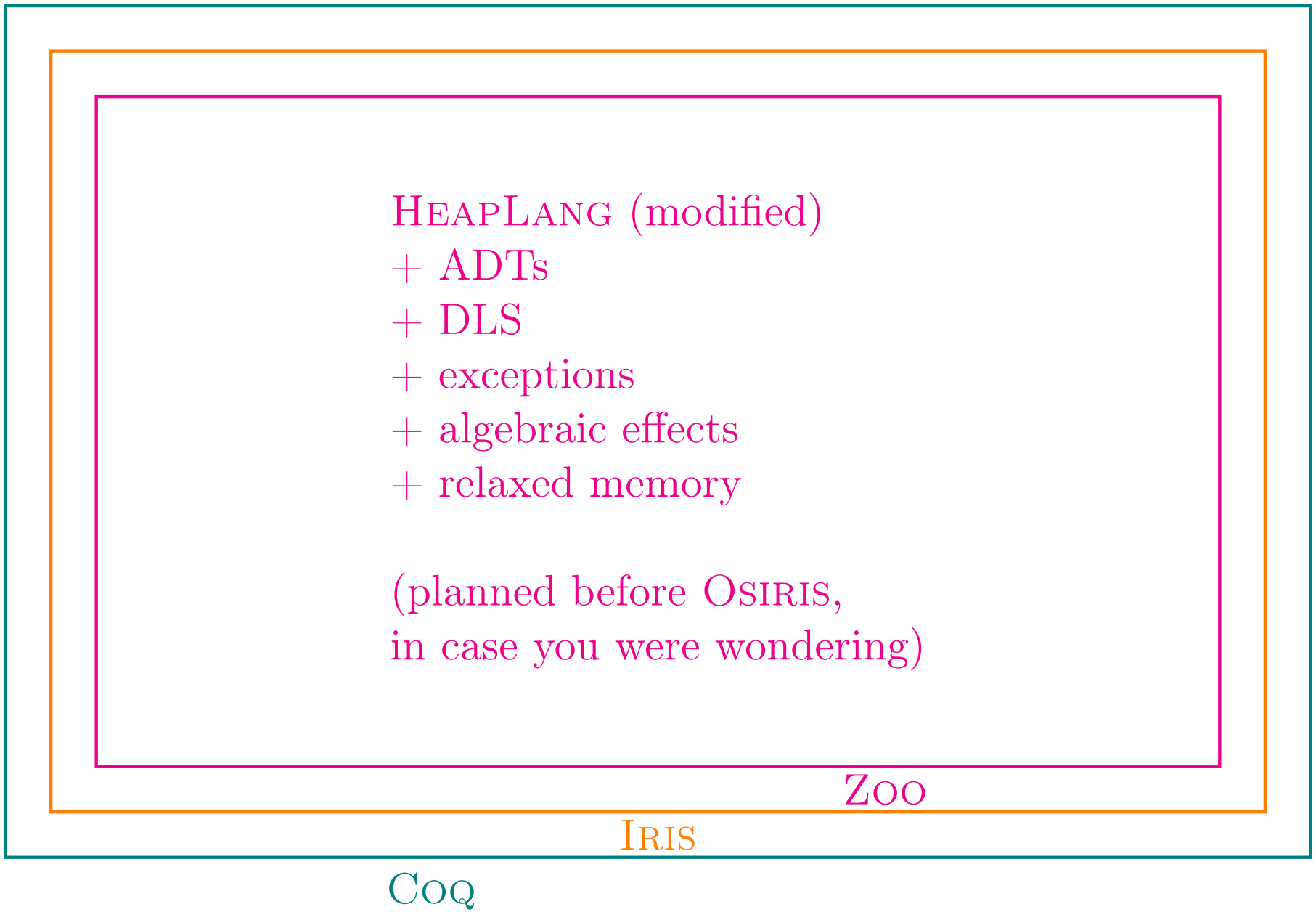
June 17, 2024

Zoology

Specimen ① : Michael-Scott queue

Specimen ② : Kcas

ZOOlogy: what is it?



ZOOlogy: what is it?

