Université Paris Cité
École doctorale de sciences mathématiques de Paris Centre (ED 386)
INRIA

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

Clément ALLAIN

Thèse de doctorat d'informatique

dirigée par François POTTIER et Gabriel SCHERER

Abstract

The release of OCaml 5 in December 2022 introduced parallelism in the OCaml runtime. It drove the need for safe and efficient concurrent data structures. New libraries like `Saturn` address this need. This is an opportunity to apply and further state-of-the-art program verification techniques.

We present Zoo, a framework for verifying fine-grained concurrent OCaml 5 algorithms. Following a pragmatic approach, we define a limited but sufficient fragment of the language to faithfully express these algorithms: ZooLang. We formalize its semantics carefully via a deep embedding in the Rocq proof assistant, uncovering subtle aspects of physical equality. We provide a tool to translate source OCaml programs into ZooLang syntax embedded inside Rocq, where they can be specified and verified using the Iris concurrent separation logic.

We illustrate the use of Zoo via a number of case studies: a subset of the OCaml standard library, a library of persistent data structures, a parallelism-safe file descriptor from the `Eio` library, a collection of fined-grained concurrent data structures from the `Saturn` library, a task scheduler based on the `Domainslib` library, a state-of-the-art multi-word compare-and-set algorithm at the core of the `Kcas` library.

In `Saturn`, we verify stacks, queues (list-based, array-based, stack-based), bags and work-stealing deques. To cover a wide range of use cases, we provide specialized variants: bounded or unbounded, single-producer or multi-producer, single-consumer or multi-consumer. In particular, we prove strong specifications for the Chase-Lev work-stealing deque, which involves intricate logical state and advanced use of Iris prophecy variables.

In the process, we also extend OCaml to more efficiently express certain concurrent programs, by introducing atomic record fields and atomic arrays. Our work on formalizing the semantics of physical equality revealed that it is under-specified in existing descriptions of the language; in existing verification frameworks, the feature is also too restricted to support compare-and-set in idiomatic OCaml concurrent programs.

Keywords. OCaml, Rocq, program verification, separation logic, concurrent data structures

Résumé

La sortie d'OCaml 5 en décembre 2022 a introduit le parallélisme dans le langage OCaml. Cela a suscité le besoin de structures de données concurrentes sûres et efficaces. De nouvelles bibliothèques comme `Saturn` répondent à ce besoin. C'est une opportunité d'appliquer et de faire progresser les techniques de vérification de programmes de pointe.

Nous présentons Zoo, un cadriciel pour la vérification d'algorithmes OCaml 5 concurrents à grain fin. Suivant une approche pragmatique, nous définissons un fragment limité mais suffisant du langage pour exprimer fidèlement ces algorithmes : ZooLang. Nous formalisons soigneusement sa sémantique via un plongement profond dans l'assistant de preuve Rocq, en insistant sur certains aspects subtils de l'égalité physique. Nous fournissons un outil de traduction de programmes OCaml en syntaxe ZooLang plongée dans Rocq, où ils peuvent être spécifiés et vérifiés à l'aide de la logique de séparation concurrente Iris.

Nous illustrons l'utilisation de Zoo à travers plusieurs études de cas : un sous-ensemble de la bibliothèque standard d'OCaml, une bibliothèque de structures de données persistantes, un descripteur de fichier sûr pour le parallélisme issu de la bibliothèque `Eio`, une collection de structures de données concurrentes à grain fin provenant de la bibliothèque `Saturn`, un ordonnanceur de tâches basé sur la bibliothèque `Domainslib`, un algorithme compare-and-set multi-mot de pointe au cœur de la bibliothèque `Kcas`.

Dans `Saturn`, nous vérifions des piles, des files (basées sur des listes, des tableaux ou des piles) et des sacs. Afin de couvrir un large éventail d'utilisations, nous proposons des variantes spécialisées : bornées ou non bornées, à producteur unique ou multiple, à consommateur unique ou multiple. En particulier, nous prouvons des spécifications fortes pour la file de vol de tâches de Chase-Lev, ce qui implique un état logique complexe ainsi qu'un usage avancé des variables prophétiques d'Iris.

Ce faisant, nous étendons également OCaml afin d'exprimer plus efficacement certains programmes concurrents, en introduisant des champs d'enregistrement atomiques et des tableaux atomiques. Notre travail de formalisation de la sémantique de l'égalité physique a révélé que cette dernière est sous-spécifiée dans les descriptions existantes du langage ; dans les travaux antérieurs en vérification, cette notion est par ailleurs trop limitée pour permettre l'utilisation de compare-and-set dans des programmes OCaml concurrents idiomatiques.

Mots-clés. OCaml, Rocq, vérification de programmes, logique de séparation, structures de données concurrentes

Contents

Chapter 1

Introduction

1.1 About Coq

In this thesis, all proofs are mechanized in the Rocq proof assistant, formerly known as Coq. We shall systematically use the new name — no offense meant. For those who cannot tolerate it — especially French people waiting for bits to be also renamed —, we can provide an alternative Coq-ed version.

1.2 A little story

Cambium. In the basement[1] of 48 rue Barrault, Paris, live not antiques but the well-known Cambium tribe. At Cambium, people have faith, commune and spread the word of God. On the walls and above the coffee machine, OCaml posters testify to their enduring commitment.

We were enthusiastically although critically part of the tribe for three years. This is the (slightly romanticized) story of our thesis.

Divine command. God [2022] once said:

> "You shall verify everything in OCaml."

So we went verifying everything in OCaml: program transformations, (parts of) the runtime system, tricky programs and libraries (both sequential and concurrent), *etc.* Unfortunately, though, we did not have ten years to finish this thesis. Consequently, we focus on the most important part of our work: the development of the Zoo framework for the verification of realistic fine-grained concurrent algorithms. We leave the verification of the rest of the OCaml kingdom for future work.

OCaml 5. In December 2022, at the start of our thesis, OCaml 5 was released by merging the Multicore OCaml [Sivaramakrishnan et al., 2020] runtime. It is the first version of OCaml to support parallelism. It provided basic parallel programming facilities through the standard library, including parallel threads called "domains", atomic references and blocking synchronization mechanisms. The third-party library `Domainslib` offered a simple task scheduler, used to benchmark the parallel runtime. A world of parallel software was waiting to be invented.

[1] At the "garden level", to be more precise — although the garden is less obvious than the trash cans.

A growing ecosystem. Shared-memory parallelism is a difficult programming domain; existing ecosystems (C++, Java, Haskell, Rust, Go...) took decades to evolve comprehensive libraries of parallel abstractions and data structures. In the last couple years, a handful of contributors to the OCaml ecosystem have been implementing libraries for concurrent and parallel programming, in particular `Saturn` [Karvonen and Morel, 2025b], a library of lock-free concurrent data structures, `Eio` [Madhavapeddy and Leonard, 2025], a library for asynchronous IO and structured concurrency, and `Kcas` [Karvonen, 2025a], an implementation of software transactional memory.

Verification of concurrent algorithms. Concurrent algorithms, especially fine-grained ones, are difficult to reason about. Their implementation tend to be fairly short, a few dozens of lines. There is only a handful of experts able to write such code, and many potential users. They are difficult to test comprehensively. These characteristics make them ideally suited for mechanized program verification.

We embarked on a mission to mechanize correctness proofs of OCaml concurrent algorithms and data structures as they are being written, in contact with their authors, rather than years later. In the process, we not only gained confidence in these complex new building blocks, but also improved the OCaml language and its verification ecosystem.

1.3 Iris

The state-of-the-art approach for mechanized verification of fine-grained concurrent algorithms is Iris [Jung et al., 2018b], a mechanized higher-order concurrent separation logic [Brookes and O'Hearn, 2016] with user-defined ghost state.

Its expressivity allows to precisely capture subtle invariants and reason about exotic concurrent behaviors [Dongol and Derrick, 2014], especially external [Vindum et al., 2022] and future-dependent [Jung et al., 2020; Vindum and Birkedal, 2021; Chang et al., 2023; Patel et al., 2024] linearization points.

Iris has been used successfully to verify various concurrent data structures: stacks [Iris development team, 2025b; Jung et al., 2023], queues [Jung et al., 2020; Vindum and Birkedal, 2021; Mével and Jourdan, 2021; Vindum et al., 2022; Carbonneaux et al., 2022; Jung et al., 2023; Somers and Krebbers, 2024], a priority queue [Park et al., 2025], search structure templates [Krishna et al., 2020; Patel et al., 2021, 2024; Nguyen et al., 2024; Park et al., 2025], skiplists [Carrott, 2022; Park et al., 2025], a binary search tree [Sharma, 2021].

Iris also supports relaxed memory verification [Mével et al., 2020; Mével and Jourdan, 2021; Dang et al., 2022; Park et al., 2024, 2025; Jung et al., 2025]. In our work, we assume a sequentially consistent memory model, but moving to the OCaml 5 relaxed memory model [Dolan et al., 2018] is the next thing on the list.

Moreover, Iris comes with basic automation thanks to Diaframe [Mulder et al., 2022; Mulder and Krebbers, 2023]. We use it extensively in our work.

1.4 Overview

The rest of the thesis is organized as follows.

Preparations. In Chapter 2, we review OCaml 5's parallel programming facilities, including new features that we introduced in the language. In Chapter 3, we introduce the Iris [Jung et al., 2018b] concurrent separation logic, focusing on the mechanisms needed to verify concurrent programs.

Zoo. In Chapter 4, we introduce Zoo, a framework for verifying fine-grained concurrent OCaml 5 programs. In Chapter 5, we detail Zoo's support for prophecy variables [Jung et al., 2020], including new abstractions that we will be useful in Chapter 9.

Case studies. To illustrate the applicability of Zoo, we verify various sequential and concurrent case studies: standard data structures (Chapter 6), persistent data structures (Chapter 7), a parallelism-safe file descriptor from `Eio` [Madhavapeddy and Leonard, 2025] (Chapter 8), lock-free data structures mainly from `Saturn` [Karvonen and Morel, 2025b] (Chapter 9), a task scheduler (Chapter 10), an implementation of multi-word compare-and-set from `Kcas` [Karvonen, 2025a] (Chapter 11).

Memory safety. In Chapter 12, we address the question of memory safety in OCaml 5. We propose a formal methodology to verify it in Zoo using semantic typing à la RustBelt [Jung et al., 2018a].

1.5 OCaml code

The verified case studies represent a significant amount of code. Consequently, we decided not to include it in the body of the thesis. However, it is available online and we systematically provide links to the relevant parts in the text.

1.6 Rocq mechanization

Our results are mechanized in the Rocq proof assistant. Since proofs would be extremely tedious to reproduce, we do not present them in detail. For the most interesting ones, we describe important points.

1.7 Publications

Several parts of this thesis were published in the following articles:

- *Correct tout seul, sûr à plusieurs*,
  JFLA 2024,
  Clément Allain, Gabriel Scherer
  (Chapter 12).

- *Snapshottable Stores*,
  ICFP 2024,
  Clément Allain, Basile Clément, Alexandre Moine, Gabriel Scherer
  (Section 7.4).

- *Saturn: a library of verified concurrent data structures for OCaml 5*,
  OCaml Workshop 2024,
  Clément Allain, Vesa Karvonen, Carine Morel
  (Chapter 9).

- *Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic*,
  JFLA 2025,
  Clément Allain
  (Chapters 4, 6 and 8).

- *Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic*,
  POPL 2026,
  Clément Allain, Gabriel Scherer
  (Sections 2.3.2.2 and 2.3.2.3 and Chapters 4, 6 to 9 and 12).

Two other articles are in preparation, respectively presenting:

- The verification of the Chase-Lev work-stealing deque (Chapter 5 and Section 9.7) as implemented in the `Saturn` library and the verified `Parabs` library (Chapter 10) offering parallel abstractions atop a task scheduler,

- The verification of a state-of-the-art multi-word compare-and-set algorithm at the core of the `Kcas` library (Chapter 11).

In parallel, we also completed a previous research project on the verification of the Tail Modulo Cons program transformation. This work led to the following publication:

- *Tail Modulo Cons, OCaml, and Relational Separation Logic*,
  POPL 2025,
  Clément Allain, Frédéric Bour, Basile Clément, Francois Pottier, Gabriel Scherer.

Chapter 2

Parallelism in OCaml 5

OCaml 5 was released in December 2022. It is the first version of the OCaml programming language to support parallelism [Sivaramakrishnan et al., 2020]. In this chapter, we review the current parallel programming facilites, including primitive abstractions and third-party libraries.

2.1 Domains

OCaml 5 introduced *domains*[1], the units of parallelism. Domains are distinct from *threads*[2], the units of concurrency that existed before OCaml 5. Multiple domains can run OCaml code in parallel, on separate cores. Inside a domain, multiple threads can coexist; they are executed concurrently, one at a time. Domains and threads share the same memory space and garbage collector. Domain-local storage[3] is provided primitively. Currently, thread-local storage is not provided primitively but has been implemented in a third-party library[4].

