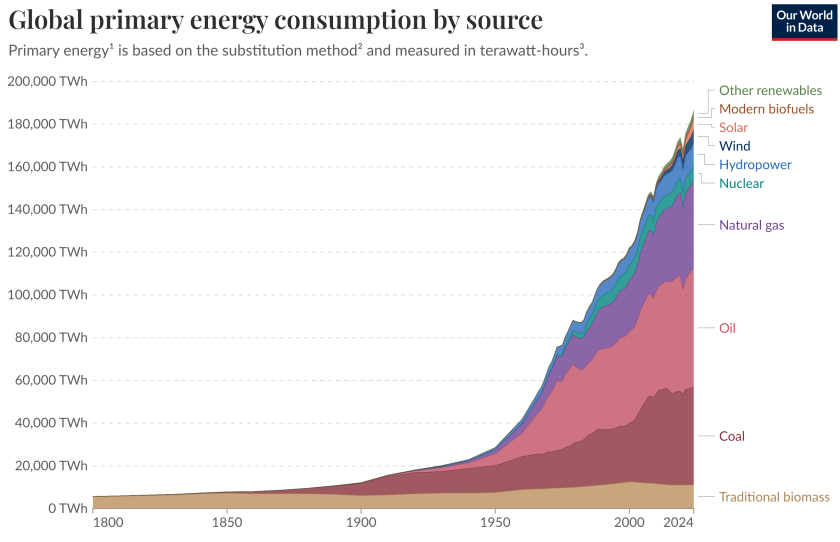
Verification of fine-grained concurrent OCaml 5 algorithms using separation logic

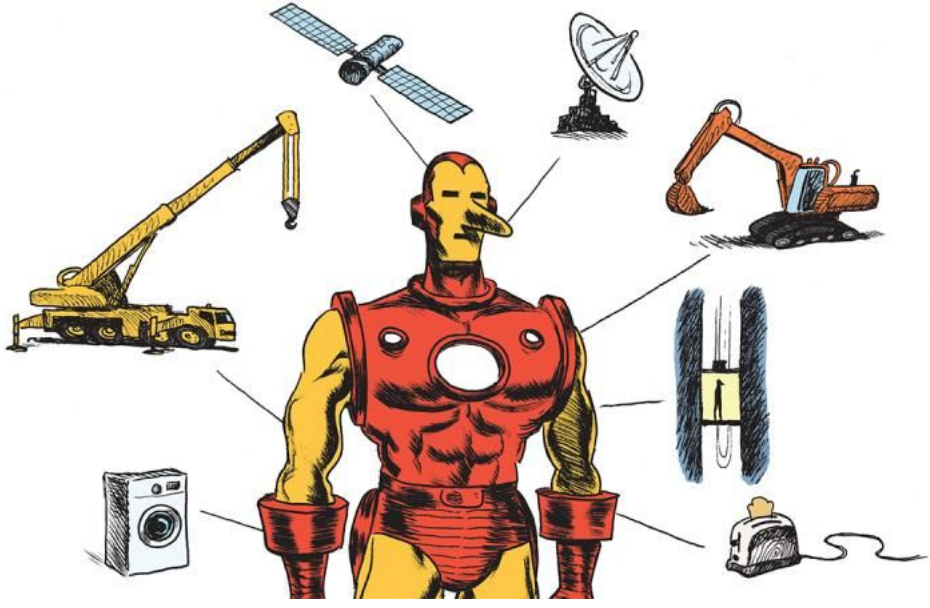
Clément Allain

December 17, 2025

Energy

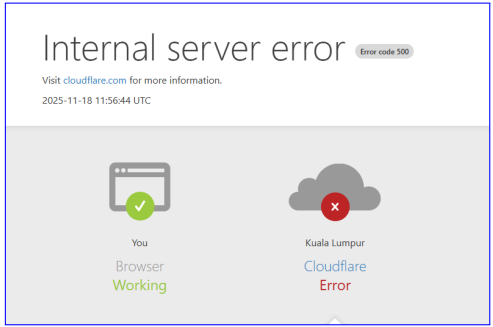

Global primary energy consumption by source
Primary energy[1] is based on the substitution method[2] and measured in terawatt-hours[3].
Our World in Data
200,000 TWh
180,000 TWh
160,000 TWh
140,000 TWh
120,000 TWh
100,000 TWh
80,000 TWh
60,000 TWh
40,000 TWh
20,000 TWh
0 TWh
1800
1850
1900
1950
2000
2024
Other renewables
Modern biofuels
Solar
Wind
Hydropower
Nuclear
Natural gas
Oil
Coal
Traditional biomass
Data source: Energy Institute - Statistical Review of World Energy (2025); Smil (2017)
Note: In the absence of more recent data, traditional biomass is assumed constant since 2015.
OurWorldinData.org/energy | CC BY

Energy... for machines



Programming languages... to control machines

Software failure: Cloudflare outage (18/11/2025)



3h outage
(20% of the web)

Bot management configuration file

```
71      /// Fetch edge features based on `input` struct into [`Features`] buffer.
72      pub fn fetch_features(
73          &mut self,
74          input: &dyn BotsInput,
75          features: &mut Features,
76      ) -> Result<(), (ErrorFlags, i32)> {
77          // update features checksum (lower 32 bits) and copy edge feature names
78          features.checksum &= 0xFFFF_FFFF_0000_0000;
79          features.checksum |= u64::from(self.config.checksum);
80          let (feature_values, _) = features
81              .append_with_names(&self.config.feature_names)
82              .unwrap();
```

Software failure: Ariane 5 (04/06/1996)

$500M

Integer overflow

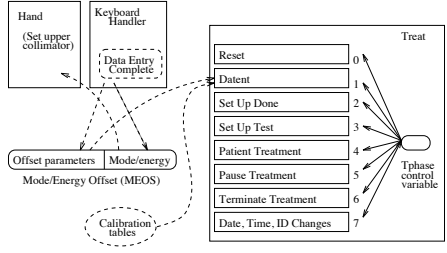
```
L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
                                G_M_INFO_DERIVE(T_ALG.E_BV));
if L_M_BV_32 > 32767 then
   P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
   P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M
end if;

P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                ((1.0/C_M_LSB_BH) *
                                G_M_INFO_DERIVE(T_ALG.E_BH)))
end LIRE_DERIVE;
```