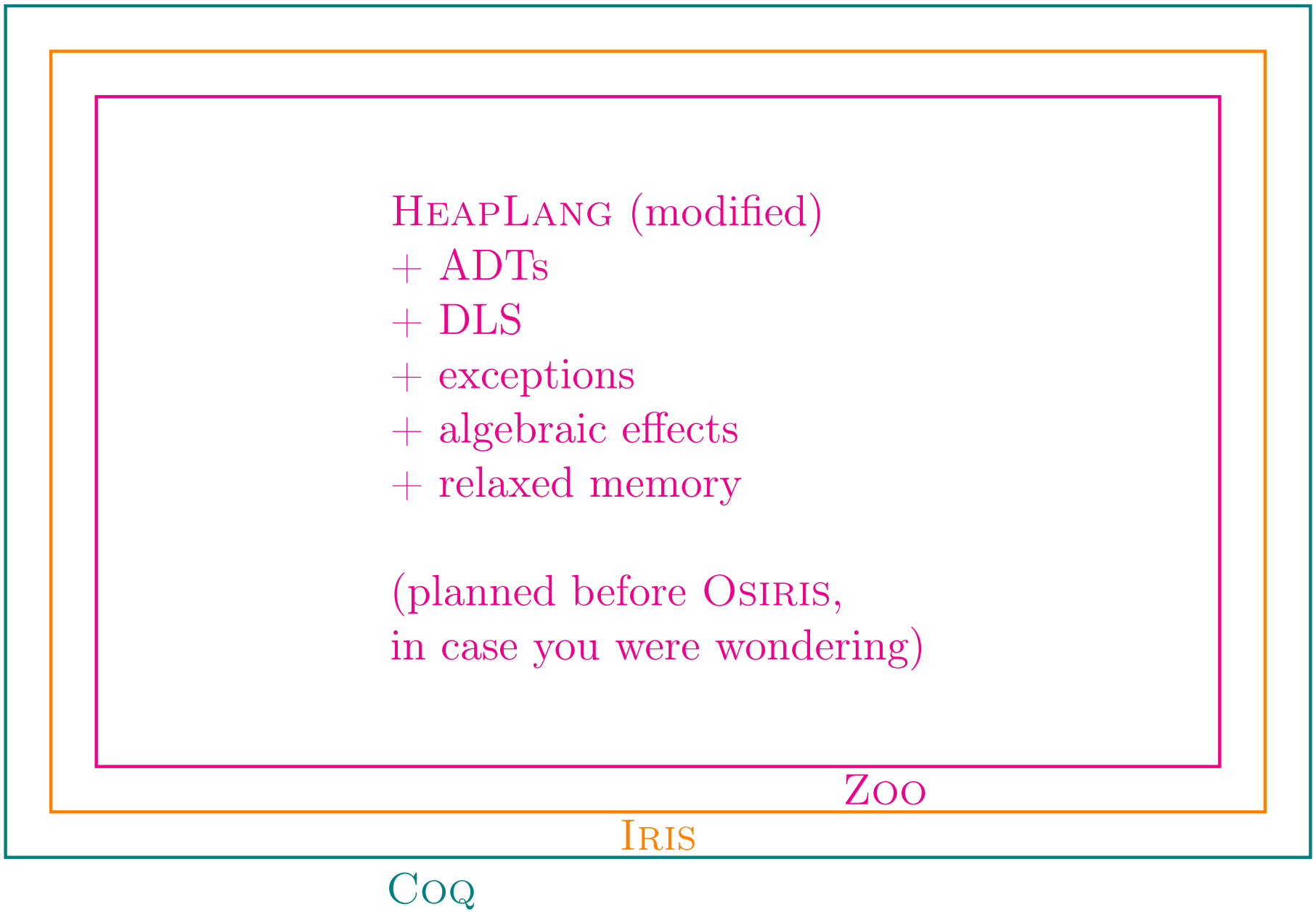
```
MetaCoq Run (zoo_variant "list" [
  "Nil" ;
  "Cons"
]).

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.
```

