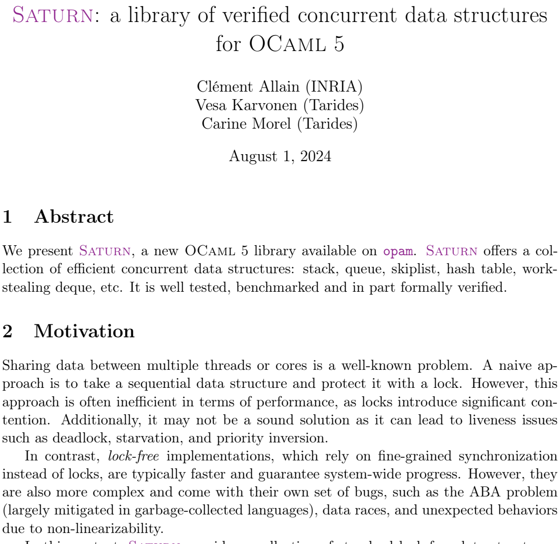
Snapshottable stores

Clément Allain (INRIA Paris)
Basile Clément (OCamlPro)
Alexandre Moine (INRIA Paris)
Gabriel Scherer (INRIA Paris)

December 12, 2024

Vérification d'algorithmes concurrents



Saturn: a library of verified concurrent data structures for OCaml 5

Clément Allain (INRIA)
Vesa Karvonen (Tarides)
Carine Morel (Tarides)

August 1, 2024

1 Abstract

We present Saturn, a new OCaml 5 library available on `opam`. Saturn offers a collection of efficient concurrent data structures: stack, queue, skiplist, hash table, work-stealing deque, etc. It is well tested, benchmarked and in part formally verified.

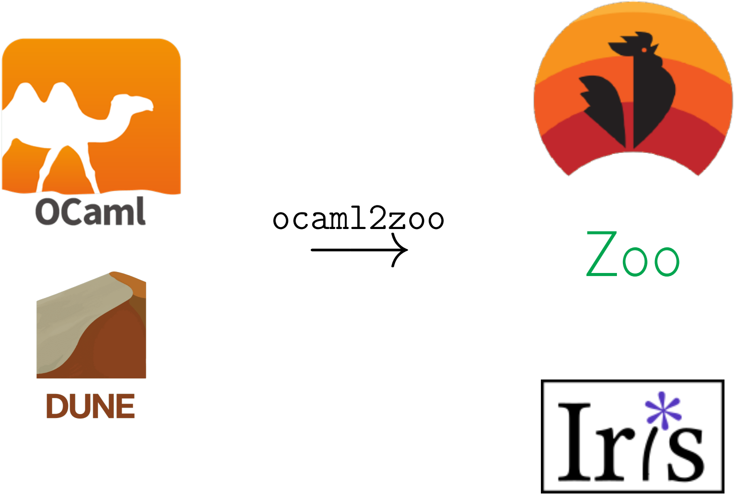
2 Motivation

Sharing data between multiple threads or cores is a well-known problem. A naive approach is to take a sequential data structure and protect it with a lock. However, this approach is often inefficient in terms of performance, as locks introduce significant contention. Additionally, it may not be a sound solution as it can lead to liveness issues such as deadlock, starvation, and priority inversion.

In contrast, *lock-free* implementations, which rely on fine-grained synchronization instead of locks, are typically faster and guarantee system-wide progress. However, they are also more complex and come with their own set of bugs, such as the ABA problem (largely mitigated in garbage-collected languages), data races, and unexpected behaviors due to non-linearizability.