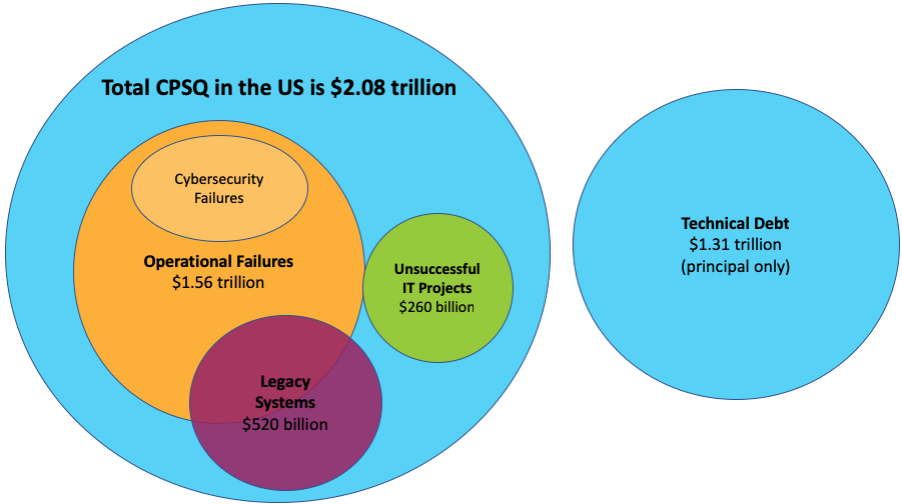
Software failure: Therac-25 (06/1985 - 01/1987)

6 human lives

Race condition (concurrency bug)

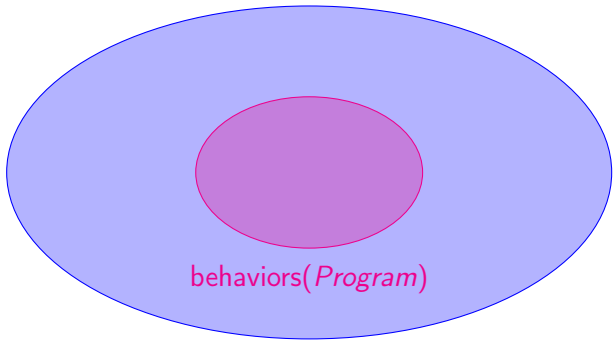


Cost of Poor Software Quality (CPSQ) in the US in 2020



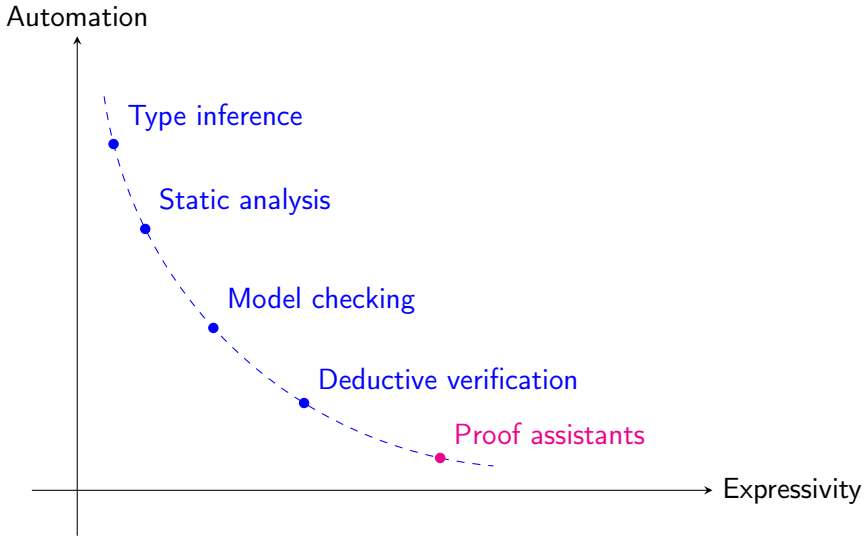
(Consortium for Information & Software Quality)

Formal methods
Improve the confidence in software

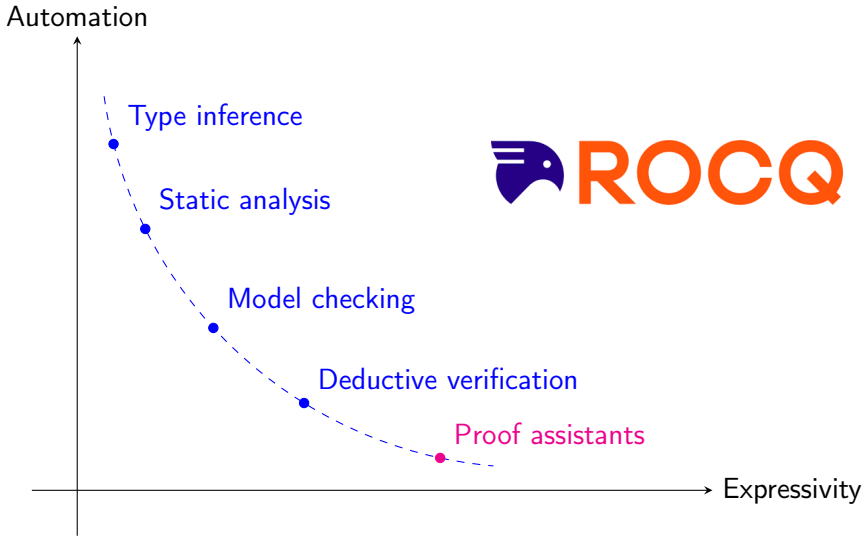


Property
(correctness, security, time usage, space usage)

Formal methods



Formal methods



ROCQ

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

`Parabs`: A library of parallel abstractions

Chase-Lev work-stealing deque

Conclusion

- Typically *French* programming language (it's quite good).

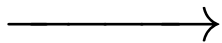
- **Functional**: first-class functions, algebraic data types.
- **Imperative**: references, mutable records.
- **Strongly typed**: well-typedness $\implies$ safety.
- **Garbage-collected**: automatic memory management.

From OCaml 5 (2022):

- **Parallel**: domains, atomic references, blocking mechanisms.
- **Concurrent**: algebraic effects.

Zoo

`ocaml2zoo`



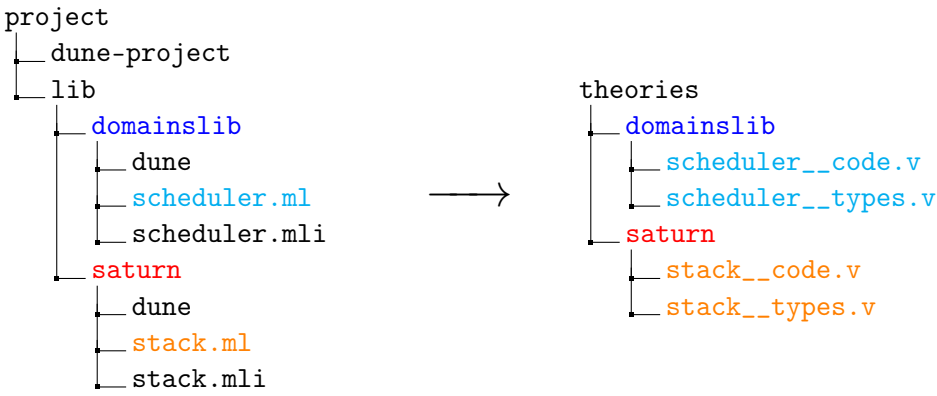
Zoo

Iris separation logic (2015)