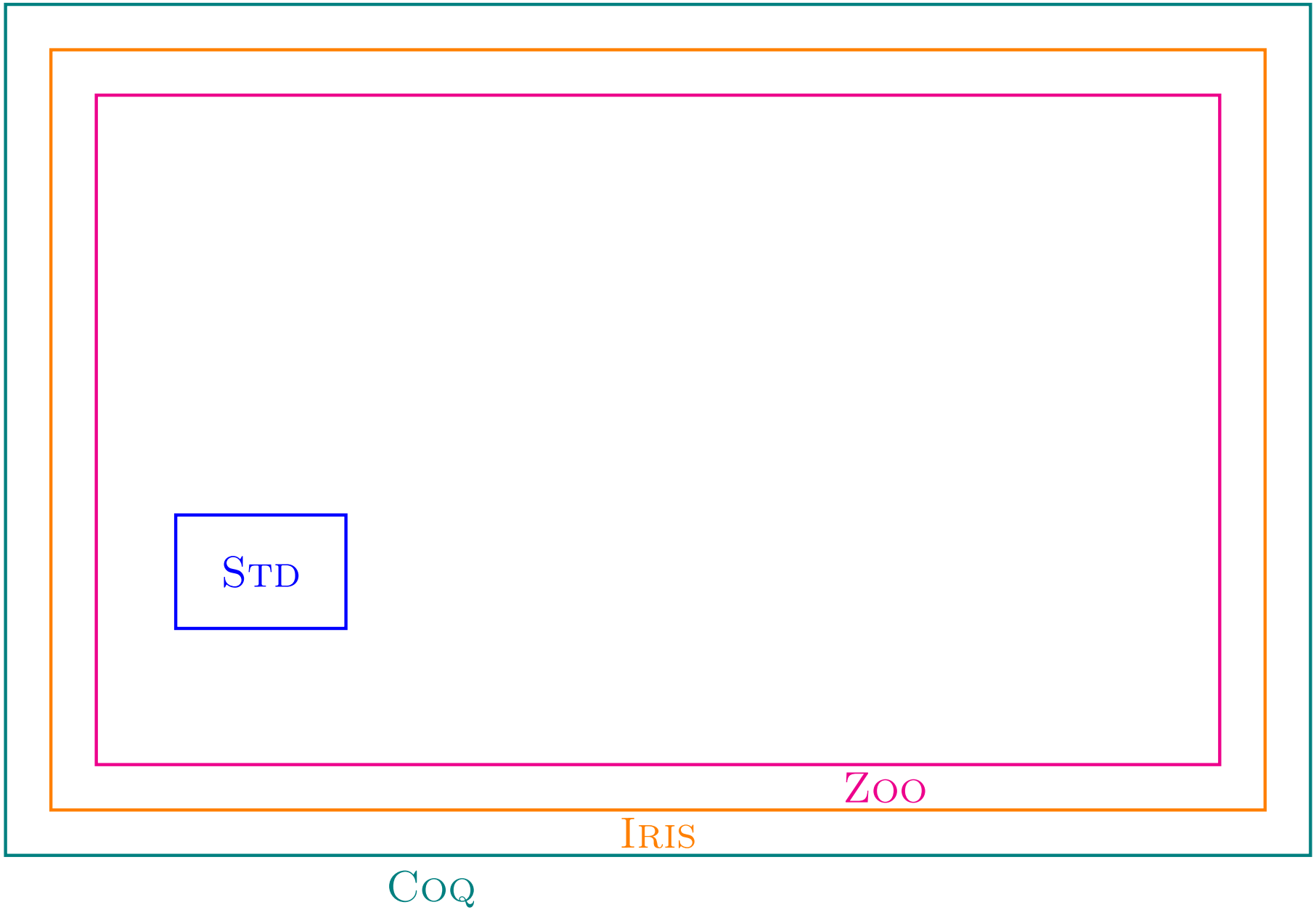
IRIS

COQ

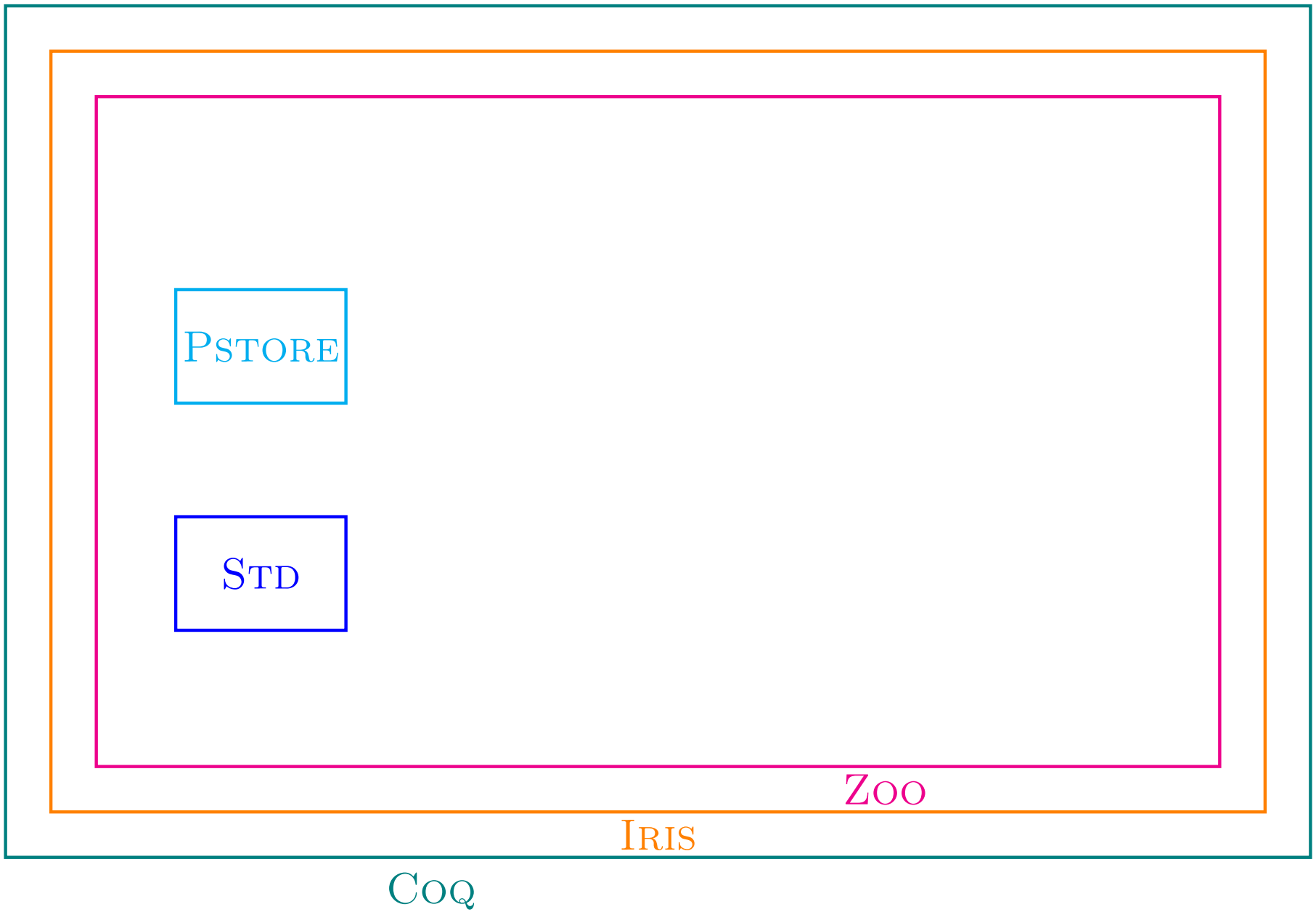
ZOOlogy: what is it?