In this thesis, especially in Chapter 4, we will only consider domains. More generally, we will focus on parallel facilites and mostly forget about concurrent facilites, including threads and algebraic effects [Sivaramakrishnan et al., 2021].

Domains are managed through the standard `Domain`[5] module. For example, one can (naively[6]) compute Fibonacci numbers using the `fibonacci` function of Figure 2.1. `Domain.spawn fn` creates a new domain to execute `fn`. `Domain.join d` blocks until domain `d` finishes and returns the result of its computation (`fn` in the previous example).

Another common primitive that we will often use in Chapter 9 is `Domain.cpu_relax`, of type `unit -> unit`. It is used to make a domain *back off* to reduce contention when multiple domains try to access some shared state in parallel. It is meant to improve performance and does not affect correctness. For even better performance, *exponential backoff*, as implemented in the `Backoff` [Karvonen and Morel, 2025a] library, may be relevant.

[1] `https://ocaml.org/manual/5.3/api/Domain.html`
[2] `https://ocaml.org/manual/5.3/api/Thread.html`
[3] `https://ocaml.org/manual/5.3/api/Domain.DLS.html`
[4] `https://github.com/c-cube/thread-local-storage`
[5] `https://ocaml.org/manual/5.3/api/Domain.html`
[6] Actually, this implementation rapidly fails as it spawns too many domains, see Section 2.4 for a correct implementation.

```
let rec fibonacci n =
  if n <= 1 then
    1
  else
    let dom1 = Domain.spawn (fun () -> fibonacci (n - 1)) in
    let dom2 = Domain.spawn (fun () -> fibonacci (n - 2)) in
    Domain.join dom1 + Domain.join dom2
```

Figure 2.1: Implementation of the Fibonacci function using `Domain`

2.2 Memory model

A simple and natural concurrency model is *sequential consistency* [Lamport, 1979]: every concurrent behavior corresponds to a sequential interleaving of the instructions of the different domains. OCaml 5 adopted a *relaxed memory model* [Dolan et al., 2018] that allows more behaviors. Programmers have to make sure their program is correct for every valid behavior.

In practice, a good mental model compatible with the operational semantics [Dolan et al., 2018] consists in assuming domains have *distinct views* of shared memory. When a domain modifies a shared data structure, the modification is not immediately observable by other domains. To make this modification public, the initial domain has to transfer its memory view using *synchronization mechanisms* (see Section 2.3).

In this thesis, we will not consider relaxed behaviors; we will assume a sequentially consistent memory model. This assumption is not realistic in the sense that proving an OCaml program is correct under a sequentially consistent memory model is not sufficient to prove it correct under the relaxed memory model. We discuss this limitation in Section 4.4. The main reason is that verifying a parallel scheduler like the one presented in Chapter 10 for a sequentially consistent memory model is already quite challenging.

2.3 Synchronization

To synchronize domains, OCaml 5 provides blocking and non-blocking mechanisms.

2.3.1 Blocking synchronization

The standard library provides basic blocking synchronization mechanisms: lock[7], semaphore[8], condition variable[9]. Their role is both to perform synchronization and control access to shared resources. For instance, a lock can be used to enforce mutual exclusion. We further describe and specify these mechanisms in Chapter 6.

2.3.2 Non-blocking synchronization

Non-blocking mechanisms only performs synchronization. They are crucial for implementing lock-free algorithms, as in Chapter 9.

[7] `https://ocaml.org/manual/5.3/api/Mutex.html`
[8] `https://ocaml.org/manual/5.3/api/Semaphore.html`
[9] `https://ocaml.org/manual/5.3/api/Condition.html`

```ocaml
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
  )

let rec pop t =
  match Atomic.get t with
  | [] ->
      None
  | v :: new_ as old ->
      if Atomic.compare_and_set t old new_ then (
        Some v
      ) else (
        Domain.cpu_relax () ;
        pop t
      )
```

Figure 2.2: Concurrent stack implementation

2.3.2.1 Atomic references

The only non-blocking synchronization mechanism available until version 5.4 is *atomic references*. According to the operational semantics, an atomic reference is a special memory location carrying both a physical value and a logical memory view.

The `Atomic`[10] standard module provides primitives to manipulate atomic references. `Atomic.make v` creates a new reference containing `v`. `Atomic.set r v` writes `v` to `r` and *releases* the memory view of the current domain. Symmetrically, `Atomic.get r` reads the content of `r` and *acquires* the memory view of the last writer domain.

Besides these three basic primitives, a few others are provided. `Atomic.exchange r v` writes `v` to `r` and returns the former value. `Atomic.fetch_and_add r n` atomically increments `r` by `n` and returns the former value. Last but not least, `Atomic.compare_and_set r v1 v2` atomically compares the content of `r` with `v1` and returns either `false` if the comparison fails or `true` if the comparison succeeds; in this case, it sets `r` to `v2`.

Using atomic references, we implement a lock-free concurrent stack [Treiber, 1986] in Figure 2.2. The data structure consists of an atomic reference to a list of values. The `push` and `pop` operations follow a pattern that is very common in lock-free programming: (1) collect information about the shared state; (2) decide whether to keep going or abort

[10] `https://ocaml.org/manual/5.3/api/Atomic.html`

```
type 'a t =
  'a atomic_loc

val get :
  'a t -> 'a
val set :
  'a t -> 'a -> unit

val exchange :
  'a t -> 'a -> 'a
val compare_and_set :
  'a t -> 'a -> 'a -> bool
val fetch_and_add :
  int t -> int -> int

val incr :
  int t -> unit
val decr :
  int t -> unit
```

Figure 2.3: `Stdlib.Atomic.Loc` interface

based on that information; (3) if the operation keeps going, locally prepare a new desired state; (4) try to commit the new state using `Atomic.compare_and_set`; (5) if it succeeds, the operation itself succeeds; (6) otherwise, the operation restarts. Interestingly, this pattern boils down to the notion of *atomic transaction*.

2.3.2.2 Atomic fields

Atomic references are enough to write concurrent algorithms — for example, they are sufficient for the memory-safe algorithms of the `Saturn` library[11]. However, atomic references introduce an indirection that can make the algorithm both more complex and less efficient. Consequently, to avoid this indirection, programmers mindful of performance use a low-level trick: the first field of a record can be made atomic by reinterpreting the record as an atomic reference using `Obj.magic`. This trick (currently) works because (1) the low-level representation of an atomic reference is the same as a one-field record and (2) the OCaml compiler does not reorder fields. It comes at the cost of readability, memory safety and possibly correctness — such tricks may violate assumptions made by the compiler. Besides, it is very limited: only one field can be made atomic.

This situation made both programming and verification more difficult. Together with Gabriel Scherer, we introduced *atomic record fields* in the language based on a design proposed by Basile Clément. It was integrated upstream in May 2025, to be included in the upcoming release of OCaml 5.4.

Declaring a record field as atomic simply requires an `[@atomic]` attribute — and could eventually become a proper keyword of the language. For example, atomic references can be redefined this way:

[11]We discuss memory safety in Chapter 12. Suffice it to say that some algorithms in `Saturn` are memory-unsafe because they use unsafe features of the language, *e.g.* `Obj.magic`.

```
type 'a atomic_ref =
  { mutable contents: 'a [@atomic]; }
```

As expected, the usual field accesses — *e.g.* `r.contents` and `r.contents <- v` — are performed atomically. However, expressing other atomic primitives is more tricky. Indeed, we would like to avoid adding a new language construct for each of them.

We introduced a built-in type `'a atomic_loc` representing an *atomic location* holding a value of type `'a`. Such locations are constructed using a syntax extension:

```
[%atomic.loc <expr>.<field>]
```

The new `Atomic.Loc` standard module, whose interface is given in Figure 2.3, is the counterpart of `Atomic` for atomic locations. For example, one can write:

```
Atomic.Loc.fetch_and_add [%atomic.loc r.contents] 1
```

Internally, a value of type `'a atomic_loc` is represented as a pair of a record and an integer offset for the desired field, and the `atomic.loc` extension builds this pair in a well-typed manner. When a primitive of the `Atomic.Loc` module is applied to an `atomic.loc` expression, the compiler can optimize away the construction of the pair.

2.3.2.3 Atomic arrays

We also implemented *atomic arrays*[12], another facility commonly requested by authors of efficient concurrent algorithms. More precisely, we introduced a new `Atomic.Array` module in the standard library; its interface is given in Figure 2.4. As of today, it only includes a few functions corresponding to the `Atomic` primitives. We plan to integrate the feature upstream.

Our implementation builds on top of the `'a Atomic.Loc.t` type and relies on two low-level primitives that we introduced in the compiler:

```
external unsafe_index
  : 'a Atomic.Array.t -> int -> 'a Atomic.Loc.t
  = "%atomic_unsafe_index"
external index
  : 'a Atomic.Array.t -> int -> 'a Atomic.Loc.t
  = "%atomic_index"
```

Given an atomic array and an index, the `index` function returns an atomic location corresponding to the element at the index, after performing a bound check. `unsafe_index` omits the bound check — additional performance at the cost of memory-safety — and allows to express the atomic counterpart of `Array.unsafe_get` and `Array.unsafe_set`. The primitives of the module `Atomic.Loc` can then be used directly on these atomic locations.

2.4 Third-party libraries

The naive implementation of the Fibonacci function of Figure 2.1 does not scale up because it spawns too many domains — potentially many more than the number of

[12] `https://github.com/clef-men/ocaml/tree/atomic_array`

```ocaml
type 'a t

val make :
  int -> 'a -> 'a t

val init :
  int -> (int -> 'a) -> 'a t

val length :
  'a t -> int

val unsafe_get :
  'a t -> int -> 'a
val get :
  'a t -> int -> 'a

val unsafe_set :
  'a t -> int -> 'a -> unit
val set :
  'a t -> int -> 'a -> unit

val unsafe_exchange :
  'a t -> int -> 'a -> 'a
val exchange :
  'a t -> int -> 'a -> 'a

val unsafe_compare_and_set :
  'a t -> int -> 'a -> 'a -> bool
val compare_and_set :
  'a t -> int -> 'a -> 'a -> bool

val unsafe_fetch_and_add :
  int t -> int -> int -> int
val fetch_and_add :
  int t -> int -> int -> int
```

Figure 2.4: `Stdlib.Atomic.Array` interface

```ocaml
module Task =
  Domainslib.Task

let num_domains =
  Domain.recommended_domain_count ()

let rec fibonacci pool n =
  if n <= 1 then
    1
  else
    let task1 = Task.async pool (fun () -> fibonacci pool (n - 1)) in
    let task2 = Task.async pool (fun () -> fibonacci pool (n - 2)) in
    Task.await pool task1 + Task.await pool task2

let fibonacci n =
  let pool = Task.setup_pool ~num_domains:(num_domains - 1) () in
  let res = Task.run pool (fun () -> fibonacci pool n) in
  Task.teardown_pool pool ;
  res
```

Figure 2.5: Implementation of the Fibonacci function using `Domainslib`

available cores. Not only does performance rapidly degrade but OCaml 5 also has a limit on the number of active domains.

To improve this implementation — and more generally to write parallel algorithms —, we would like a more flexible interface — a higher-level interface such that we would not have to manage domains ourselves. The fact is that OCaml 5 only provides low-level parallel primitives, leaving third-party libraries propose high-level abstractions. Since the release of OCaml 5, a number of such libraries have been developed.

`Domainslib` [Multicore OCaml development team, 2025], `Eio` [Madhavapeddy and Leonard, 2025], `Miou` [Calascibetta, 2025], `Moonpool` [Cruanes, 2025] and a few other libraries provide *task schedulers*. Internally, these schedulers manage a pool of domains. They also manage a set of tasks that are executed on the different domains of the pool. The ordering of the tasks and the way they can interact with the scheduler — especially via algebraic effects — depend on the library.

For example, using `Domainslib`, we can reimplement the Fibonacci function, as shown in Figure 2.5. The `Task.async` function spawns a new task and returns a *promise* that can be awaited using `Task.await` to get the result of the task. In Chapter 10, we implement and verify a similar scheduler.

These schedulers rely both on locking mechanisms and concurrent data structures to manage the domains and the tasks. The `Saturn` [Karvonen and Morel, 2025b] library provides a collection of standard lock-free data structures ready for use. For example, `Domainslib` relies on the work-stealing deque implemented in `Saturn`; `Moonpool` relies on a bounded version. In Chapter 9, we verify part of `Saturn`, including the work-stealing deque (bounded and unbounded).

In this growing ecosystem, other parallel abstractions have been developed. `Riot` [Ostera, 2025] provides a parallel scheduler based on the *actor model* [Hewitt et al., 1973]. `Kcas` [Karvonen, 2025a] provides a *software transactional memory* [Shavit and Touitou, 1995] implementation.