- Higher-order ghost state
- User-defined ghost state
- Mechanized in Rocq
- Basic automation through Diaframe

- Invariants
- Atomic updates
- Prophecy variables

Zoo in practice




```
$ ocaml2zoo project theories
```

Zoo in practice

```
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
  )
```


```
Definition stack_push : val :=
  rec: "stack_push" "t" "v" =>
    let: "old" := !"t" in
    let: "new" := ‘Cons( "v", "old" ) in
    if: ~ CAS "t" "old" "new" then (
      domain_cpu_relax () ;;
      "stack_push" "t" "v"
    ).
```

Zoo in practice

```
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.
```

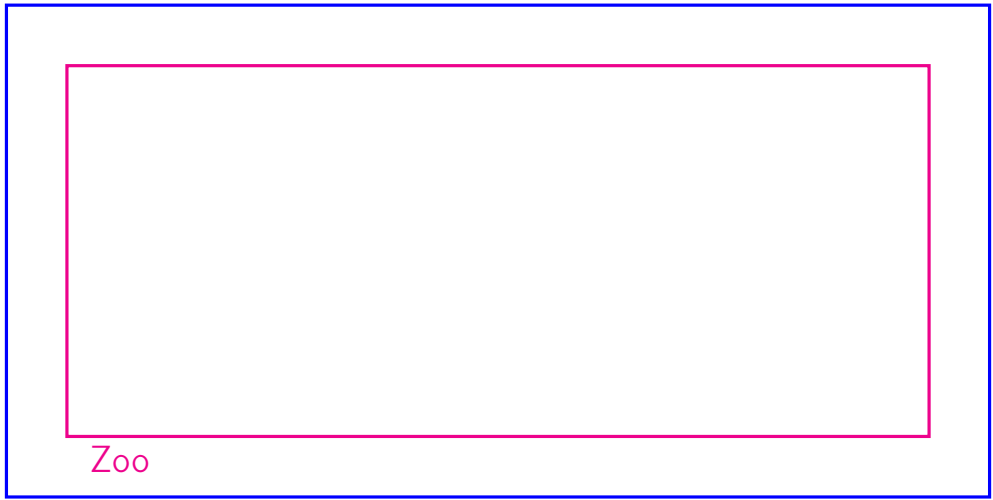
stack_push is *linearizable*

Zoo features

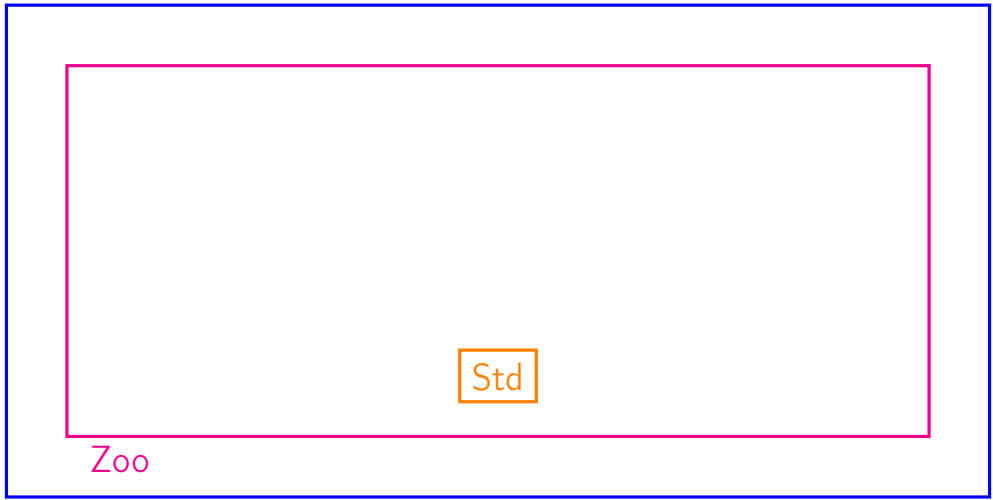
- Algebraic data types
- Records
- Mutually recursive functions
- Physical equality
- Structural equality
- Prophecy variables
- Diaframe (basic automation)
- Standard library
- Atomic references
- Atomic record fields
- Atomic arrays
- Generative constructors