ZOOlogy: what is it?

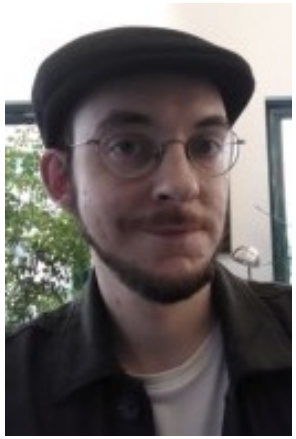


ZOOlogy: what is it?



Zoology: what is it?

Clément Allain



Basile Clément

Alexandre Moine

Gabriel Scherer

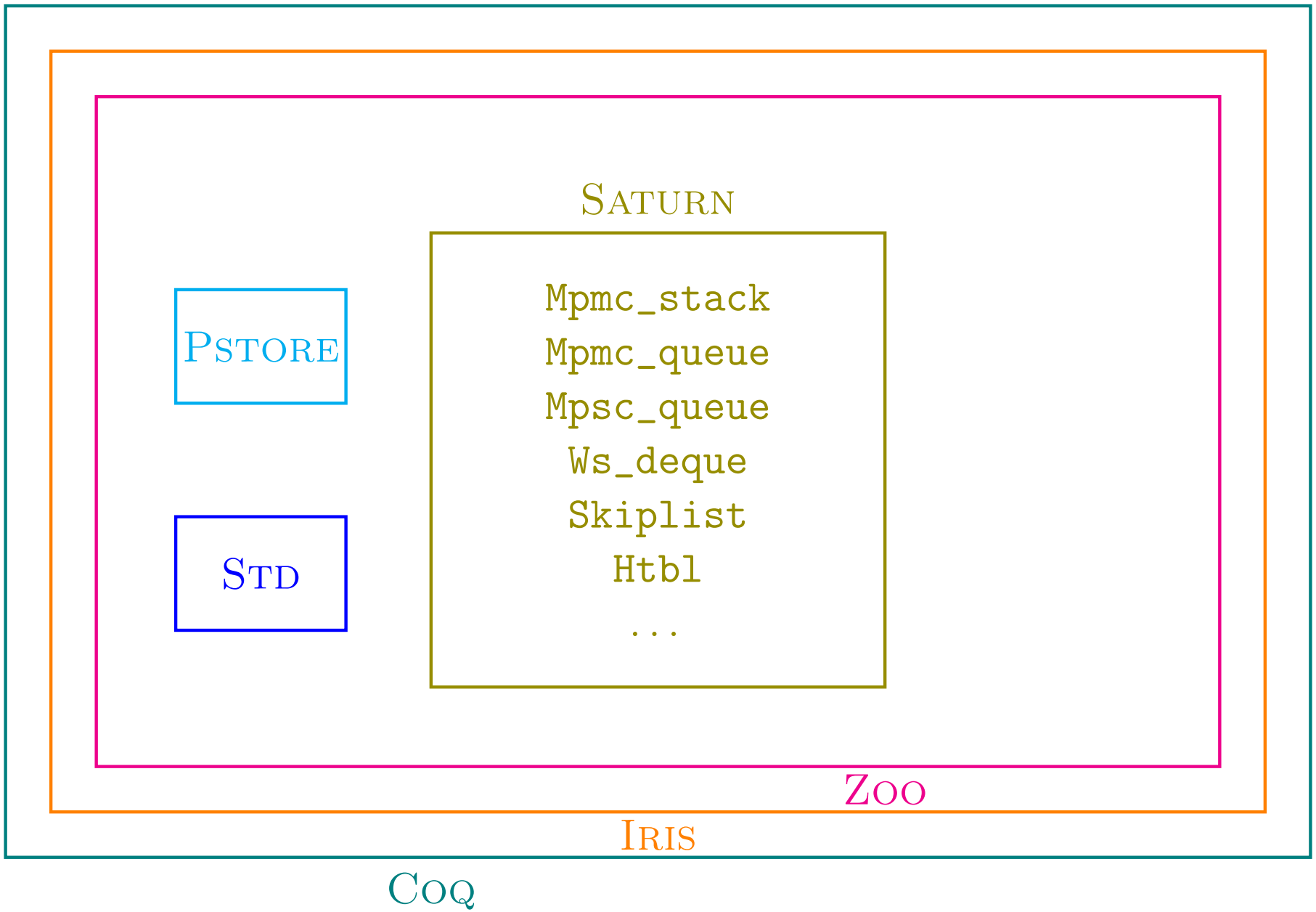
The Pstore team!