2.5 Future work

In this chapter, we presented two new language features that we introduced to better express concurrent algorithms: atomic record fields (Section 2.3.2.2) and atomic arrays (Section 2.3.2.3). Another useful feature that we were asked to address by concurrency experts is *record field cache line alignment*, which consists in providing programmers an idiomatic way to force a record field to be aligned to cache line size. Similarly, we should also support cache-line-aligned atomic arrays (each cell would be properly aligned). These features are crucial for performance[13], to avoid *false sharing*[14] — see, for example, the two highly efficient C++ concurrent queues developed by Rigtorp [2025a,b].

While the `Atomic`[15] module does provide a primitive `make_contended` to create a cache-line-aligned atomic reference, there is currently no counterpart for atomic fields and atomic arrays. Instead, as usual, programmers rely on unsafe expedients [Karvonen, 2025b].

[13] https://www.1024cores.net/home/lock-free-algorithms/first-things-first
[14] https://en.wikipedia.org/wiki/False_sharing
[15] https://ocaml.org/manual/5.3/api/Atomic.html

Chapter 3

Iris arsenal

Separation logic [O'Hearn et al., 2001; Reynolds, 2002; O'Hearn, 2007] is a logic of *resources*. It allows expressing both partial ownership — permission to access a resource — and full ownership — exclusive permission to access a resource — in a composable way. For example, the *fractional* [Boyland, 2003] *points-to* assertion $\ell \overset{q}{\mapsto} v$ gives read-only permission while the *full points-to* assertion $\ell \mapsto v$ gives read and write permission. Crucially, assertions are *stable under interference*: as long as a thread holds $\ell \overset{q}{\mapsto} v$, ℓ points to v no matter what other threads do. *Separating conjunction* $P_1 * P_2$ combines *disjoint* resources P_1 and P_2. For example, $\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2$ implies ℓ_1 and ℓ_2 are distinct. It enables *local reasoning*, as illustrated by the *frame rule*:

$$\frac{\{\, P \,\}\ e\ \{\, v.\, Q \,\}}{\{\, P * R \,\}\ e\ \{\, v.\, Q * R \,\}}$$

From top to bottom, this rules says that we can always add resources to a specification; they are unaffected by the program. From bottom to top, it says we can forget about some resources to specify a program, focusing on strictly needed resources.

In this thesis, we use Iris [Jung et al., 2018b], a state-of-the-art separation logic. Iris has been fully mechanized [Krebbers et al., 2018] in the Rocq proof assistant. It is currently the most advanced logic for verifying fined-grained concurrent algorithms, thanks to flexible and powerful mechanisms. In this chapter, we present most of the mechanisms we need except *prophecy variables*, that we describe in Chapter 5.

3.1 User-defined higher-order ghost state

One of the most important features of Iris is its *user-defined higher-order ghost state*, a very flexible form of ghost state.

Ghost state, as opposed to *physical state*, is a formal technique that consists in introducing purely logical resources in order to verify a program. For instance, when verifying a fine-grained concurrent data structure, it is often the case that the physical state of the data structure does not determine the logical state; in other words, many distinct logical states may correspond to the same physical state. In this case, ghost state can be used to keep track of the logical state throughout the execution.

Iris offers *higher-order* ghost state. This means ghost state may refer to (with a restriction) and therefore depends on the type of Iris propositions. Naturally, Iris propositions depends on ghost state. Consequently, ghost state and propositions are defined in a mutually recursive way. This feature is crucial for defining *invariants* (see Section 3.6) and verifying complex concurrent algorithms, especially those with *external linearization points* [Dongol and Derrick, 2014].

Iris also offers *user-defined* ghost state. This means the user of the logic can introduce new types of resources according to his needs. More precisely, the (base) logic is parametrized by a user-supplied *resource algebra*. In this thesis, in particular in Chapter 9, we heavily rely on this feature.

3.2 Ghost update

During the proof, we need to update the ghost state. In Iris, this is performed primarily using the *basic update modality*: $\dot{\Rrightarrow} P$ means P holds after a ghost state update. These updates are purely logical; they are not visible in the program.

3.3 Persistent assertion

In Iris, assertions are affine: using a resource consumes it, removes it from the proof context. Some assertions, however, are *persistent*. Once a persistent assertion holds, it holds forever; using it does not consume it. This enables *duplication* ($P \vdash P * P$) and *sharing*. In particular, pure (meta-level) assertions embedded into the logic are persistent.

Formally, persistence is defined in terms of the *persistence modality*:

$$\mathsf{persistent}\ P \triangleq P \vdash \square\, P$$

Informally, $\square\, P$ means P holds without asserting any exclusive ownership; in other words, it only expresses knowledge. Naturally, $\square\, P$ is persistent.

3.4 Sequential specification

In this thesis, we use two kinds of specifications: *sequential specifications* — described in this section — and *atomic specifications* — described in Section 3.7. Sequential specifications take the form of Hoare triples:

$$\{\, P \,\}\ e\ \{\, \Phi \,\}$$

where P is an Iris assertion, e an expression and Φ an Iris predicate over values[1].

Informally, this triple says: if the precondition P holds, we can safely execute e and, if the execution terminates, the returned value satisfies the postcondition Φ. It is a persistent resource, allowing executing e many times.

Most of the time, we will present sequential specifications in a more spacious way, like in the following example:

$$\begin{array}{c}\textsc{stack-push-spec-seq}\\ \mathsf{stack\text{-}model}\ t\ vs \\ \hline \mathtt{stack\_push}\ t\ v \\ \hline \mathtt{()}.\ \mathsf{stack\text{-}model}\ t\ (v :: vs)\end{array}$$

This specification says that, given the ownership of stack t — represented by $\mathsf{stack\text{-}model}\ t\ vs$ —, $\mathtt{stack\_push}\ t\ v$, if it terminates, pushes v onto the stack and returns the unit value.

[1]We define expressions and values in Chapter 4.

3.5 Weakest precondition

Hoare triples are defined as follows:

$$\{P\}\ e\ \{\Phi\} \triangleq \square(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\})$$

As in Section 3.3, the persistence modality is responsible for making the resource persistent.

The *weakest precondition* resource $\mathsf{wp}\ e\ \{\Phi\}$ is not persistent: contrary to Hoare triples, it can depend on exclusive ownership. Informally, it says that: once only, we can execute e and, if the execution terminates, the returned value satisfies the postcondition Φ.

In practice, we use weakest precondition to specify higher-order functions: functions that take functions as arguments. Indeed, higher-order functions typically run the functions they are given only once, or once per element in the case of iterators. Therefore, requiring a Hoare triple is often stronger than necessary; requiring a weakest precondition may be enough.

3.6 Invariant

To share exclusive resources between threads, Iris provides a special mechanism: *invariants*. The proposition $\boxed{P}^{\iota}$ represents an invariant containing proposition P and annotated with *namespace* ι. As we will see, this namespace prevents reentrancy (accessing the same invariant twice); when it is the *full mask* $\top$, we may omit it. Informally, $\boxed{P}^{\iota}$ states that P holds at each step of the program execution. Crucially, invariants are persistent, so they can be shared.

Invariants can be allocated using the following rule:

$$\frac{\begin{array}{@{}l@{}}\text{WP-INV-ALLOC}\\ P \qquad \boxed{P}^{\iota} \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\}\end{array}}{\mathsf{wp}\ e\ \{\Phi\}}$$

They can only be accessed *atomically* (during a single execution step), as shown by the rule:

$$\frac{\begin{array}{@{}l@{}}\text{WP-INV-ACCESS}\\ \mathsf{atomic}\ e \qquad \boxed{P}^{\iota} \qquad \iota \subseteq \mathcal{E} \qquad \triangleright P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\iota}\ e\ \{v.\ \triangleright P * \Phi\ v\}\end{array}}{\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}}$$

As e is *atomic* (reduces to a value in a single step), it is safe to access the invariant. Accessing $\boxed{P}^{\iota}$ means: (1) opening the invariant, that is acquiring P and marking ι (removing it from mask $\mathcal{E}$); (2) closing it after executing e, that is giving P back in the postcondition of the weakest precondition.

Importantly, for soundness reasons, P is weakened using the *later modality* $\triangleright P$. While $P \vdash \triangleright P$ always holds, $\triangleright P \vdash P$ does not always hold. The usual way of getting rid of a later modality is to take a step in the program, *e.g.* reduce e.

In general, weakest preconditions $\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}$ are annotated with a *mask* $\mathcal{E}$ to keep track of opened invariants. When this mask is the full mask, meaning no invariants were opened, we may omit the annotation, as in Section 3.5.

Most of the time, WP-INV-ACCESS is the only rule needed to interact with an invariant during the proof. However, it is possible and sometimes necessary to access an invariant in a purely logical way, without actually taking a step. This is expressed by the following rule:

$$\text{INV-ACCESS}\quad \frac{\boxed{P}^{\iota} \qquad \iota \subseteq \mathcal{E}}{{}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\setminus\iota} (\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\iota}{\Rrightarrow}^{\mathcal{E}} \mathsf{True}))}$$

This rule is very similar to WP-INV-ACCESS except there is no weakest precondition and therefore no backing atomic expression. It relies on the *fancy update modality* ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$, which subsumes the basic update modality of Section 3.2. The masks $\mathcal{E}_1$ and $\mathcal{E}_2$ represent the enabled invariants respectively "outside" and "inside" the fancy update; when they are both the full mask, we may omit them. To produce this modality, one can use the following rules:

$$\text{FUPD-WP}\quad \frac{\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}}{\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}} \qquad \text{WP-FUPD}\quad \frac{\mathsf{wp}_{\mathcal{E}}\ e\ \{v.\ \Rrightarrow_{\mathcal{E}} \Phi\ v\}}{\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}} \qquad \text{FUPD-TRANS}\quad \frac{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\ {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3} P}{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_3} P} \qquad \text{FUPD-WAND}\quad \frac{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P \qquad P \mathrel{-\!\!*} Q}{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} Q}$$

where $\Rrightarrow_{\mathcal{E}} P \triangleq {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$.

3.7 Atomic specification

In the sequential specification STACK-PUSH-SPEC-SEQ of Section 3.4, the operation is given the exclusive ownership of the stack, which lets it update the data structure without interference from other threads. For a concurrent stack — and more generally for a concurrent data structure —, however, things get more complicated.

Indeed, requiring exclusive ownership of the stack would inhibit concurrency. Thus, we typically introduce a persistent predicate that we call the *invariant* of the data structure — not to be confused with Iris invariants. This invariant contains the resources shared by the different threads.

Having an invariant enables concurrency but does not say how the data structure is updated. To specify concurrent operations, we use the notion of *logical atomicity* [da Rocha Pinto et al., 2014]. An operation is said to be logically atomic if it appears to take effect atomically at some point during its execution; this point is called the *linearization point* of the operation. Birkedal et al. showed that this notion implies *linearizability* [Herlihy and Wing, 1990] in a sequentially consistent memory model.

In Iris, logical atomicity takes the form of *atomic specifications*:

$$\{ P_{priv} \}\ \langle \overline{x}.\ P_{pub} \rangle\ e\ ⨾\ \mathcal{E}\ \langle \overline{y}.\ Q \rangle\ \{ \Phi \}$$

P_{priv} and Φ are standard *private* pre- and postcondition for the user of the specification, similarly to Hoare triples. P_{pub} and Q are *public* pre- and postcondition; they specify the linearization point of the operation. Quantifiers $\overline{x}$ represent the *demonic nature* of P_{pub}: the exact state at the linearization point, given by P_{pub}, is unknown until it happens. Quantifiers $\overline{y}$ represent the *angelic nature* of Q: at the linearization point, the operation can choose how to instantiate the new state Q. Mask $\mathcal{E}$ represents the opened invariants at the linearization point.

In sum, the atomic specification says: if the private precondition P_{priv} holds, we can safely execute e and, if the execution terminates, (1) the returned value satisfies the private postcondition Φ and (2) at some point during the execution, the state was atomically updated from P_{pub} to Q.

Most of the time, we will present atomic specifications in a more spacious way, like in the following example:

STACK-PUSH-SPEC-ATOMIC

$$\begin{array}{c} \mathsf{stack\text{-}inv}\ t\ \iota \\ \hdashline vs.\ \mathsf{stack\text{-}model}\ t\ vs \\ \hline \texttt{stack\_push}\ t\ v\ ⨟\ \iota \\ \hline \mathsf{stack\text{-}model}\ t\ (v :: vs) \\ \hdashline \texttt{()}.\ \mathsf{True} \end{array}$$

This specification says that, given a concurrent stack t, `stack_push` $t\ v$, if it terminates, atomically updates the logical content of t from some values vs to $v :: vs$ and returns the unit value.