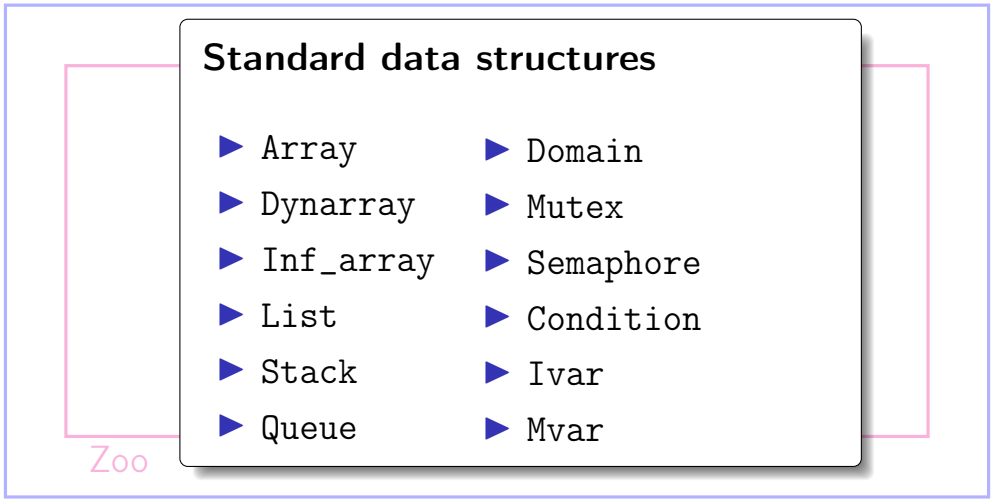
Contributions

Contributions

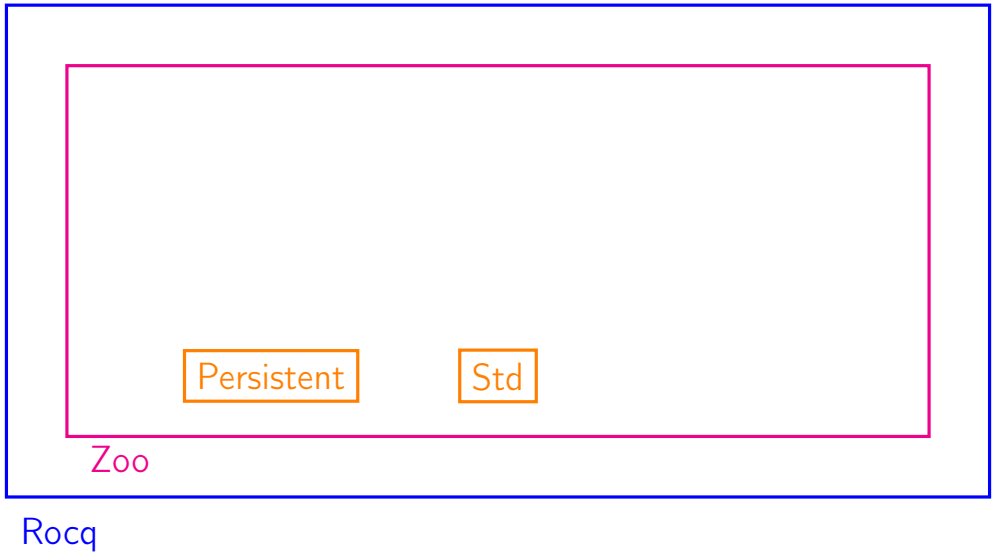


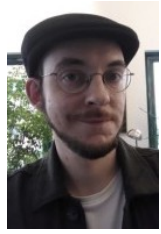
Contributions



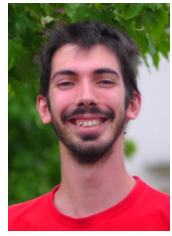

Standard data structures
Array
Dynarray
Inf_array
List
Stack
Queue
Domain
Mutex
Semaphore
Condition
Ivar
Mvar
Zoo
Rocq

Contributions





Basile Clément



Gabriel Scherer

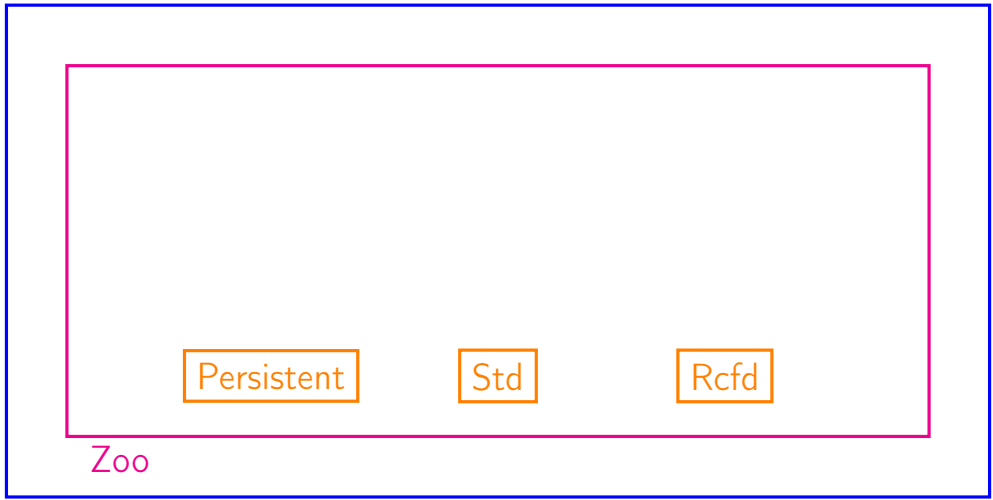
Persistent data structures

- Persistent array
- Persistent store
- Persistent union-find

Zoo

Rocq

Contributions



Thomas Leonard

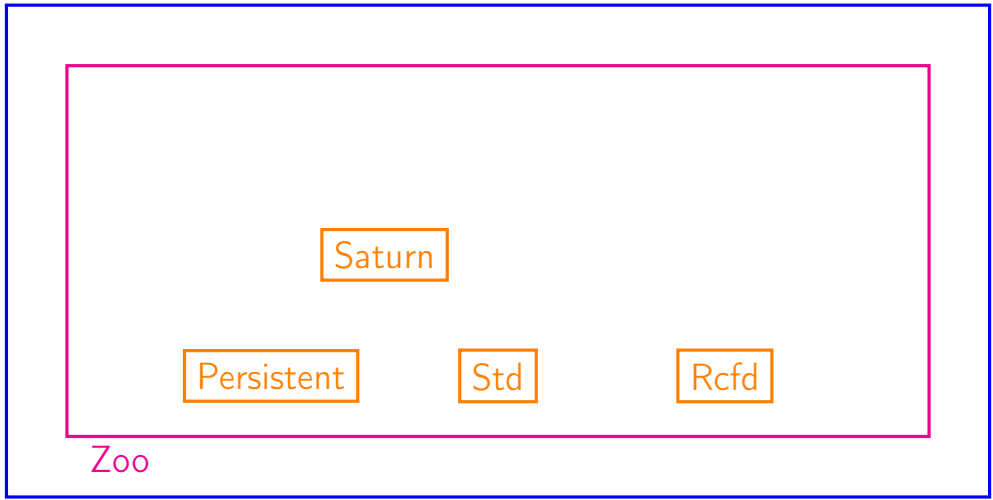
Parallelism-safe file descriptor

- Generative constructors
- Intricate concurrent protocol
- Two ownership regimes

Zoo

Rocq

Contributions



Vesa Karvonen

Carine Morel

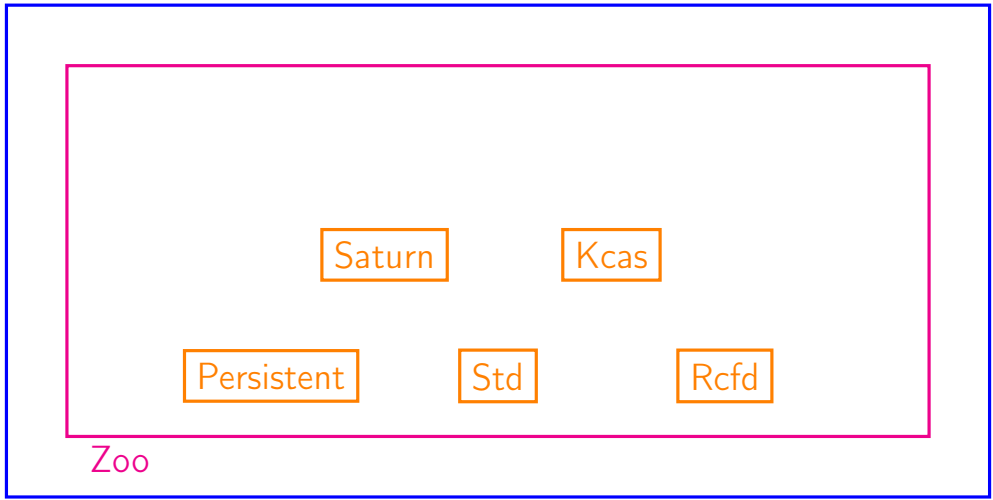
Standard lock-free data structures

- Stacks
- List-based queues
- Array-based queues
- Stack-based queues
- Work-stealing deques