Zoo

Iris

Coq

ZOOlogy: what is it?



ZOOlogy: what is it?

Vesa Karvonen

Carine Morel

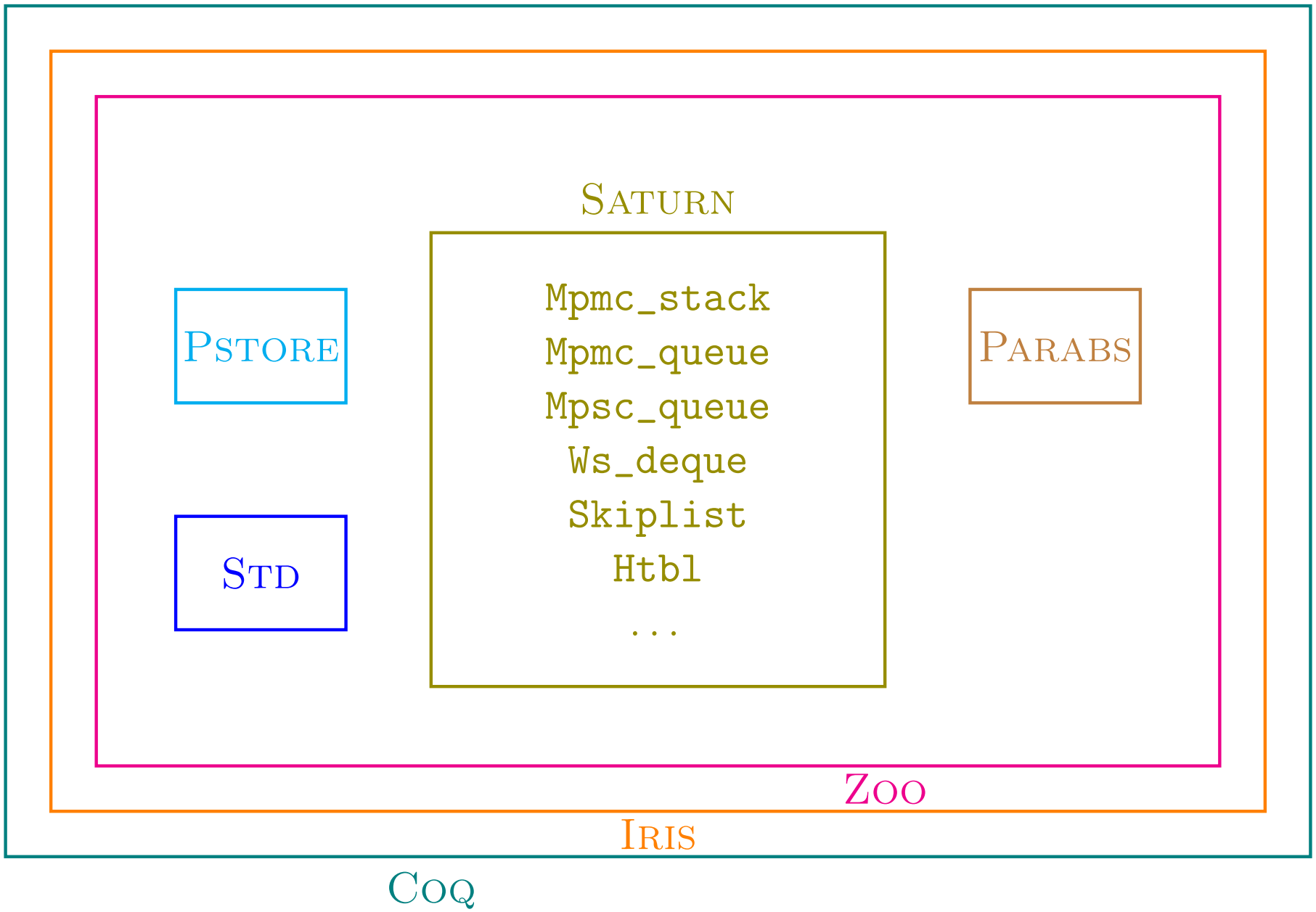
The SATURN team!