3.8 Atomic update

Atomic specifications are defined as follows:

$$\{\, P_{priv} \,\}\ \langle\, \overline{x}.\, P_{pub} \,\rangle\ e\ ⨟\ \mathcal{E}\ \langle\, \overline{y}.\, Q \,\rangle\ \{\, \Phi \,\} \ \triangleq\ \begin{array}{l} \forall\, \Psi. \\ P_{priv} \twoheadrightarrow \\ \langle\, \overline{x}.\, P_{pub} \mid \overline{y}.\, Q \Rrightarrow \Phi\ \overline{x}\ \overline{y} \twoheadrightarrow \Psi\ \overline{x}\ \overline{y} \,\rangle_{\mathcal{E}} \twoheadrightarrow \\ \mathsf{wp}\ e\ \big\{\, \Psi \,\big\} \end{array}$$

Similarly to Hoare triples, it relies on the weakest precondition notion. More precisely, it requires to prove $\mathsf{wp}\ e\ \big\{\, \Psi \,\big\}$ for *any* Ψ under two hypotheses: (1) the private precondition P_{priv}, (2) an *atomic update* of the form $\langle\, \overline{x}.\, P_{pub} \mid \overline{y}.\, Q \Rrightarrow P \,\rangle_{\mathcal{E}}$ where $P \triangleq \Phi\ \overline{x}\ \overline{y} \twoheadrightarrow \Psi\ \overline{x}\ \overline{y}$. Crucially, as Ψ is universally quantified, the only way to prove $\mathsf{wp}\ e\ \big\{\, \Psi \,\big\}$ is to use this atomic update.

Essentially, an atomic update is the Iris reification of a linearization point. For example, the atomic update for STACK-PUSH-SPEC-ATOMIC corresponds to:

$$\langle\, vs.\, \mathsf{stack\text{-}model}\ t\ vs \mid \mathsf{stack\text{-}model}\ t\ (v :: vs) \Rrightarrow \mathsf{True} \,\rangle_{\iota}$$

As a first approximation, an atomic update behaves like an invariant, in the sense that it can be atomically accessed. However, contrary to invariants, there are two ways to close an atomic update after opening it, as shown by the rule:

AU-ACCESS

$$\frac{\langle\, \overline{x}.\, P_{pub} \mid \overline{y}.\, Q \Rrightarrow P \,\rangle_{\mathcal{E}}}{{}^{\top\setminus\mathcal{E}}{\Rrightarrow}^{\emptyset}\ \exists\, \overline{x}.\, P_{pub} * \Big(\big(P_{pub} \twoheadrightarrow {}^{\emptyset}{\Rrightarrow}^{\top\setminus\mathcal{E}}\ \langle\, \overline{x}.\, P_{pub} \mid \overline{y}.\, Q \Rrightarrow P \,\rangle_{\mathcal{E}}\big) \wedge \big(\forall\, \overline{y}.\, Q \twoheadrightarrow {}^{\emptyset}{\Rrightarrow}^{\top\setminus\mathcal{E}}\ P\big)\Big)}$$

Opening the atomic update yields P_{pub} for some $\overline{x}$ along with a conjunction representing two ways of closing the update. (1) We can abort: we give back P_{pub} and retrieve the atomic update. (2) We can commit: we choose $\overline{y}$ and exchange the public postcondition Q for the postcondition P. This mechanism can be compared to the programming pattern mentioned in Section 2.3.2.1. It is basically a logical retry loop for reasoning about atomic transactions.

Chapter 4

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

HeapLang. In the Iris literature, most works on the verification of concurrent algorithms [Jung et al., 2020; Vindum and Birkedal, 2021; Vindum et al., 2022; Carrott, 2022; Carbonneaux et al., 2022; Mulder et al., 2022; Mulder and Krebbers, 2023; Jung et al., 2023; Somers and Krebbers, 2024; Krishna et al., 2020; Patel et al., 2021, 2024; Lee et al., 2025] rely on HeapLang [Iris development team, 2025a], the exemplar Iris language. HeapLang is a concurrent, imperative, untyped, call-by-value functional language. To the best of our knowledge, it is currently the closest language[1] to OCaml 5 in the Iris ecosystem.

We started our verification effort in HeapLang, but it eventually proved impractical to verify realistic OCaml libraries. Indeed, it lacks basic abstractions such as algebraic data types (tuples, mutable and immutable records, variants) and mutually recursive functions. Consequently, verifying OCaml programs in HeapLang requires non-trivial translation choices and introduces various encodings, to the point that the relation between the source and verified programs can become difficult to maintain and reason about. It also has very few standard data structures that can be directly reused. These limitations are well-known in the Iris community.

Physical equality. Another, maybe less obvious, shortcoming of HeapLang is the soundness of its semantics with respect to OCaml, in other words how faithful it is to the original language. One ubiquitous — particularly in lock-free algorithms relying on low-level atomic primitives — and subtle point is *physical equality*. In Section 4.2.3, we show that (1) HeapLang's semantics for physical equality is not compatible with OCaml and (2) OCaml's informal semantics is actually too imprecise to verify basic concurrent algorithms. To remedy this, we propose a new formal semantics for physical equality and structural equality. We hope this work will influence the way these notions are specified in OCaml.

ZooLang. We developed a more practical OCaml-like verification language: ZooLang. This language consists of a subset of OCaml 5 equipped with a formal semantics and an Iris-based program logic. This subset includes the basic abstractions mentioned above as well as atomic record fields (see Section 2.3.2.2). Following a pragmatic approach,

[1]The recent Osiris [Seassau et al., 2025] language targets a subset of OCaml. However, it does not support parallelism and its practicality remains to be shown. See Section 4.3 for further discussion.

```
project                                       theories
├── dune-project                              ├── domainslib
└── lib                                       │   ├── scheduler__code.v
    ├── domainslib                            │   ├── scheduler__opaque.v
    │   ├── dune                    ⟹         │   └── scheduler__types.v
    │   ├── scheduler.ml                      └── saturn
    │   └── scheduler.mli                         ├── queue__code.v
    └── saturn                                    ├── queue__opaque.v
        ├── dune                                  └── queue__types.v
        ├── queue.ml
        └── queue.mli

$ ocaml2zoo project theories
```

Figure 4.1: Translation of a Dune project using `ocaml2zoo`

we added new features as we applied it to more verification scenarios. ZooLang is fully mechanized in Rocq.

Zoo. We were influenced by the Perennial framework [Chajed et al., 2019, 2021, 2022; Chang et al., 2023], which achieved similar goals for the Go language with a focus on crash-safety. As in Perennial, we also provide a translator from (a subset of) OCaml to ZooLang: `ocaml2zoo`. We call the resulting framework Zoo.

4.1 Zoo in practice

In this section, we give an overview of the framework. We provide a minimal example[2] demonstrating its use.

To verify an OCaml library, the first thing to do is to run `ocaml2zoo`, which translates OCaml source files[3] into Rocq files containing their ZooLang representation. This tool can process entire Dune projects. Moreover, external OCaml dependencies are supported; it is up to the user to provide their verified Rocq version, either in the current Rocq project or through Rocq dependencies.

For example, Figure 4.1 shows the translation of a simple project with two libraries. Each `.ml` source file gives three Rocq files corresponding to (1) ZooLang types, (2) ZooLang code and (3) instructions for opacifying the code once it is verified, acting like a basic abstraction barrier. Assuming `queue.ml` implements a concurrent stack as in Figure 2.2, the file `queue__code.v` generated by `ocaml2zoo` contains the ZooLang representation given in Figure 4.2 — we postpone the description of ZooLang to Section 4.2.

Once this translation is done, the user can specify and verify the generated code in Rocq, using the full Iris arsenal presented in Chapter 3. For example, the specification STACK-PUSH-SPEC-ATOMIC corresponds to the Rocq lemma of Figure 4.3.

[2] `https://github.com/clef-men/zoo-demo`

[3] Actually, `ocaml2zoo` processes binary annotation files (`.cmt` files).

```
Definition stack_create : val :=
  fun: <> =>
    ref [].

Definition stack_push : val :=
  rec: "push" "t" "v" =>
    let: "old" := !"t" in
    let: "new_" := "v" :: "old" in
    if: ~ CAS "t".[contents] "old" "new_" then (
      domain_yield () ;;
      "push" "t" "v"
    ).

Definition stack_pop : val :=
  rec: "pop" "t" =>
    match: !"t" with
    | [] =>
        §None
    | "v" :: "new_" as "old" =>
        if: CAS "t".[contents] "old" "new_" then (
          ‘Some( "v" )
        ) else (
          domain_yield () ;;
          "pop" "t"
        )
    end.
```

Figure 4.2: ZooLang translation of the concurrent stack of Figure 2.2 as generated by `ocaml2zoo`

```
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.
```

Figure 4.3: Rocq lemma corresponding to STACK-PUSH-SPEC-ATOMIC

4.2 ZooLang

We now present the ZooLang language: its low-level syntax (Section 4.2.1), high-level syntax (Section 4.2.2), semantics (Section 4.2.5) and derived Iris program logic.

4.2.1 Low-level syntax

The low-level ZooLang syntax is displayed in Figure 4.4. Users do not usually interact directly with it; the high-level syntax (see Section 4.2.2) is more convenient.

Literals. Literals are be divided into two categories: source literals and runtime literals; the former can appear in source code while the latter are produced during execution. A source literal is either a boolean or an unbounded integer. A runtime literal is either a memory location, a prophecy variable [Jung et al., 2020] or poison; prophecy variables are represented using logical identifiers that will play a crucial role in the semantics.

Values. A value is either a literal, a function or an immutable block. A function consists of a list of recursive functions and the index of the function in this list.

An immutable block consists of a tag and a list of value fields; it is annotated with a *generativity* whose meaning will be explained in Section 4.2.3.

Expressions. An expression is either a value, a named variable, a (possibly recursive) function, an application, a let-binding, a unary operator, a binary operator, a physical comparison, a conditional, a for-loop, an allocation, a block, a (shallow) match, a block tag read, a block size read, a memory read, a memory write, an atomic exchange, an atomic compare-and-set, an atomic fetch-and-add, a fork, a domain-local read, a domain-local write, a prophecy variable, a prophecy resolution.

Similarly to values, a block consists of a tag and a list of expression fields; it is annotated with a *mutability* whose meaning will also be explained in Section 4.2.3.

A match expression consists of a list of regular branches and a fallback branch (a binder and an expression body). A match branch consists of a shallow pattern and an expression body; supporting deep patterns is left for future work — we never needed them in practice.

Coercions. For convenience, we use the following coercions: $\mathsf{LitBool}$, LitInt, LitLoc, $\mathsf{LitProph}$, Val, ValLit.

Syntactic sugar. For convenience again, we define some syntactic sugar displayed at the bottom of Figure 4.4.

4.2.2 High-level syntax

On top of the low-level syntax, we define the high-level syntax of Figure 4.5 using Rocq notations, omitting mutually recursive toplevel functions that are treated specially. This is the surface syntax as it appears in Rocq. Overall, it should look familiar to the OCaml programmer — as least, it is meant to.