Zoo

Rocq

Contributions



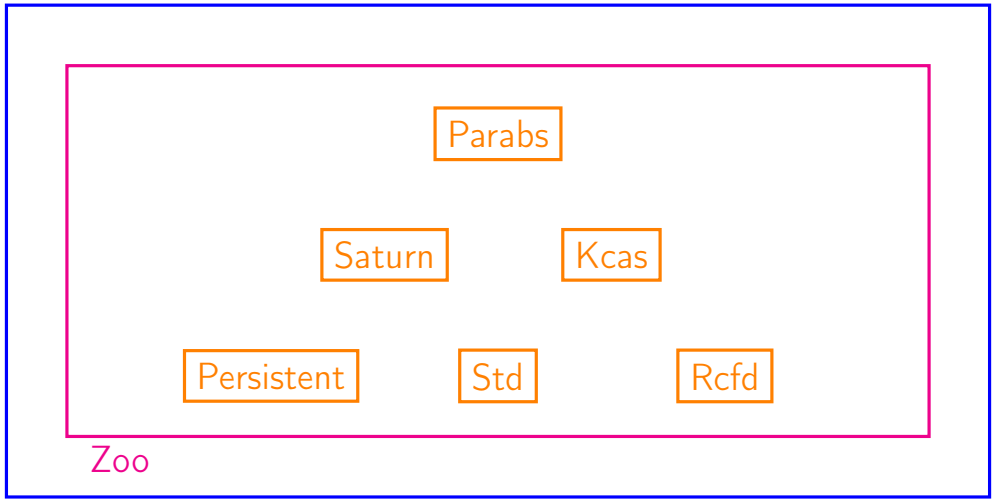
Vesa Karvonen

Lock-free multi-word compare-and-set
(at the core of the `Kcas` library)

Zoo

Rocq

Contributions



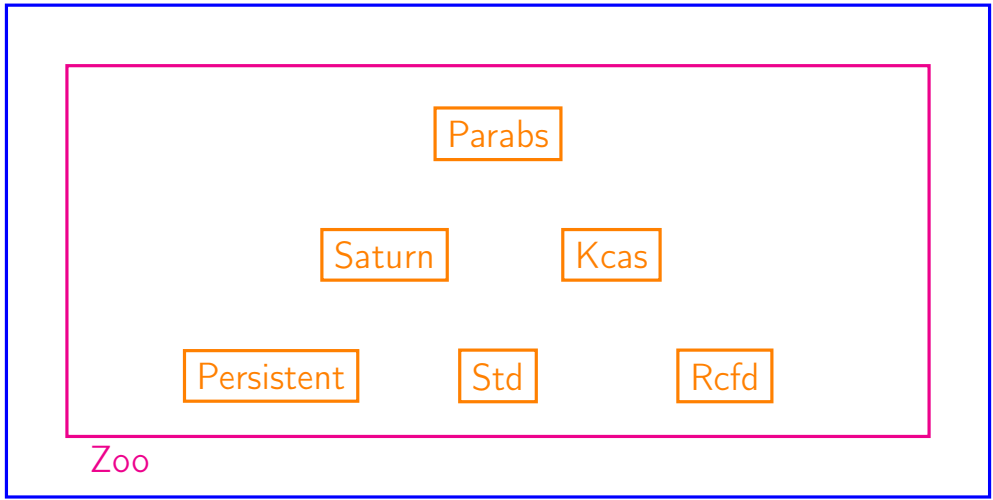
Parallel abstractions

- Work-stealing scheduler
- Futures
- Parallel iterators
- Task graph (DAG-calculus)

Zoo

Rocq

Contributions



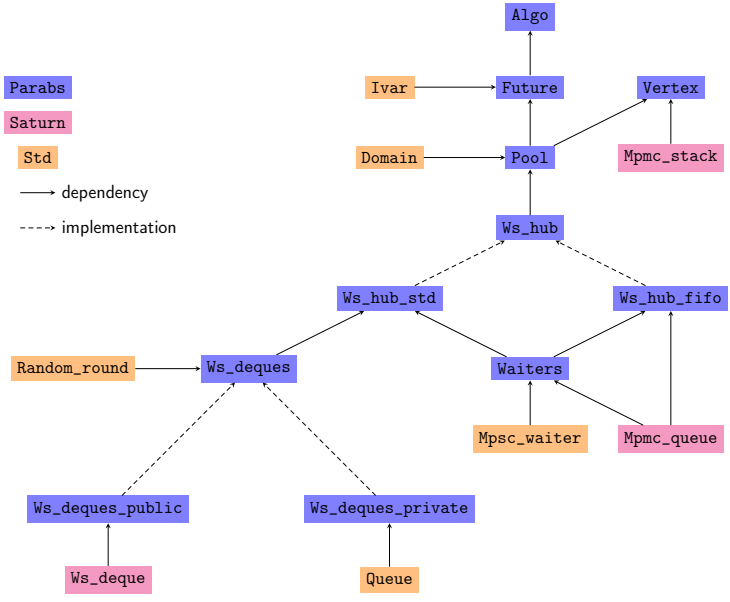
Zoo: A framework for the verification of concurrent OCaml 5 programs