In this context, Saturn provides a collection of standard lock-free data structures

OCaml vers Zoo



OCaml

DUNE

$\xrightarrow{\texttt{ocaml2zoo}}$

Zoo

Iris

Store : deux interfaces

```
type t
type store = t
val create : unit -> t
module Ref : sig
  type 'a t
  val make : store -> 'a -> 'a t
  val get : store -> 'a t -> 'a
  val set : store -> 'a t -> 'a -> unit
end
```

```
type snapshot
val capture :
  store -> snapshot
val restore :
  store -> snapshot -> unit
```

```
type transaction
val transaction :
  store -> transaction
val rollback :
  store -> transaction -> unit
val commit :
  store -> transaction -> unit
```

ocaml2zoo appliqué à Store

```
let restore t s =
  if t != s.snap_store then (
    assert false
  ) else (
    let root = s.snap_root in
    match !root with
    | Root ->
        ()
    | Diff _ ->
        reroot root ;
        t.gen <- s.snap_gen + 1 ;
        t.root <- root
  )
```

```
Definition pstore_restore : val :=
  fun: "t" "s" =>
    if: "t" != "s".<snap_store> then (
      Fail
    ) else (
      let: "root" := "s".<snap_root> in
      match: !"root" with
      | Root =>
          ()
      | Diff <> <> <> <> =>
          pstore_reroot "root" ;;
          "t" <-{gen} "s".<snap_gen> + #1 ;;
          "t" <-{root} "root"
      end
    ).
```

Spécification : un simple état mutable...

$$\begin{array}{c}\{\,\mathrm{True}\,\}\\ \hline \texttt{create ()}\\ \hline \{\,t.\ \mathrm{store}\ t\ \emptyset\,\}\end{array}$$

$$\begin{array}{c}\{\,\mathrm{store}\ t\ \sigma\,\}\\ \hline \texttt{ref}\ t\ v\\ \hline \{\,r.\ r \notin \mathrm{dom}(\sigma) * \mathrm{store}\ t\ \sigma[r \mapsto v]\,\}\end{array}$$

$$\begin{array}{c}\{\,\mathrm{store}\ t\ \sigma * \sigma(r) = v\,\}\\ \hline \texttt{get}\ t\ r\\ \hline \{\,v.\ \mathrm{store}\ t\ \sigma\,\}\end{array} \qquad \begin{array}{c}\{\,\mathrm{store}\ t\ \sigma * r \in \mathrm{dom}(\sigma)\,\}\\ \hline \texttt{set}\ t\ r\ v\\ \hline \{\,v.\ \mathrm{store}\ t\ \sigma[r \mapsto v]\,\}\end{array}$$

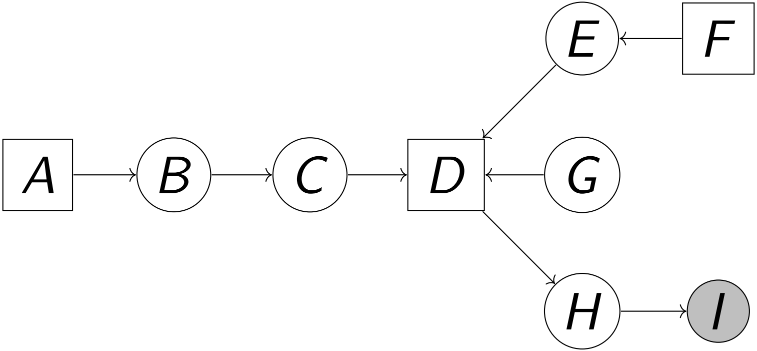
. . . avec des versions persistantes

$$\begin{array}{c} \{\, \mathrm{store}\ t\ \sigma \,\} \\ \hline \texttt{capture}\ t \\ \hline \{\, s.\, \mathrm{store}\ t\ \sigma * \mathrm{snapshot}\ t\ s\ \sigma \,\} \end{array}$$

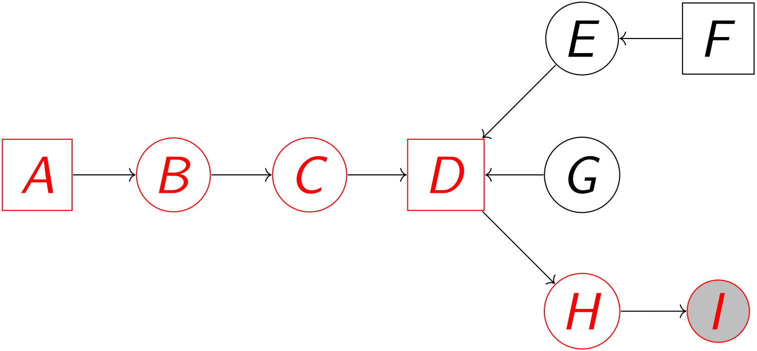
$$\begin{array}{c} \{\, \mathrm{store}\ t\ \sigma * \mathrm{snapshot}\ t\ s\ \sigma' \,\} \\ \hline \texttt{restore}\ t\ s \\ \hline \{\, ().\, \mathrm{store}\ t\ \sigma' \,\} \end{array}$$

Merci de votre attention!