ZOO

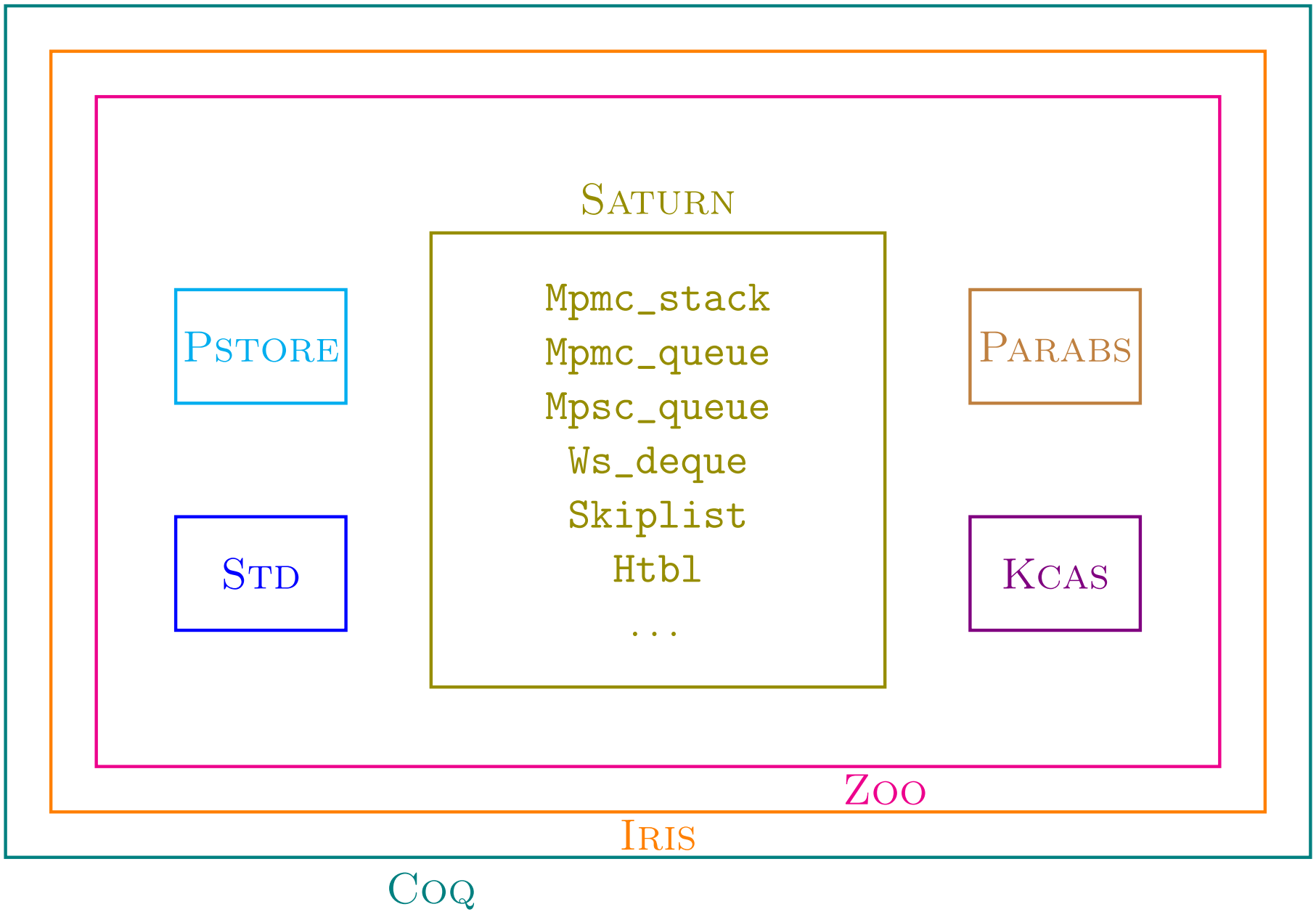
IRIS

COQ

ZOOlogy: what is it?