Expressions include standard constructs like booleans, integers, (possibly recursive) anonymous functions, applications, let-bindings, sequences, unary and binary operators,

boolean	b	$\in$	$\mathbb{B}$
integer	n	$\in$	$\mathbb{Z}$
location	ℓ		
prophet identifier	pid		
block identifier	bid		
tag	tag	$\in$	$\mathbb{N}$
index	i	$\in$	$\mathbb{N}$
identifier	x	$\in$	String
binder	bdr, f	$::=$	$\mathsf{BinderAnon} \mid \mathsf{BinderNamed}\ x$
unary operator	$unop$	$::=$	$\mathsf{UnopNeg} \mid \mathsf{UnopMinus} \mid \mathsf{UnopIsImmediate}$
binary operator	$binop$	$::=$	$\mathsf{BinopPlus} \mid \mathsf{BinopMinus}$
		$\mid$	$\mathsf{BinopMult} \mid \mathsf{BinopQuot} \mid \mathsf{BinopRem}$
		$\mid$	$\mathsf{BinopLand} \mid \mathsf{BinopLor} \mid \mathsf{BinopLsl} \mid \mathsf{BinopLsr}$
		$\mid$	$\mathsf{BinopLe} \mid \mathsf{BinopLt} \mid \mathsf{BinopGe} \mid \mathsf{BinopGt}$
mutability	mut	$::=$	$\mathsf{Mutable}$
		$\mid$	$\mathsf{ImmutableNongenerative}$
		$\mid$	$\mathsf{ImmutableGenerativeWeak}$
		$\mid$	$\mathsf{ImmutableGenerativeStrong}$
generativity	gen	$::=$	$\mathsf{Generative}\ bid^{?} \mid \mathsf{Nongenerative}$
literal	lit	$::=$	$\mathsf{LitBool}\ b \mid \mathsf{LitInt}\ n \mid \mathsf{LitLoc}\ \ell$
		$\mid$	$\mathsf{LitProph}\ pid \mid \mathsf{LitPoison}$
value	v	$::=$	$\mathsf{ValLit}\ lit$
		$\mid$	$\mathsf{ValRecs}\ i\ \overline{rec}$
		$\mid$	$\mathsf{ValBlock}\ gen\ tag\ \overline{v}$
expression	e	$::=$	$\mathsf{Val}\ v \mid \mathsf{Var}\ x$
		$\mid$	$\mathsf{Rec}\ f\ bdr\ e \mid \mathsf{App}\ e_1\ e_2 \mid \mathsf{Let}\ bdr\ e_1\ e_2$
		$\mid$	$\mathsf{Unop}\ unop\ e \mid \mathsf{Binop}\ binop\ e_1\ e_2 \mid \mathsf{Equal}\ e_1\ e_2$
		$\mid$	$\mathsf{If}\ e_0\ e_1\ e_2 \mid \mathsf{For}\ e_1\ e_2\ e_3$
		$\mid$	$\mathsf{Alloc}\ e_1\ e_2 \mid \mathsf{Block}\ mut\ tag\ \overline{e}$
		$\mid$	$\mathsf{Match}\ e\ bdr_{fb}\ e_{fb}\ \overline{br}$
		$\mid$	$\mathsf{GetTag}\ e \mid \mathsf{GetSize}\ e$
		$\mid$	$\mathsf{Load}\ e_1\ e_2 \mid \mathsf{Store}\ e_1\ e_2\ e_3$
		$\mid$	$\mathsf{Xchg}\ e_1\ e_2 \mid \mathsf{CAS}\ e_0\ e_1\ e_2 \mid \mathsf{FAA}\ e_1\ e_2$
		$\mid$	$\mathsf{Fork}\ e$
		$\mid$	$\mathsf{GetLocal} \mid \mathsf{SetLocal}\ e$
		$\mid$	$\mathsf{Proph} \mid \mathsf{Resolve}\ e_0\ e_1\ e_2$
recursive definition	rec	$::=$	$\{\, \mathsf{fun}\text{: } f;\ \mathsf{param}\text{: } bdr;\ \mathsf{body}\text{: } e \,\}$
pattern	pat	$::=$	$\{\, \mathsf{tag}\text{: } tag;\ \mathsf{fields}\text{: } \overline{bdr};\ \mathsf{as}\text{: } bdr \,\}$
branch	br	$::=$	$\{\, \mathsf{pat}\text{: } pat;\ \mathsf{expr}\text{: } e \,\}$
	$\mathsf{Seq}\ e_1\ e_2$	$\triangleq$	$\mathsf{Let}\ \mathsf{BinderAnon}\ e_1\ e_2$
	$\mathsf{Tuple}\ \overline{e}$	$\triangleq$	$\mathsf{Block}\ \mathsf{Nongenerative}\ 0\ \overline{e}$
	$\mathsf{Proj}_i\ e$	$\triangleq$	$\mathsf{Load}\ e\ (\mathsf{Val}\ (\mathsf{ValLit}\ (\mathsf{LitInt}\ i)))$
	$\mathsf{ValFun}\ bdr\ e$	$\triangleq$	$\mathsf{ValRecs}\ 0\ [\{\, \mathsf{fun}\text{: } \mathsf{BinderAnon};\ \mathsf{param}\text{: } bdr;\ \mathsf{body}\text{: } e \,\}]$
	$\mathsf{ValTuple}\ \overline{v}$	$\triangleq$	$\mathsf{ValBlock}\ \mathsf{Nongenerative}\ 0\ \overline{v}$
	$\mathsf{ValUnit}$	$\triangleq$	$\mathsf{ValTuple}\ []$

Figure 4.4: Low-level syntax

$$
\begin{array}{llll}
\text{Rocq term} & t & & \\
\text{constructor} & C & & \\
\text{projection} & \mathit{proj} & & \\
\text{record field} & \mathit{fld} & & \\
\text{identifier} & s, f & \in & \text{String} \\
\text{boolean} & b & \in & \mathbb{B} \\
\text{integer} & n & \in & \mathbb{Z} \\
\text{binder} & x & ::= & \texttt{<>} \mid s \\
\text{unary operator} & \oplus & ::= & \texttt{\textasciitilde} \mid \texttt{-} \\
\text{binary operator} & \otimes & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{`quot`} \mid \texttt{`rem`} \mid \texttt{`land`} \mid \texttt{`lor`} \mid \texttt{`lsl`} \mid \texttt{`lsr`} \\
 & & \mid & \texttt{<} \mid \texttt{$\leq$} \mid \texttt{>} \mid \texttt{$\geq$} \mid \texttt{=} \mid \texttt{$\neq$} \mid \texttt{==} \mid \texttt{!=} \\
 & & \mid & \texttt{and} \mid \texttt{or} \\
\text{toplevel value} & v & ::= & t \mid b \mid n \\
 & & \mid & \texttt{fun:}\ x_1 \dots x_n\ \texttt{=>}\ e \mid \texttt{rec:}\ f\ x_1 \dots x_n\ \texttt{=>}\ e \\
 & & \mid & \S C \mid \texttt{`}C\ \texttt{(}v_1\texttt{,} \dots \texttt{,}v_n\texttt{)} \mid \texttt{(}v_1\texttt{,} \dots \texttt{,}v_n\texttt{)} \\
 & & \mid & \texttt{[]} \mid v_1\ \texttt{::}\ v_2 \\
\text{expression} & e & ::= & t \mid s \mid b \mid n \\
 & & \mid & \texttt{fun:}\ x_1 \dots x_n\ \texttt{=>}\ e \mid \texttt{rec:}\ f\ x_1 \dots x_n\ \texttt{=>}\ e \mid e_1\ e_2 \\
 & & \mid & \texttt{let:}\ x\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid e_1\ \texttt{;;}\ e_2 \\
 & & \mid & \texttt{let:}\ f\ x_1 \dots x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{letrec:}\ f\ x_1 \dots x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \\
 & & \mid & \texttt{let:}\ \texttt{`}C\ x_1 \dots x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{let:}\ x_1\texttt{,} \dots \texttt{,}x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \\
 & & \mid & \oplus\, e \mid e_1 \otimes e_2 \\
 & & \mid & \texttt{if:}\ e_0\ \texttt{then}\ e_1\ (\texttt{else}\ e_2)^? \\
 & & \mid & \texttt{for:}\ x\ \texttt{:=}\ e_1\ \texttt{to}\ e_2\ \texttt{begin}\ e_3\ \texttt{end} \\
 & & \mid & \S C \mid \texttt{`}C\ \texttt{(}e_1\texttt{,} \dots \texttt{,}e_n\texttt{)} \mid \texttt{(}e_1\texttt{,} \dots \texttt{,}e_n\texttt{)} \mid e\texttt{.<}\mathit{proj}\texttt{>} \\
 & & \mid & \texttt{[]} \mid e_1\ \texttt{::}\ e_2 \\
 & & \mid & \texttt{`}C\ \texttt{\{}e_1\texttt{,} \dots \texttt{,}e_n\texttt{\}} \mid \texttt{\{}e_1\texttt{,} \dots \texttt{,}e_n\texttt{\}} \mid e\texttt{.\{}\mathit{fld}\texttt{\}} \mid e_1\ \texttt{<-\{}\mathit{fld}\texttt{\}}\ e_2 \\
 & & \mid & \texttt{ref}\ e \mid \texttt{!}\,e \mid e_1\ \texttt{<-}\ e_2 \\
 & & \mid & \texttt{match:}\ e_0\ \texttt{with}\ br_1 \texttt{|} \dots \texttt{|}\ br_n\ (\texttt{|\_}\ (\texttt{as}\ s)^?\ \texttt{=>}\ e)^?\ \texttt{end} \\
 & & \mid & e\texttt{.[}\mathit{fld}\texttt{]} \mid \texttt{Xchg}\ e_1\ e_2 \mid \texttt{CAS}\ e_1\ e_2\ e_3 \mid \texttt{FAA}\ e_1\ e_2 \\
 & & \mid & \texttt{Proph} \mid \texttt{Resolve}\ e_0\ e_1\ e_2 \\
\text{branch} & br & ::= & C\ (x_1 \dots x_n)^?\ (\texttt{as}\ s)^?\ \texttt{=>}\ e \\
 & & \mid & \texttt{[]}\ (\texttt{as}\ s)^?\ \texttt{=>}\ e \mid x_1\ \texttt{::}\ x_2\ (\texttt{as}\ s)^?\ \texttt{=>}\ e
\end{array}
$$

Figure 4.5: High-level syntax (omitting mutually recursive toplevel functions)

conditionals, for-loops, tuples. In any expression, one can refer to a Rocq term representing a ZooLang value (of type `val`) using its Rocq identifier. ZooLang is deeply embedded: variables are quoted as strings.

Data constructors are supported through two constructs: §C represents a constant constructor (*e.g.* `§None`) while ‘C `(`e_1`,` ... `,`e_n`)` represents a non-constant constructor (*e.g.* `‘Some (`e`)`). Unlike OCaml, ZooLang has projections of the form e`.`<*proj*> (*e.g.* `(`e_1`,`e_2`).<1>`) that can be used to obtain a specific component of a tuple or data constructor.

Mutable memory blocks are constructed using either the untagged record syntax `{`e_1`,` ... `,`e_n`}` or the tagged record syntax ‘C `{`e_1`,` ... `,`e_n`}`. Reading a record field can be performed using e`.{`*fld*`}` and writing to a record field using e_1 `<-{`*fld*`}` e_2. References are also supported through the usual constructs: `ref` e creates a reference, `!`e reads a reference and e_1 `<-` e_2 writes into a reference.

Algebraic data types. To simulate variants and records, we designed a machinery to define constructors, projections and record fields. For example, one may define a list-like type with:

```
Notation "'Nil'"  := (in_type "t" 0) (in custom zoo_tag).
Notation "'Cons'" := (in_type "t" 1) (in custom zoo_tag).
```

Users do not need to write this incantation directly, as it is generated by `ocaml2zoo` from the OCaml type declaration. Suffice it to say that it introduces the two tags in the `zoo_tag` custom entry, on which the notations for data constructors rely. The `in_type` term is needed to distinguish the tags of distinct data types; crucially, it cannot be simplified away by Rocq, as this could lead to confusion during the reduction of expressions.

Given this incantation, one may directly use the tags `Nil` and `Cons` in data constructors using the corresponding ZooLang constructs:

```
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.
```

Similarly, one may define a record-like type with two mutable fields `f1` and `f2`:

```
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".
```

Mutually recursive functions. ZooLang supports non-recursive and recursive functions but only *toplevel* mutually recursive functions. It is non-trivial to properly handle mutual recursion: when applying a mutually recursive function, a naive approach would replace calls to sibling functions by their respective bodies, but this typically makes the resulting expression unreadable. To prevent it, the mutually recursive functions have to know one another to preserve their names during β-reduction. We simulate this using some boilerplate that can be generated by `ocaml2zoo`. For example, one may define two mutually recursive functions `f` and `g` as follows:

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

(* boilerplate *)
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.
```

Concurrent primitives. ZooLang supports concurrent primitives both on atomic references (see Section 2.3.2.1) and atomic record fields (see Section 2.3.2.2) according to the table below. The OCaml expressions listed in the left-hand column translate into the ZooLang expressions in the right-hand column. Notice that an atomic location `[%atomic.loc` e`.`fld`]` (of type `_ Atomic.Loc.t`) translates directly into e`.[`fld`]`.

OCaml	Zoo
`Atomic.get` e	`!`e
`Atomic.set` e_1 e_2	e_1 `<-` e_2
`Atomic.exchange` e_1 e_2	`Xchg` e_1`.[contents]` e_2
`Atomic.compare_and_set` e_1 e_2 e_3	`CAS` e_1`.[contents]` e_2 e_3
`Atomic.fetch_and_add` e_1 e_2	`FAA` e_1`.[contents]` e_2
`Atomic.Loc.exchange [%atomic.loc` e_1`.`fld`]` e_2	`Xchg` e_1`.[`fld`]` e_2
`Atomic.Loc.compare_and_set [%atomic.loc` e_1`.`fld`]` e_2 e_3	`CAS` e_1`.[`fld`]` e_2 e_3
`Atomic.Loc.fetch_and_add [%atomic.loc` e_1`.`fld`]` e_2	`FAA` e_1`.[`fld`]` e_2

One important aspect of this translation is that atomic accesses (`Atomic.get` and `Atomic.set`) correspond to plain reads and writes. This is because we are working in a sequentially consistent memory model: there is no difference between atomic and non-atomic memory locations.

4.2.3 Physical equality

The notion of *physical equality* is ubiquitous in fine-grained concurrent algorithms. It appears not only in the semantics of the (==) operator, but also in the semantics of the `Atomic.compare_and_set` primitive (see Section 2.3.2.1), which atomically sets an atomic reference to a desired value if its current content is physically equal to an expected value. This primitive is commonly used to try committing an atomic operation in a retry loop, as in the `push` and `pop` functions of Figure 2.2.

4.2.3.1 Physical equality in HeapLang

In HeapLang, this primitive is provided but restricted. Indeed, its semantics is only defined if either the expected or the desired value fits in a single memory word in the HeapLang value representation: literals (booleans, integers and pointers[4]) and literal injections[5]; otherwise, the program is stuck. In practice, this restriction forces the programmer to introduce an indirection [Iris development team, 2025b; Jung et al., 2020; Vindum and Birkedal, 2021] to physically compare complex values, *e.g.* lists. Furthermore, when the semantics is defined, values are compared using their Rocq representations; physical equality boils down to Rocq equality.

4.2.3.2 Physical equality in OCaml

In OCaml, physical equality is more tricky and often considered dangerous. *Structural equality*, which we describe in Section 4.2.4, should be the preferred way of comparing values. However, physical equality is typically much faster than structural equality, as it basically compiles to only one assembly instruction. Also, the `Atomic.compare_and_set` requires the comparison to be atomic, ruling out structural equality.

In particular, the semantics of physical equality is *non-deterministic*. To see why, consider the case of immutable blocks, representing constructors and immutable records, *e.g.* `Some 0`. The physical comparison of two seemingly identical immutable blocks, according to the Rocq representation (essentially a tag and a list of fields), may return `false`:

```
let test = Some 0 == Some 0 (* maybe false *)
```

Indeed, at runtime, a non-empty immutable block is represented by a pointer to a tagged memory block. In this case, physical equality is just pointer comparison. It is clear that two pointers being distinct does not imply the pointed memory blocks are. In other words, we cannot determine the result of physical comparison just by looking at the abstract values.

The question is then: what guarantees do we get when physical equality returns `true` and when it returns `false`? Given such guarantees, denoted by $v_1 \approx v_2$ and $v_1 \not\approx v_2$, the non-deterministic semantics is reflected in the logic through the following specification:

$$\frac{\dfrac{\mathsf{True}}{\mathsf{Equal}\ v_1\ v_2}}{b.\ \textbf{if}\ b\ \textbf{then}\ v_1 \approx v_2\ \textbf{else}\ v_1 \not\approx v_2}$$

The OCaml manual documents a partial specification for physical equality, which is precise for basic types such as references, but does not clearly extend to structured values containing a mix of immutable and mutable constructors. The only guarantee that it provides for all values is: if two values are physically equal, they are also structurally equal. This means we do not learn anything when two values are physically distinct.

In the following, we will explore both cases, looking at the optimizations that the compiler or the runtime system may perform. We will show that the aforementioned guarantee is arguably not sufficient to verify interesting concurrent programs and attempt to establish stronger guarantees.

[4]HeapLang allows arbitrary pointer arithmetic and therefore inner pointers. This is forbidden in both OCaml and ZooLang, as any reachable value has to be compatible with the garbage collector.

[5]HeapLang has no primitive notion of constructor, only pairs and injections (left and right).

4.2.3.3 When physical equality returns `true`

Let us go back to the concurrent stack of Figure 2.2 and more specifically the `push` function. To prove STACK-PUSH-SPEC-ATOMIC, we rely on the fact that, if `Atomic.compare_and_set` returns `true`, we actually observe the same list of values in the sense of Rocq equality. However, assuming only structural equality as per OCaml's specification of physical equality, this cannot be proven. To see why, consider, *e.g.*, a stack of references (`'a ref`). As structural equality is indeed *structural*, it traverses the references without comparing their *physical identities*. In other words, we cannot conclude the references are *exactly* the same. Hence, we cannot prove the specification.

This conclusion might seem surprising and counterintuitive. Indeed, we know that physical equality essentially boils down to a comparison instruction, so we should be able to say more. Departing from OCaml's imprecise specification, let us attempt to establish stronger guarantees.

The easy cases are mutable blocks (locations) and functions. Each of these two classes is disjoint from the others. We can reasonably assume that, when physical equality returns `true` and one of the compared values belongs to either of these classes, the two values are actually the same in Rocq. As far as we are aware, there is no optimization that could break this.

In the low-level representation of OCaml values, booleans, integers and empty immutable blocks are represented by immediate integers. These low-level representations induce conflicts: two seemingly distinct values in Rocq may have the same low-level representation. For example, the following tests all return `true`[6]:

```
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 *)
```

The semantics of unrestricted physical equality has to reflect these conflicts. In our experience, restricting compared values similarly to typing is quite burdensome; the specification of polymorphic data structures using physical equality has to be systematically restricted. In summary, when physical equality on immediate values returns `true`, it is guaranteed that they have the same low-level representation.

Finally, let us consider the case of non-empty immutable blocks. At runtime, they are represented by pointers to tagged memory blocks. At first approximation, it is tempting to say that physically equal immutable blocks are definitionally equal in Rocq. Alas, this is not true. To explain why, we have to recall that the OCaml compiler and the runtime system (*e.g.*, through hash-consing) may perform *sharing*: immutable blocks containing physically equal fields may be shared. For example, the following tests may return `true`:

```
let test1 = Some 0 == Some 0 (* maybe true *)
let test2 = [0;1]  == [0;1]  (* maybe true *)
```

On its own, sharing is not a problem. However, coupled with representation conflicts, it can be surprising. Indeed, consider the `any` type defined as:

```
type any = Any : 'a -> any
```

The following tests may return `true`:

[6] `Obj.repr` is an unsafe primitive revealing the memory representation of a value.

```ocaml
let test1 = Any false == Any 0 (* maybe true *)
let test2 = Any None  == Any 0 (* maybe true *)
let test3 = Any []    == Any 0 (* maybe true *)
```

Now, going back to the `push` function of Figure 2.2, we have a problem. Given a stack of `any`, it is possible for the `Atomic.compare_and_set` to observe a current list (*e.g.*, `[Any 0]`) physically equal to the expected list (*e.g.*, `[Any false]`) while these are actually distinct in Rocq. In short, the expected specification STACK-PUSH-SPEC-ATOMIC is incorrect. To fix it, we would need to reason *modulo physical equality*, which is non-standard and quite burdensome.

We believe this really is a shortcoming, at least from the verification perspective. Therefore, we propose to extend OCaml with *generative immutable blocks*[7]. These generative blocks are just like regular immutable blocks, except they cannot be shared. Hence, if physical equality on two generative blocks returns `true`, these blocks are definitionally equal in Rocq. At user level, this notion is materialized by *generative constructors*. For instance, to verify the expected `push` specification, we can use a generative version of lists:

```ocaml
type 'a list =
  | Nil
  | Cons of 'a * 'a list [@generative]
```

4.2.3.4 When physical equality returns `false`

Most formalizations of physical equality in the literature do not give any guarantee when physical equality returns `false`. Many use-cases of physical equality, in particular retry loops, can be verified with only sufficient conditions on `true`. However, in some specific cases, more information is needed.

Consider the `Rcfd` module from the `Eio` [Madhavapeddy and Leonard, 2025] library, an excerpt of which is given in Figure 4.6[8]. Thomas Leonard, its author, suggested that we verify this real-life example because of its intricate logical state (see Chapter 8). However, we found out that it is also relevant regarding the semantics of physical equality. Essentially, it consists in wrapping a file descriptor in a thread-safe way using reference-counting. At creation in the `make` function, the wrapper starts in the `Open` state. At some point, it can switch to the `Closing` state in the `close` function and can never go back to the `Open` state. Crucially, the `Open` state does not change throughout the lifetime of the data structure.

The interest of `Rcfd` lies in the `close` function. First, the function reads the state. If this state is `Closing`, it returns `false`; the wrapper has been closed. If this state is `Open`, it tries to switch to the `Closing` state using `Atomic.Loc.compare_and_set`; if this attempt fails, it also returns `false`. In this particular case, we would like to prove that the wrapper has been closed, or equivalently that `Atomic.Loc.compare_and_set` cannot have observed `Open`. Intuitively, this is true because there is only one `Open`.

Obviously, we need some kind of guarantee related to the *physical identity* of `Open` when `Atomic.Loc.compare_and_set` returns `false`. If `Open` were a mutable block, we could argue that this block cannot be physically distinct from itself; no optimization we know of would allow that. Unfortunately, it is an immutable block, and immutable blocks are subject to more optimizations. In fact, something surprising but allowed[9] by

[7] `https://github.com/clef-men/ocaml/tree/generative_constructors`

[8] We make use of *atomic record fields* as introduced in Section 2.3.2.2.

[9] This has been confirmed by OCaml experts developing the Flambda backend.

```
type state =
  | Open of Unix.file_descr
  | Closing of (unit -> unit)

type t =
  { mutable ops: int [@atomic] ;
    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 (
        if t.ops == 0
        && Atomic.Loc.compare_and_set [%atomic.loc t.state] next closed
        then
          close () ;
        true
      ) else (
        false
      )
```

Figure 4.6: `Rcfd` module from `Eio` (excerpt)

OCaml can happen: *unsharing*, the dual of sharing. Indeed, any immutable block can be unshared, *i.e.* reallocated. For example, the following test may theoretically return `false`:

```
let x = Some 0
let test = x == x (* maybe false *)
```

Going back to `Rcfd`, we have a problem: in the second branch, the `Open` block corresponding to `prev` could be unshared, which would make `Atomic.Loc.compare_and_set` fail. Hence, we cannot prove the expected specification; in fact, the program as it is written has a bug.

To remedy this unfortunate situation, we propose to reuse the notions of generative immutable blocks, that we introduced to prevent sharing, to also forbid unsharing by the OCaml compiler – we implemented this in an experiment branch of OCaml.

In our semantics, each generative block is annotated with a *logical identifier*[10]representing its physical identity, much like a location for a mutable block. If physical equality on two generative blocks returns `false`, the two identifiers are necessarily distinct. Given this semantics, we can verify the `close` function. Indeed, if `Atomic.Loc.compare_and_set` fails, we now know that the identifiers of the two blocks, if any, are distinct. As there is only one `Open` block whose identifier does not change, it cannot be the case that the current state is `Open`, hence it is `Closing`. We can verify this function after adding the following annotation:

```
type state =
  | Open of Unix.file_descr [@generative]
  | Closing of (unit -> unit)
```

4.2.3.5 Summary

In summary, we give the following informal specification to physical equality in ZooLang, which can serve as a basis for specifying physical equality in OCaml:

- On values whose low-level representation is an integer (integers, booleans, empty blocks), physical equality is equality of low-level integers.
- On mutable blocks, represented as memory locations, physical equality is equality of locations.
- On generative immutable blocks, physical equality is equality of identities.
- On non-generative immutable blocks, physical equality is under-specified, but it implies that the two blocks have the same tags and their fields are recursively physical equal.
- Two values that do not fall into any of the above categories are never physically equal.

4.2.3.6 Formalization

Formally, we define two relations $v_1 \approx v_2$ and $v_1 \not\approx v_2$ that respectively satisfy the rules of Figure 4.7 and Figure 4.8. In the Rocq mechanization, we developed a tactic

[10] Actually, for practical reasons, we distinguish identified and unidentified generative blocks.

$$\frac{v_1 = v_2}{v_1 \approx v_2}\ \textsc{similar-refl} \qquad \frac{v_1 \approx v_2}{v_2 \approx v_1}\ \textsc{similar-sym} \qquad \frac{v_1 \approx v_2 \qquad v_2 \approx v_3}{v_1 \approx v_3}\ \textsc{similar-trans}$$

$$\frac{b_1 \approx b_2}{b_1 = b_2}\ \textsc{similar-bool} \qquad \frac{n_1 \approx n_2}{n_1 = n_2}\ \textsc{similar-int} \qquad \frac{\ell_1 \approx \ell_2}{\ell_1 = \ell_2}\ \textsc{similar-loc}$$

$$\frac{\mathsf{ValBlock\ Nongenerative}\ tag_1\ \overline{v}_1 \approx \mathsf{ValBlock\ Nongenerative}\ tag_2\ \overline{v}_2}{tag_1 = tag_2 \wedge \overline{v}_1 \approx \overline{v}_2}\ \textsc{similar-block-nongenerative}$$

$$\frac{\begin{array}{c}0 < \mathsf{length}\ \overline{v}_1 \vee 0 < \mathsf{length}\ \overline{v}_2 \\ \mathsf{ValBlock}\ (\mathsf{Generative}\ bid_1)\ tag_1\ \overline{v}_1 \approx \mathsf{ValBlock}\ (\mathsf{Generative}\ bid_2)\ tag_2\ \overline{v}_2\end{array}}{bid_1 = bid_1 \wedge tag_1 = tag_2 \wedge \overline{v}_1 = \overline{v}_2}\ \textsc{similar-block-generative}$$

$$\frac{\begin{array}{c}0 < \mathsf{length}\ \overline{v}_1 \vee 0 < \mathsf{length}\ \overline{v}_2 \\ \mathsf{ValBlock\ Nongenerative}\ tag_1\ \overline{v}_1 \approx \mathsf{ValBlock}\ (\mathsf{Generative}\ bid_2)\ tag_2\ \overline{v}_2\end{array}}{\mathsf{False}}\ \textsc{similar-nongenerative-generative}$$

$$\frac{lit \approx \mathsf{ValRecs}\ i\ \overline{rec}}{\mathsf{False}}\ \textsc{similar-lit-recs} \qquad \frac{0 < \mathsf{length}\ \overline{v} \qquad lit \approx \mathsf{ValBlock}\ gen\ tag\ \overline{v}}{\mathsf{False}}\ \textsc{similar-lit-block}$$

$$\frac{\mathsf{ValRecs}\ i\ \overline{rec} \approx \mathsf{ValBlock}\ gen\ tag\ \overline{v}}{\mathsf{False}}\ \textsc{similar-recs-block}$$

Figure 4.7: Value similarity

$$\frac{v_1 \not\approx v_2}{v_2 \not\approx v_1}\ \textsc{nonsimilar-sym}$$

$$\frac{b_1 \not\approx b_2}{b_1 \neq b_2}\ \textsc{nonsimilar-bool} \qquad \frac{n_1 \not\approx n_2}{n_1 \neq n_2}\ \textsc{nonsimilar-int} \qquad \frac{\ell_1 \not\approx \ell_2}{\ell_1 \neq \ell_2}\ \textsc{nonsimilar-loc}$$

$$\frac{\mathsf{ValBlock}\ gen_1\ tag_1\ [] \not\approx \mathsf{ValBlock}\ gen_2\ tag_2\ []}{tag_1 \neq tag_2}\ \textsc{nonsimilar-block-empty}$$

$$\frac{0 < \mathsf{length}\ \overline{v} \qquad \mathsf{ValBlock}\ (\mathsf{Generative}\ bid_1)\ tag\ \overline{v} \not\approx \mathsf{ValBlock}\ (\mathsf{Generative}\ bid_2)\ tag\ \overline{v}}{bid_1 \neq bid_2}\ \textsc{nonsimilar-block-generative}$$

Figure 4.8: Value non-similarity

that automatically simplifies physical equality assumptions using these rules; in practice, we found it to be very effective.

Crucially, the two relations also satisfy the following "compatibility" rules:

$$\textsc{similar-or-nonsimilar} \quad v_1 \approx v_2 \lor v_1 \not\approx v_2 \qquad\qquad \frac{v_1 \not\approx v_2 \qquad v_2 \approx v_3}{v_1 \not\approx v_3}\ \textsc{nonsimilar-similar}$$

SIMILAR-OR-NONSIMILAR is required for proving that physical equality is always safe to execute; without it, physical equality would have to be restricted to "safe" values. As for NONSIMILAR-SIMILAR, it is needed to verify the algorithm of Chapter 11.

4.2.4 Structural equality

Structural equality is also supported. More precisely, it is not part of the semantics of the language but implemented using low-level primitives (see Figure 4.9). The reason is that it is in fact difficult to specify for arbitrary values. In general, we have to compare graphs — which implies structural comparison may diverge.

Accordingly, the specification of $v_1 = v_2$ requires the (partial) ownership of a *memory footprint* corresponding to the union of the two compared graphs, giving the permission to traverse them safely. If it terminates, the comparison decides whether the two graphs are bisimilar (modulo representation conflicts, as described in Section 4.2.3). The resulting