`Parabs`: A library of parallel abstractions

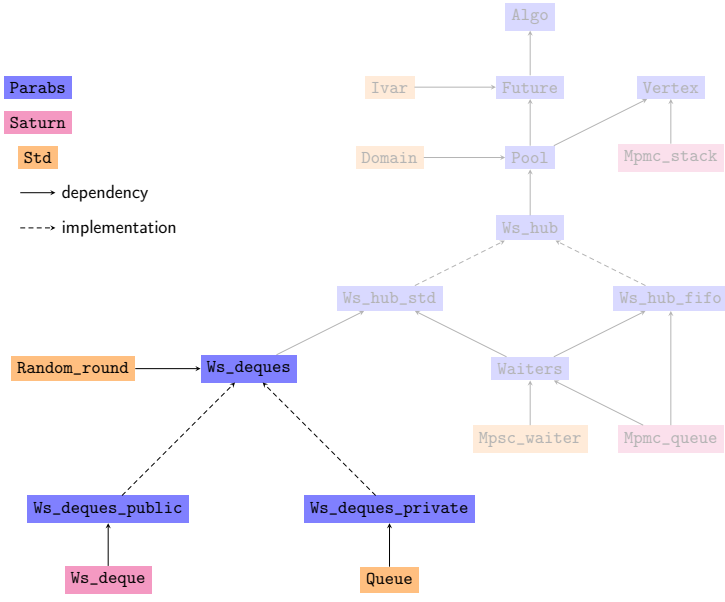
Chase-Lev work-stealing deque

Conclusion

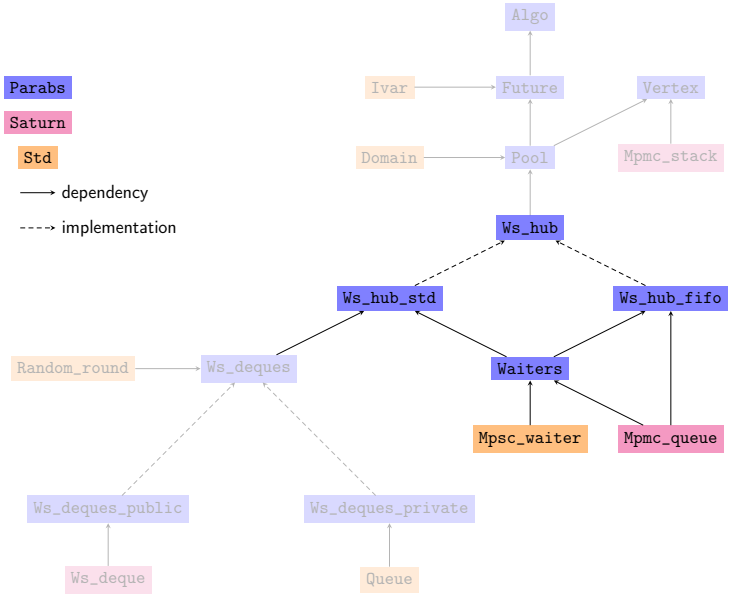
Overview


Algo
Parabs
Saturn
Std
dependency
implementation
Ivar
Future
Vertex
Domain
Pool
Mpmc_stack
Ws_hub
Ws_hub_std
Ws_hub_fifo
Random_round
Ws_deques
Waiters
Mpsc_waiter
Mpmc_queue
Ws_deques_public
Ws_deques_private
Ws_deque
Queue

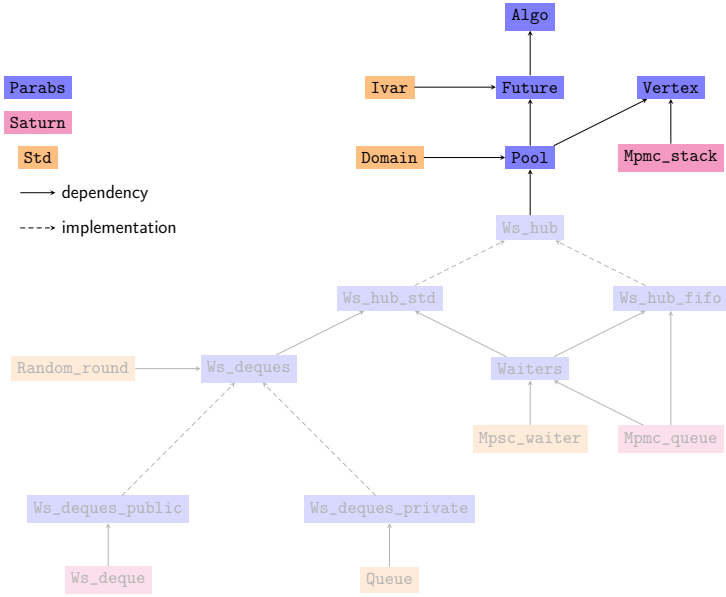
Overview


Parabs
Saturn
Std
dependency
implementation
Algo
Ivar
Future
Vertex
Domain
Pool
Mpmc_stack
Ws_hub
Ws_hub_std
Ws_hub_fifo
Random_round
Ws_deques
Waiters
Mpsc_waiter
Mpmc_queue
Ws_deques_public
Ws_deques_private
Ws_deque
Queue

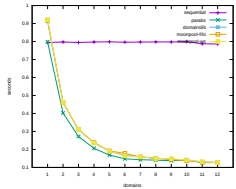
Overview


Parabs
Saturn
Std
dependency
implementation
Algo
Ivar
Future
Vertex
Domain
Pool
Mpmc_stack
Ws_hub
Ws_hub_std
Ws_hub_fifo
Random_round
Ws_deques
Waiters
Mpsc_waiter
Mpmc_queue
Ws_deques_public
Ws_deques_private
Ws_deque
Queue

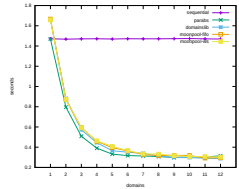
Overview


Parabs
Saturn
Std
dependency
implementation
Algo
Ivar
Future
Vertex
Domain
Pool
Mpmc_stack
Ws_hub
Ws_hub_std
Ws_hub_fifo
Random_round
Ws_deques
Waiters
Mpsc_waiter
Mpmc_queue
Ws_deques_public
Ws_deques_private
Ws_deque
Queue

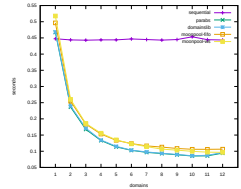
Benchmarks



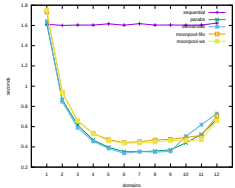
(a) `fibonacci`



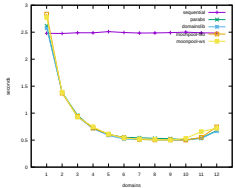
(b) `for_irregular`



(c) `iota`



(d) `lu`



(e) `matmul`

`Parabs` **has equal or better performance than** `Domainslib`.

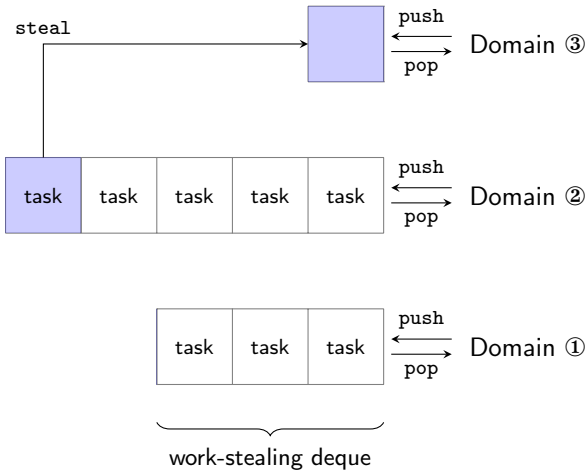
Zoo: A framework for the verification of concurrent OCaml 5 programs

`Parabs`: A library of parallel abstractions

Chase-Lev work-stealing deque

Conclusion

Work-stealing



Chase-Lev work-stealing deque

Dynamic Circular Work-Stealing Deque

David Chase
Sun Microsystems Laboratories
Mailstop UBUR02-311
1 Network Drive
Burlington, MA 01803, USA
dr2chase@sun.com

Yossi Lev
Brown University & Sun Microsystems Laboratories
Mailstop UBUR02-311
1 Network Drive
Burlington, MA 01803, USA
yosef.lev@sun.com

ABSTRACT

The non-blocking work-stealing algorithm of Arora, Blumofe, and Plaxton (henceforth *ABP work-stealing*) is on its way to becoming the multiprocessor load balancing technology of choice in both industry and academia. This highly efficient scheme is based on a collection of array-based double-ended queues (deques) with low cost synchronization among local and stealing processes. Unfortunately, the algorithm's synchronization protocol is strongly based on the use of fixed size arrays, which are prone to overflows, especially in the multiprogrammed environments for which they are designed. We present a work-stealing deque that does not have the overflow problem.