ZOOlogy: what is it?



ZOOlogy: what is it?

Vesa Karvonen

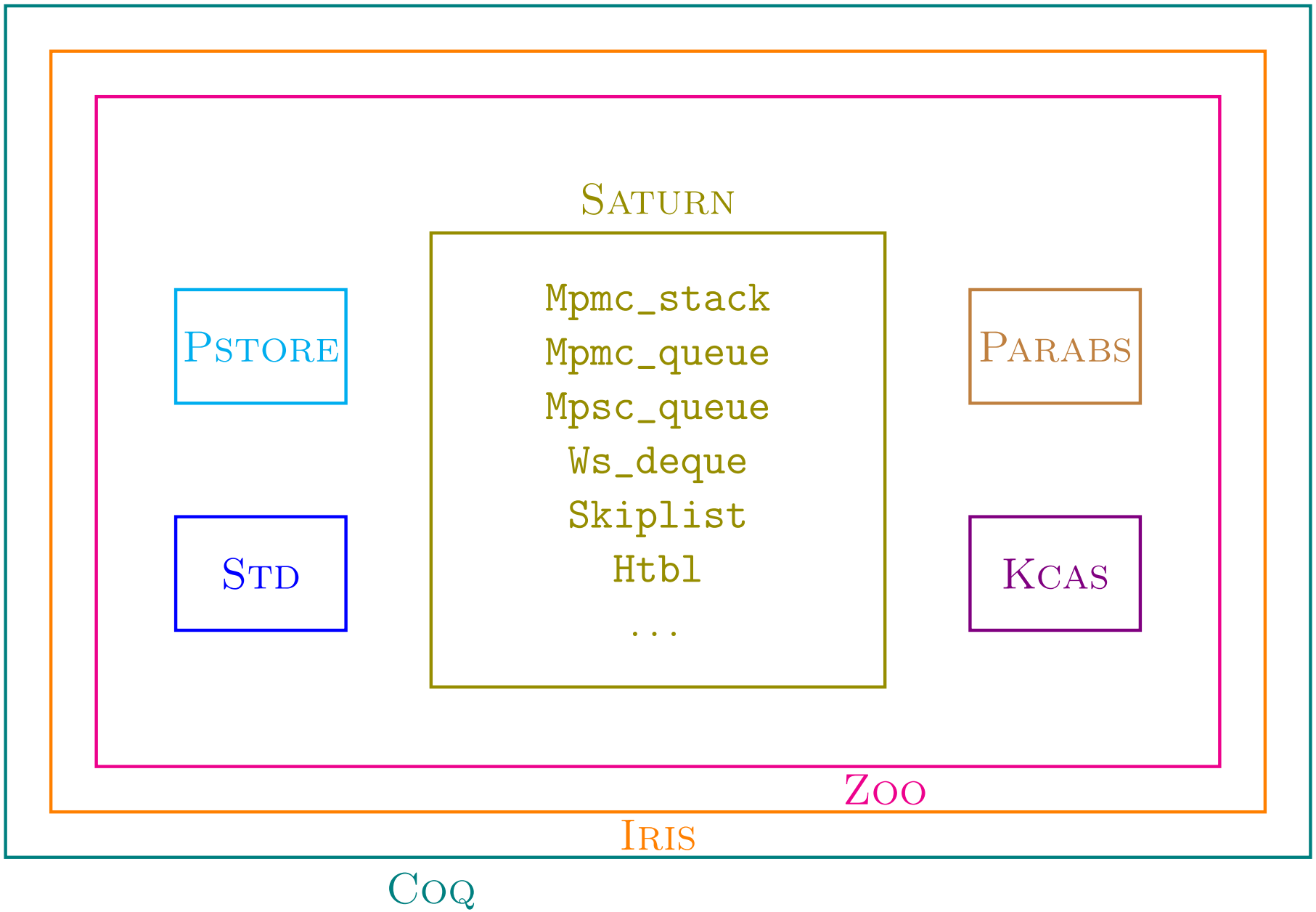
The main author of KCAS!