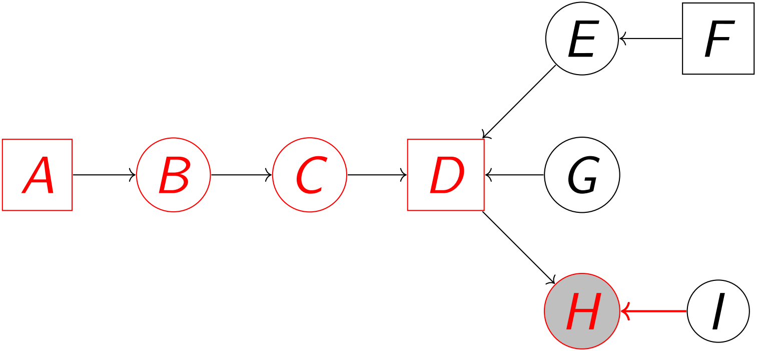
Implementation *without* elision



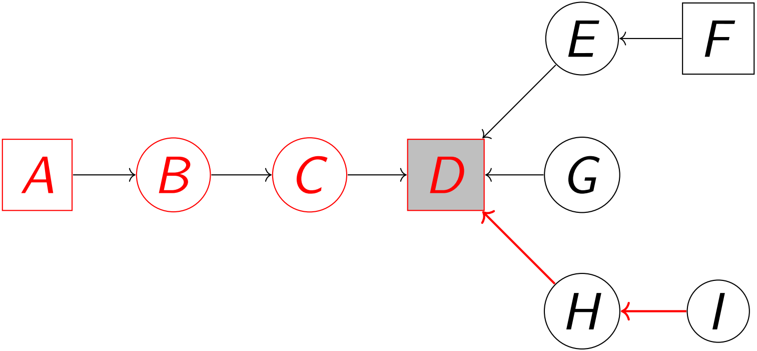
Rerooting *without* elision



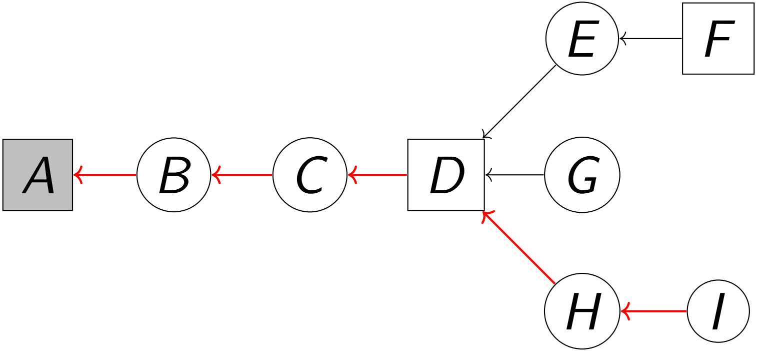
Rerooting *without* elision



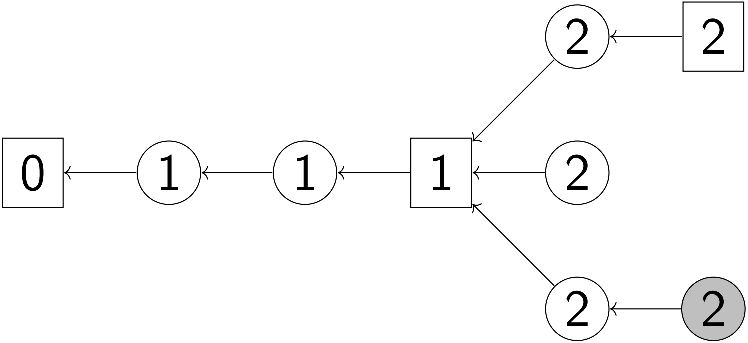
Rerooting *without* elision



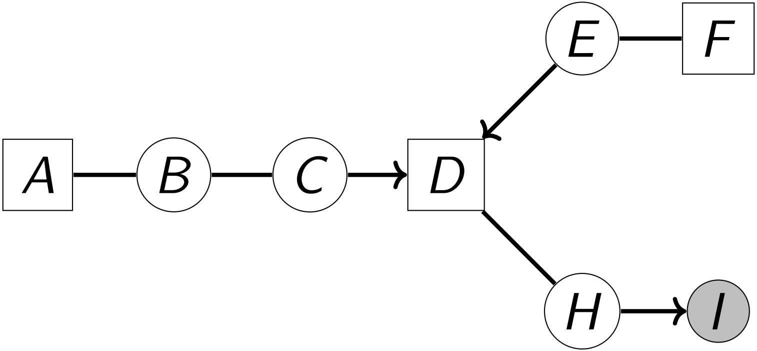
Rerooting *without* elision