The only ABP-style work-stealing algorithm that eliminates the overflow problem is the list-based one presented by Hendler, Lev and Shavit. Their algorithm indeed deals with the overflow problem, but it is complicated, and introduces a trade-off between the space and time complexity, due to the extra work required to maintain the list.

Our new algorithm presents a simple lock-free work-stealing deque, which stores the elements in a cyclic array that can grow when it overflows. The algorithm has no limit other than integer overflow (and the system's memory size) on the

1. INTRODUCTION

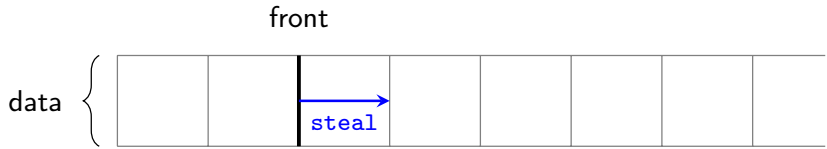
The ABP work-stealing algorithm of Arora, Blumofe, and Plaxton [2] has been gaining popularity as the multiprocessor load-balancing technology of choice in both industry and academia [2, 1, 4, 9]. The scheme implements a provably efficient work-stealing paradigm due to Blumofe and Leiserson [3] that allows each process to maintain a local work deque,[1] and steal an item from others if its deque becomes empty. The deque's owner process pushes and pops local work to and from the deque's bottom end. To minimize synchronization overhead for the deque's owner, stolen elements are taken from the top end of the deque. No elements are added to the top end of the deque. An ABP deque thus presents three methods in its interface:

- `pushBottom(Object o)`:
  Pushes *o* onto the bottom of the deque.
- `Object popBottom()`:
  Pops an object from the bottom of the deque if the deque is not empty, otherwise returns *Empty*.
- `Object steal()`:
  If the deque is empty, returns *Empty*. Otherwise, re-

Physical state

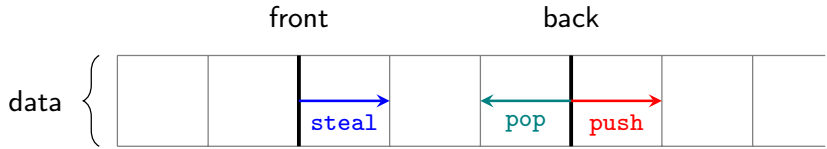
data: infinite array storing all values

Physical state



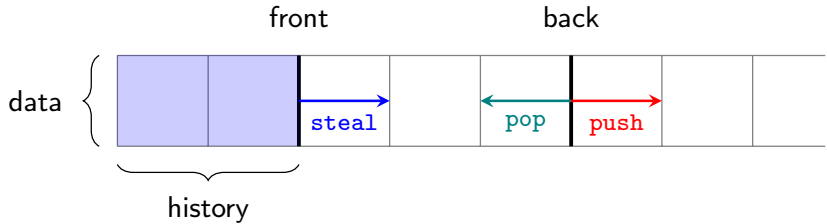
data: infinite array storing all values
front: *monotonic* thieves' index

Physical state



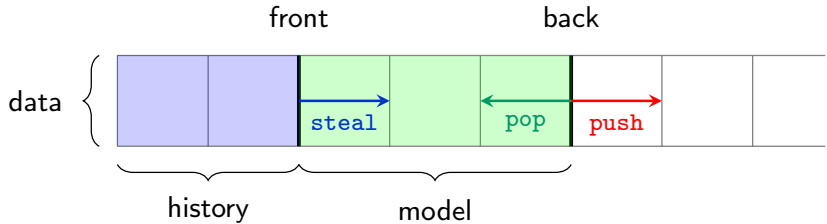
data:	infinite array storing all values
front:	*monotonic* thieves' index
back:	owner's index

Physical state



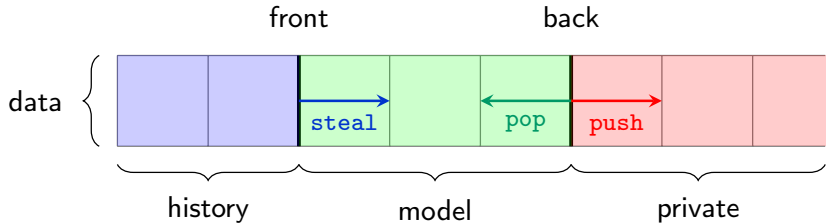
data:	infinite array storing all values
front:	*monotonic* thieves' index
back:	owner's index
history:	*monotonic* list of values

Physical state



data:	infinite array storing all values
front:	*monotonic* thieves' index
back:	owner's index
history:	*monotonic* list of values
model:	logical content of the deque

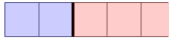
Physical state



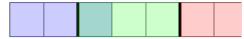
data:	infinite array storing all values
front:	*monotonic* thieves' index
back:	owner's index
history:	*monotonic* list of values
model:	logical content of the deque
private:	owner-only region