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 article (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

TODO

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

Optimized implementation

TODO

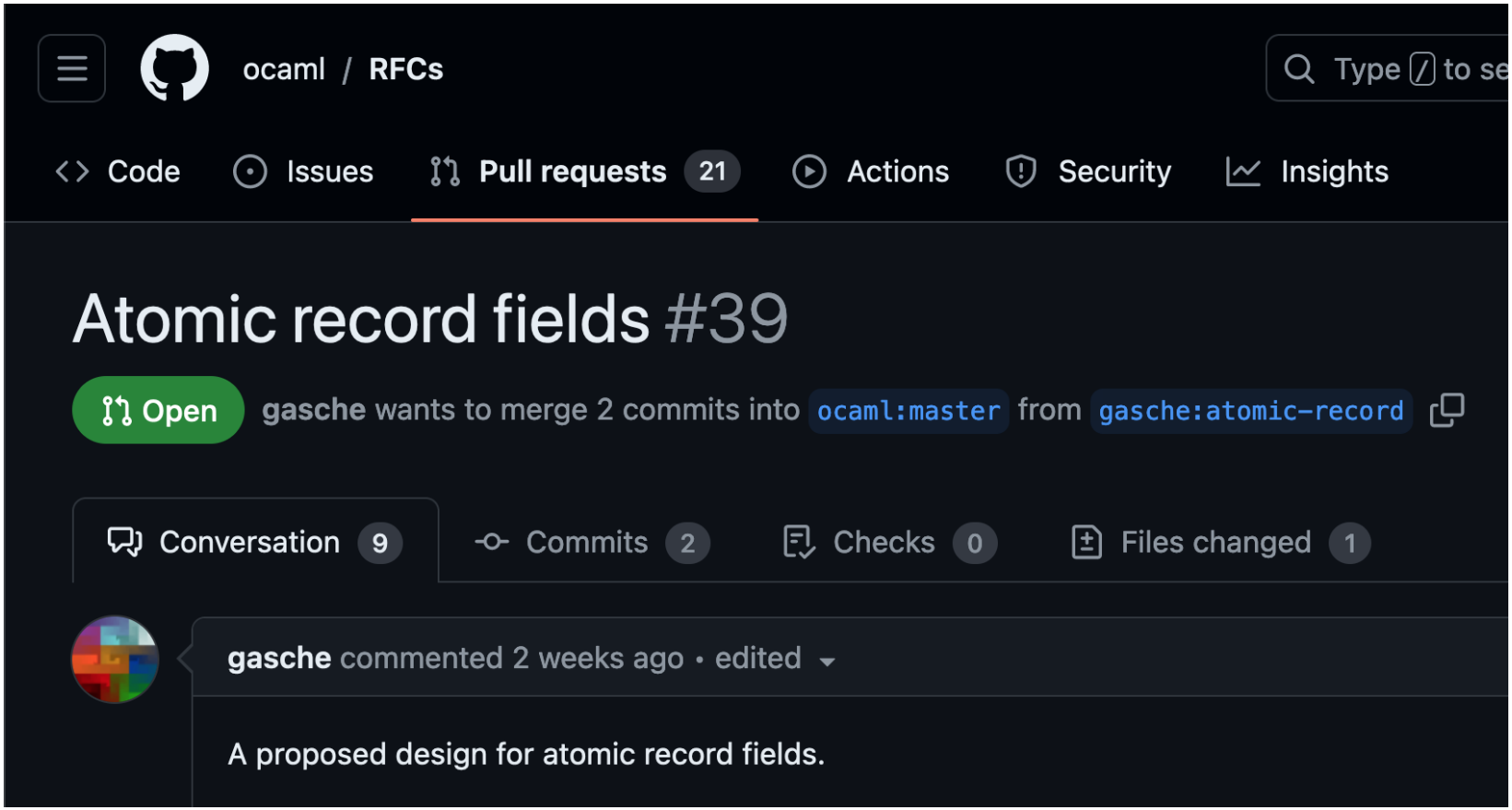
Optimized OCaml 5 implementation — first try

```
type ('a, _) t =
  | Nil : ('a, [> `Nil]) t
  | Next : {
      mutable next: ('a, [`Nil | `Next]) t ;
      mutable value: 'a ;
    } ->
    ('a, [> `Next]) t

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

let push t v =
  ...
  if Atomic.compare_and_set (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 t =
  | Nil
  | Next of {
      mutable next: 'a t [@atomic] ;
      mutable value: 'a ;
    }

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

What about `is_empty`?

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

TODO

Iris Invariant

TODO

Space complexity

Diamonds Are Forever: Reasoning about Heap Space in a Concurrent and Garbage Collected Language

Alexandre Moine Arthur Charguéraud François Pottier

Iris'23

Inria

Zoology

Specimen ① : Michael-Scott queue

Specimen ② : Kcas

Kcas

TODO

Thank you for your attention!