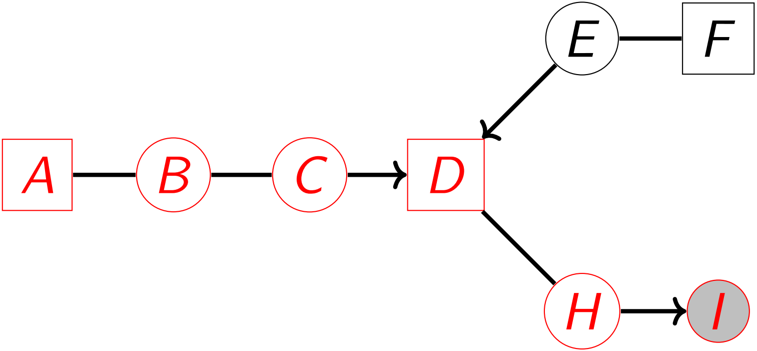
Historical tree & generations



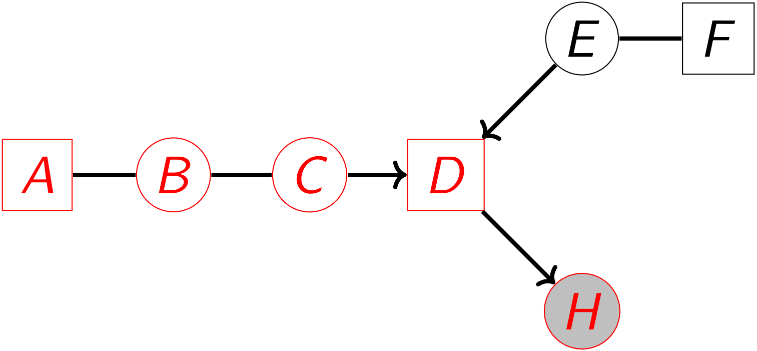
Implementation *with* elision



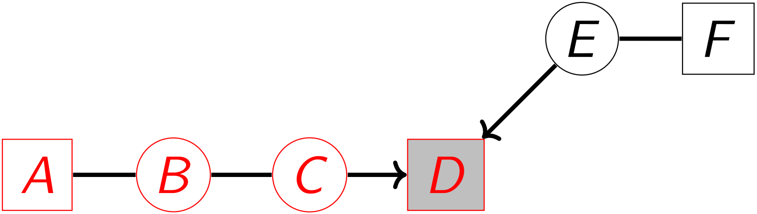
Rerooting *with* elision



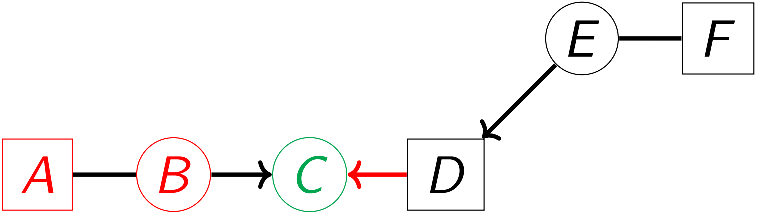
Rerooting *with* elision



Rerooting *with* elision



Rerooting *with* elision



Rerooting *with* elision