Logical state

① Empty

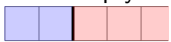


② Non-empty

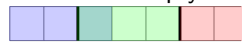


Logical state

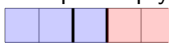
① Empty



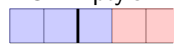
② Non-empty



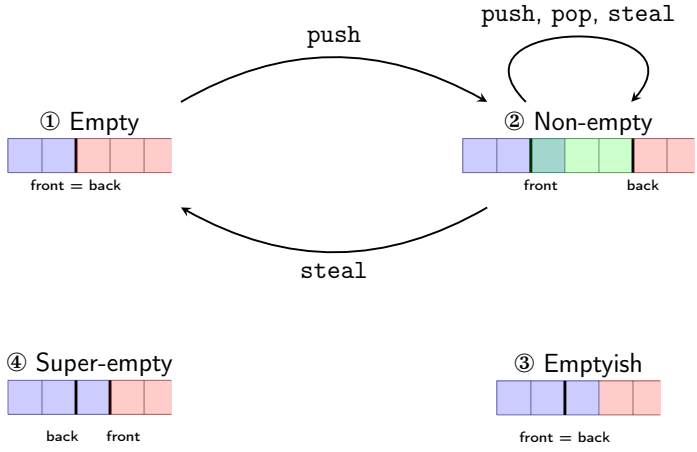
④ Super-empty



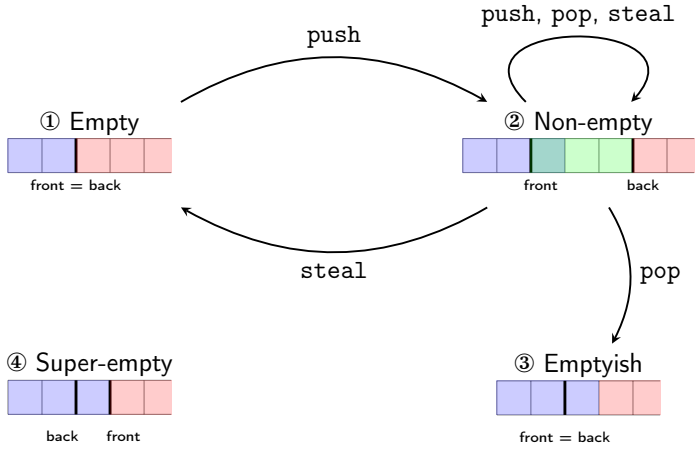
③ Emptyish



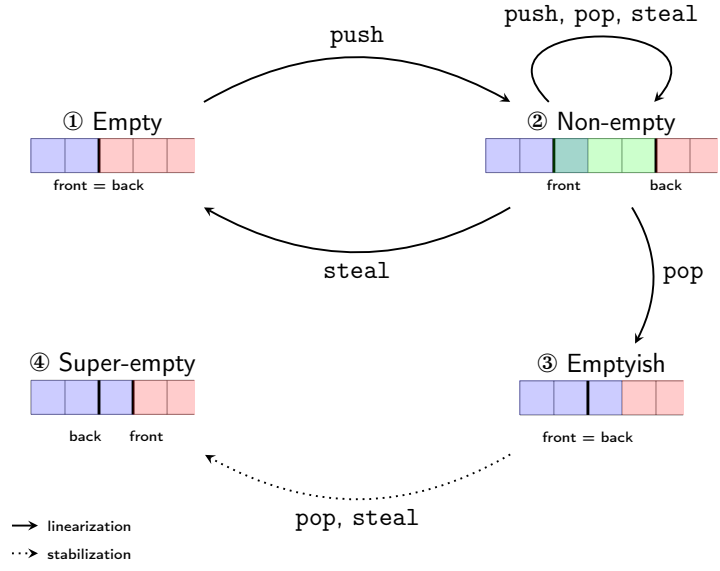
Logical state



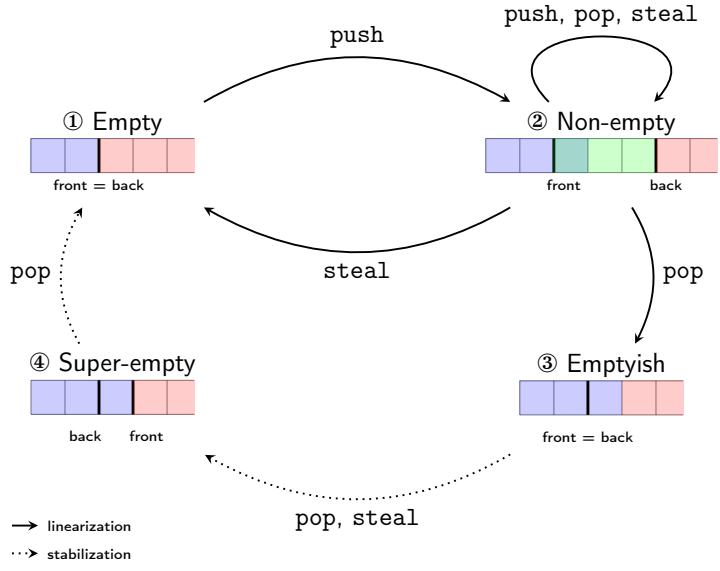
Logical state



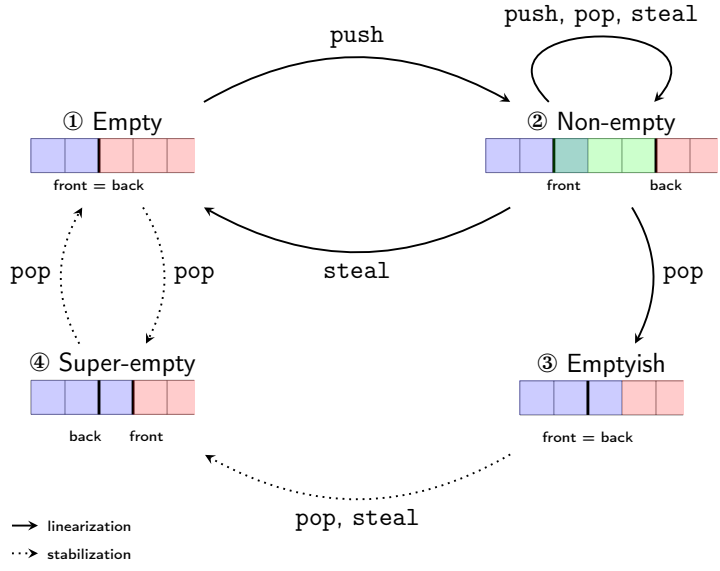
Logical state



Logical state



Logical state



Prophecy variables / Prophets

- **Primitive prophets**
  Predict the future of the execution.
- **Typed prophets**
  Prediction belongs to a user-supplied semantic type.
- **Wise prophets**
  Remember past predictions.
  Eliminate inconsistent branches.
- **Multiplexed prophets**
  Separate logically independent predictions on a single prophet.

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

`Parabs`: A library of parallel abstractions

Chase-Lev work-stealing deque

Conclusion

Takeaway

Iris-based verification frameworks can scale to real-life programming languages and (relatively) large pieces of software.

- Practical verification framework.
- Formalization of a realistic OCaml subset.
- Verification of realistic concurrent data structures.
- Extensions of Iris proof techniques.
- OCaml language improvements.

Iris-based verification frameworks can scale to real-life programming languages and (relatively) large pieces of so

Not mentioned in this presentation:

- Memory safety.
- "Tail Modulo Cons" program transformation.
- Ongoing work on (parts of) the OCaml GC.

- Extensions of Iris proof techniques.
- OCaml language improvements.

Future work

- **Language features**
  - Exceptions
  - Algebraic effects
  - Modules & functors
- **Coupling with semi-automated verification**
- **Relaxed memory**

Methodology by Glen Mével *et al.*:

1. Start with the invariant under sequential consistency;
2. Identify how information flows between domains,
   i.e. where the synchronization points are;
3. Refine the invariant with memory views accordingly.

Thank you for your attention!