```
let rec structeq v1 v2 =
  if Obj.is_int v1 then
    if Obj.is_int v2 then
      v1 == v2
    else
      false
  else if Obj.is_int v2 then
    false
  else (
    Obj.tag v1 == Obj.tag v2 &&
    let sz = Obj.size v1 in
    sz == Obj.size v2 &&
    structeq_aux v1 v2 sz
  )
and structeq_aux v1 v2 i =
  if i == 0 then
    true
  else
    let i = i - 1 in
    structeq (Obj.field v1 i) (Obj.field v2 i) &&
    structeq_aux v1 v2 i
```

Figure 4.9: Implementation of structural equality

specification is:

$$\begin{array}{l}
\quad \mathsf{val\text{-}traversable}\ \mathit{footprint}\ v_1\ * \\
\quad \mathsf{val\text{-}traversable}\ \mathit{footprint}\ v_2\ * \\
\quad \mathsf{structeq\text{-}footprint}\ \mathit{footprint} \\
\hline
\qquad\qquad\qquad v_1 = v_2 \\
\hline
b.\ \textbf{if}\ b\ \textbf{then} \\
\qquad \mathsf{val\text{-}structeq}\ \mathit{footprint}\ v_1\ v_2 \\
\quad \textbf{else} \\
\qquad \mathsf{val\text{-}structneq}\ \mathit{footprint}\ v_1\ v_2
\end{array}$$

Obviously, this general specification is not very convenient to work with. Fortunately, for *abstract* values (without any mutable part), we can prove a much simpler variant saying that structural equality coincides with physical equality:

$$\begin{array}{c}
\mathsf{val\text{-}abstract}\ v_1\ * \\
\mathsf{val\text{-}abstract}\ v_2 \\
\hline
v_1 = v_2 \\
\hline
b.\ \textbf{if}\ b\ \textbf{then}\ v_1 \approx v_2\ \textbf{else}\ v_1 \not\approx v_2
\end{array}$$

4.2.5 Semantics

We define the small-step operational semantics of ZooLang in four stages: (1) pure steps (Figure 4.10) involve only the executed expression; (2) base steps (Figure 4.11) also involve the state of the execution; (3) head steps (Figure 4.12) also involve the identifier

STEP-REC

$$\mathsf{Rec}\ f\ \ bdr\ \ e \xrightarrow{\mathsf{pure}} \mathsf{ValRecs}\ 0\ [\{\ \mathsf{fun}\!: f;\ \mathsf{param}\!: bdr;\ \mathsf{body}\!: e\ \}]$$

STEP-APP

$$\frac{\overline{rec}\,[i] = \mathsf{Some}\ rec}{\mathsf{App}\ (\mathsf{ValRecs}\ i\ \overline{rec})\ v \xrightarrow{\mathsf{pure}} \mathsf{eval\text{-}app}\ \overline{rec}\ rec.\mathsf{param}\ v\ rec.\mathsf{body}}$$

STEP-LET

$$\mathsf{Let}\ bdr\ \ v_1\ \ e_2 \xrightarrow{\mathsf{pure}} \mathsf{subst}\ bdr\ \ v_1\ \ e_2$$

STEP-UNOP

$$\frac{\mathsf{eval\text{-}unop}\ op\ \ v = \mathsf{Some}\ v'}{\mathsf{Unop}\ op\ \ v \xrightarrow{\mathsf{pure}} v'}$$

STEP-BINOP

$$\frac{\mathsf{eval\text{-}binop}\ op\ \ v_1 v_2 = \mathsf{Some}\ v'}{\mathsf{Binop}\ op\ \ v_1\ \ v_2 \xrightarrow{\mathsf{pure}} v'}$$

STEP-IF

$$\mathsf{If}\ b\ \ e_1\ \ e_2 \xrightarrow{\mathsf{pure}} \textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2$$

STEP-FOR

$$\begin{aligned}\mathsf{For}\ n_1\ \ n_2\ \ e \xrightarrow{\mathsf{pure}}\ & \textbf{if}\ \mathsf{decide}\ (n_2 \leq n_1)\ \textbf{then} \\ & \quad \mathsf{ValUnit} \\ & \textbf{else} \\ & \quad \mathsf{Seq}\ (\mathsf{App}\ e\ \ n_1)\ (\mathsf{For}\ (n_1 + 1)\ \ n_2\ \ e)\end{aligned}$$

STEP-BLOCK-IMMUTABLE-NONGENERATIVE

$$\mathsf{Block}\ \mathsf{ImmutableNongenerative}\ tag\ \overline{v} \xrightarrow{\mathsf{pure}} \mathsf{ValBlock}\ \mathsf{Nongenerative}\ tag\ \overline{v}$$

STEP-BLOCK-IMMUTABLE-GENERATIVE-WEAK

$$\mathsf{Block}\ \mathsf{ImmutableGenerativeWeak}\ tag\ \overline{v} \xrightarrow{\mathsf{pure}} \mathsf{ValBlock}\ (\mathsf{Generative}\ \mathsf{None})\ tag\ \overline{v}$$

STEP-MATCH-IMMUTABLE

$$\frac{\mathsf{eval\text{-}match}\ tag\ (\mathsf{length}\ \overline{v})\ (\mathsf{SubjectBlock}\ gen\ \overline{v})\ bdr_{\mathit{fb}}\ \ e_{\mathit{fb}}\ \ \overline{br} = \mathsf{Some}\ e}{\mathsf{Match}\ (\mathsf{ValBlock}\ gen\ tag\ \overline{v})\ bdr_{\mathit{fb}}\ \ e_{\mathit{fb}}\ \ \overline{br} \xrightarrow{\mathsf{pure}} e}$$

STEP-GET-TAG-IMMUTABLE

$$\frac{0 < \mathsf{length}\ \overline{v}}{\mathsf{GetTag}\ (\mathsf{ValBlock}\ gen\ tag\ \overline{v}) \xrightarrow{\mathsf{pure}} \mathsf{encode\text{-}tag}\ tag}$$

STEP-GET-SIZE-IMMUTABLE

$$\frac{0 < \mathsf{length}\ \overline{v}}{\mathsf{GetSize}\ (\mathsf{ValBlock}\ gen\ tag\ \overline{v}) \xrightarrow{\mathsf{pure}} \mathsf{length}\ \overline{v}}$$

STEP-GET-FIELD-IMMUTABLE

$$\frac{\overline{v}\,[\mathit{fld}] = \mathsf{Some}\ v}{\mathsf{Load}\ (\mathsf{ValBlock}\ gen\ tag\ \overline{v})\ \mathit{fld} \xrightarrow{\mathsf{pure}} v}$$

Figure 4.10: Semantics: pure step

STEP-PURE

$$\frac{e_1 \xrightarrow{\text{pure}} e_2}{e_1, \sigma \xrightarrow{\text{base}} e_2, \sigma}$$

STEP-EQUAL-FAIL

$$\frac{v_1 \not\approx v_2}{\mathsf{Equal}\ v_1\ v_2, \sigma \xrightarrow{\text{base}} \mathsf{false}, \sigma}$$

STEP-EQUAL-SUCCESS

$$\frac{v_1 \approx v_2}{\mathsf{Equal}\ v_1\ v_2, \sigma \xrightarrow{\text{base}} \mathsf{true}, \sigma}$$

STEP-ALLOC

$$\frac{0 \le n \qquad \sigma.\mathsf{headers}\,[\ell] = \mathsf{None} \qquad \forall\ i.\ i < n \to \sigma.\mathsf{heap}\,[(\ell + i)] = \mathsf{None}}{\mathsf{Alloc}\ tag\ n, \sigma \xrightarrow{\text{base}} \ell, \mathsf{state\text{-}alloc}\ \ell\ \{\,\mathsf{tag}\text{: } tag;\ \mathsf{size}\text{: } n\,\}\ (\mathsf{replicate}\ n\ \mathsf{ValUnit})\ \sigma}$$

STEP-BLOCK-MUTABLE

$$\frac{0 < \mathsf{length}\ \overline{v} \qquad \sigma.\mathsf{headers}\,[\ell] = \mathsf{None} \qquad \forall\ i.\ i < \mathsf{length}\ \overline{v} \to \sigma.\mathsf{heap}\,[(\ell + i)] = \mathsf{None}}{\mathsf{Block\ Mutable}\ tag\ \overline{v}, \sigma \xrightarrow{\text{base}} \ell, \mathsf{state\text{-}alloc}\ \ell\ \{\,\mathsf{tag}\text{: } tag;\ \mathsf{size}\text{: length}\ \overline{v}\,\}\ \overline{v}\ \sigma}$$

STEP-IMMUTABLE-GENERATIVE-STRONG

$$\mathsf{Block\ ImmutableGenerativeStrong}\ tag\ \overline{v}, \sigma \xrightarrow{\text{base}} \mathsf{ValBlock}\ (\mathsf{Generative}\ (\mathsf{Some}\ bid)\ tag\ \overline{v}, \sigma$$

STEP-MATCH-MUTABLE

$$\frac{\begin{array}{c}\sigma.\mathsf{headers}\,[\ell] = \mathsf{Some}\ hdr \\ \mathsf{eval\text{-}match}\ hdr.\mathsf{tag}\ hdr.\mathsf{size}\ (\mathsf{SubjectLoc}\ \ell)\ bdr_{fb}\ e_{fb}\ \overline{br} = \mathsf{Some}\ e\end{array}}{\mathsf{Match}\ \ell\ bdr_{fb}\ e_{fb}\ \overline{br}, \sigma \xrightarrow{\text{base}} e, \sigma}$$

STEP-GET-TAG-MUTABLE

$$\frac{\sigma.\mathsf{headers}\,[\ell] = \mathsf{Some}\ hdr}{\mathsf{GetTag}\ \ell, \sigma \xrightarrow{\text{base}} \mathsf{encode\text{-}tag}\ hdr.\mathsf{tag}, \sigma}$$

STEP-GET-SIZE-MUTABLE

$$\frac{\sigma.\mathsf{headers}\,[\ell] = \mathsf{Some}\ hdr}{\mathsf{GetSize}\ \ell, \sigma \xrightarrow{\text{base}} \mathsf{encode\text{-}tag}\ hdr.\mathsf{size}, \sigma}$$

STEP-GET-FIELD-MUTABLE

$$\frac{\sigma.\mathsf{heap}\,[(\ell + fld)] = \mathsf{Some}\ v}{\mathsf{Load}\ \ell\ fld, \sigma \xrightarrow{\text{base}} v, \sigma}$$

STEP-SET-FIELD

$$\frac{\sigma.\mathsf{heap}\,[(\ell + fld)] = \mathsf{Some}\ w}{\mathsf{Store}\ \ell\ fld\ v, \sigma \xrightarrow{\text{base}} \mathsf{ValUnit}, \sigma\,[\mathsf{heap} \mapsto \sigma.\mathsf{heap}\,[\ell + fld \mapsto v]]}$$

STEP-XCHG

$$\frac{\sigma.\mathsf{heap}\,[(\ell + fld)] = \mathsf{Some}\ w}{\mathsf{Xchg}\ (\mathsf{ValTuple}\ [\ell; fld])\ v, \sigma \xrightarrow{\text{base}} w, \sigma\,[\mathsf{heap} \mapsto \sigma.\mathsf{heap}\,[\ell + fld \mapsto v]]}$$

STEP-CAS-FAIL

$$\frac{\sigma.\mathsf{heap}\,[(\ell + fld)] = \mathsf{Some}\ v \qquad v \not\approx v_1}{\mathsf{CAS}\ (\mathsf{ValTuple}\ [\ell; fld])\ v_1\ v_2, \sigma \xrightarrow{\text{base}} \mathsf{false}, \sigma}$$

STEP-CAS-SUCCESS

$$\frac{\sigma.\mathsf{heap}\,[(\ell + fld)] = \mathsf{Some}\ v \qquad v \approx v_1}{\mathsf{CAS}\ (\mathsf{ValTuple}\ [\ell; fld])\ v_1\ v_2, \sigma \xrightarrow{\text{base}} \mathsf{true}, \sigma\,[\mathsf{heap} \mapsto \sigma.\mathsf{heap}\,[\ell + fld \mapsto v_2]]}$$

STEP-FAA

$$\frac{\sigma.\mathsf{heap}\,[(\ell + fld)] = \mathsf{Some}\ n_1}{\mathsf{FAA}\ (\mathsf{ValTuple}\ [\ell; fld])\ n_2, \sigma \xrightarrow{\text{base}} n_1, \sigma\,[\mathsf{heap} \mapsto \sigma.\mathsf{heap}\,[\ell + fld \mapsto n_1 + n_2]]}$$

Figure 4.11: Semantics: base step

STEP-BASE
$$\frac{e_1, \sigma_1 \xrightarrow{\textsf{base}} e_2, \sigma_2}{e_1, \sigma_1 \xrightarrow[tid]{\textsf{head}} e_2, \sigma_2, [], []}$$

STEP-FORK
$$\frac{\textsf{val-immediate}\ v}{\textsf{Fork}\ e, \sigma \xrightarrow[tid]{\textsf{head}} \textsf{ValUnit}, \sigma\left[\textsf{locals} \mapsto \sigma.\textsf{locals} +\!\!+ [v]\right], [e], []}$$

STEP-GET-LOCAL
$$\frac{\sigma.\textsf{locals}\,[tid] = \textsf{Some}\ v}{\textsf{GetLocal}, \sigma \xrightarrow[tid]{\textsf{head}} v, \sigma, [], []}$$

STEP-SET-LOCAL
$$\frac{\sigma.\textsf{locals}\,[tid] = \textsf{Some}\ w}{\textsf{SetLocal}\ v, \sigma \xrightarrow[tid]{\textsf{head}} \textsf{ValUnit}, \sigma\left[\textsf{locals} \mapsto \sigma.\textsf{locals}\,[tid \mapsto v]\right], [], []}$$

STEP-PROPH
$$\frac{pid \notin \sigma.\textsf{prophets}}{\textsf{Proph}, \sigma \xrightarrow[tid]{\textsf{head}} pid, \sigma\left[\textsf{prophets} \mapsto \sigma.\textsf{prophets} \uplus \{pid\}\right], [], []}$$

STEP-RESOLVE
$$\frac{e, \sigma \xrightarrow[tid]{\textsf{head}} w, \sigma', \overline{e}, \kappa}{\textsf{Resolve}\ e\ pid\ v, \sigma \xrightarrow[tid]{\textsf{head}} w, \sigma', \overline{e}, \kappa +\!\!+ [(pid, w, v)]}$$

Figure 4.12: Semantics: head step

of the current domain and may emit prophecy observations and spawn new domains; (4) thread-pool steps (Figure 4.13) involve the entire execution configuration, including all spawned domains. We omit the definition of auxiliary functions, which can be found in the mechanization: eval-app, eval-unop, eval-binop, eval-match, state-alloc.

Overall, this semantics is mostly standard; in particular, the semantics of prophecy variables is taken directly from Jung et al. [2020]. The execution state carries not only a mutable heap but also immutable headers attached to memory locations and mutable domain-local storage.

Note that the evaluation order is well-defined for all constructs: right-to-left for application, binary operators, allocations, memory accesses and prophecy resolution; left-to-right for for-loop. Theoretically speaking, this is unsound since OCaml has unspecified evaluation order[11]. However, it is well-known[12] that not assuming a right-to-left evaluation order for application is *extremely* damaging from the verification perspective. Moreover, our evaluation order is mostly respected by the OCaml compiler and many programmers rely on it.

4.2.6 Program logic

ZooLang comes with a Iris-based program logic displayed in Figure 4.14: reasoning rules expressed in separation logic and proved correct with respect to the semantics. We omit the rules for prophecy variables, that we present separately in Chapter 5.

[11] `https://ocaml.org/manual/5.3/expr.html#sss:expr-functions-application`
[12] `https://gitlab.mpi-sws.org/iris/iris/-/blob/master/iris_heap_lang/lang.v#L12`

$$
\begin{array}{llcl}
\text{evaluation frame} & k & ::= & \mathsf{App}\ \square\ v_2 \mid \mathsf{App}\ e_1\ \square \\
 & & \mid & \mathsf{Let}\ \mathit{bdr}\ \square\ e_2 \\
 & & \mid & \mathsf{Unop}\ \mathit{unop}\ \square \\
 & & \mid & \mathsf{Binop}\ \mathit{binop}\ \square\ v_2 \mid \mathsf{Binop}\ \mathit{binop}\ e_1\ \square \\
 & & \mid & \mathsf{Equal}\ \square\ v_2 \mid \mathsf{Equal}\ e_1\ \square \\
 & & \mid & \mathsf{If}\ \square\ e_1\ e_2 \\
 & & \mid & \mathsf{For}\ \square\ e_2\ e_3 \mid \mathsf{For}\ v_1\ \square\ e_3 \\
 & & \mid & \mathsf{Alloc}\ \square\ v_2 \mid \mathsf{Alloc}\ e_1\ \square \\
 & & \mid & \mathsf{Block}\ \mathit{mut}\ \mathit{tag}\ (\overline{e} \mathbin{+\!\!+} [\square] \mathbin{+\!\!+} \overline{v}) \\
 & & \mid & \mathsf{Match}\ \square\ \mathit{bdr}_{\mathit{fb}}\ e_{\mathit{fb}}\ \overline{\mathit{br}} \\
 & & \mid & \mathsf{GetTag}\ \square \\
 & & \mid & \mathsf{GetSize}\ \square \\
 & & \mid & \mathsf{Load}\ \square\ v_2 \mid \mathsf{Load}\ e_1\ \square \\
 & & \mid & \mathsf{Store}\ \square\ v_2\ v_3 \mid \mathsf{Store}\ e_1\ \square\ v_3 \mid \mathsf{Store}\ e_1\ e_2\ \square \\
 & & \mid & \mathsf{Xchg}\ \square\ v_2 \mid \mathsf{Xchg}\ e_1\ \square \\
 & & \mid & \mathsf{CAS}\ v_1\ v_2\ \square \mid \mathsf{CAS}\ e_0\ \square\ v_2 \mid \mathsf{CAS}\ e_0\ e_1\ \square \\
 & & \mid & \mathsf{FAA}\ \square\ v_2 \mid \mathsf{FAA}\ e_1\ \square \\
 & & \mid & \mathsf{SetLocal}\ \square \\
 & & \mid & \mathsf{Resolve}\ k\ v_1\ v_2 \mid \mathsf{Resolve}\ e_0\ \square\ v_2 \mid \mathsf{Resolve}\ e_0\ e_1\ \square \\
\text{evaluation context} & K & ::= & \square \mid k\,[K]
\end{array}
$$

$$
\textsc{step-head}\quad
\frac{\mathit{tp}\,[\mathit{tid}] = \mathsf{Some}\ (K\,[e_1]) \qquad e_1, \sigma_1 \xrightarrow[\mathit{tid}]{\mathsf{head}} e_2, \sigma_2, \overline{e}, \kappa}{\mathit{tp}, \sigma_1 \xrightarrow{\mathsf{tp}} \mathit{tp}\,[\mathit{tid} \mapsto K\,[e_2]] \mathbin{+\!\!+} \overline{e}, \sigma_2}
$$

Figure 4.13: Semantics: thread pool step

WP-POST
$$\frac{\Phi\ v}{\mathsf{wp}_{\mathcal{E}}\ v\ \left\{\ \Phi\ \right\}}$$

WP-BIND
$$\frac{\mathsf{wp}_{\mathcal{E}}\ e\ \left\{\ v.\ \mathsf{wp}_{\mathcal{E}}\ K[v]\ \left\{\ \Phi\ \right\}\ \right\}}{\mathsf{wp}_{\mathcal{E}}\ K[e]\ \left\{\ \Phi\ \right\}}$$

WP-PURE
$$\frac{e_1 \xrightarrow{\mathsf{pure}} e_2 \qquad \mathsf{wp}_{\mathcal{E}}\ e_2\ \left\{\ \Phi\ \right\}}{\mathsf{wp}_{\mathcal{E}}\ e_1\ \left\{\ \Phi\ \right\}}$$

WP-EQUAL
$$\frac{\triangleright\ \Big(\big(v_1 \not\approx v_2 \mathrel{-\!\!*} \Phi\ \mathsf{false}\big) \land \left(v_1 \approx v_2 \mathrel{-\!\!*} \Phi\ \mathsf{true}\right)\Big)}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{Equal}\ v_1\ v_2\ \left\{\ \Phi\ \right\}}$$

WP-ALLOC
$$\begin{array}{l} \qquad 0 \le tag * 0 \le n \\ \hline \qquad \mathsf{Alloc}\ tag\ n\ \fatsemi\ \mathcal{E} \\ \hline \ell.\ \ell \mapsto_{\mathsf{h}} \{\ \mathsf{tag}\text{: } tag;\ \mathsf{size}\text{: } n\ \} * \\ \quad \mathsf{meta\text{-}token}\ \ell\ \top * \\ \quad \mathop{\Huge *}_{i \in [\![0;n)} (\ell + i) \mapsto () \end{array}$$

WP-BLOCK-MUTABLE
$$\begin{array}{l} \qquad 0 < \mathsf{length}\ \overline{v} \\ \hline \qquad \mathsf{Block}\ \mathsf{Mutable}\ tag\ \overline{v}\ \fatsemi\ \mathcal{E} \\ \hline \ell.\ \ell \mapsto_{\mathsf{h}} \{\ \mathsf{tag}\text{: } tag;\ \mathsf{size}\text{: } \mathsf{length}\ \overline{v}\ \} * \\ \quad \mathsf{meta\text{-}token}\ \ell\ \top * \\ \quad \mathop{\Huge *}_{i \mapsto v \in \overline{v}} (\ell + i) \mapsto v \end{array}$$

WP-BLOCK-GENERATIVE
$$\begin{array}{l} \qquad \mathsf{True} \\ \hline \qquad \mathsf{Block}\ \mathsf{ImmutableGenerativeStrong}\ tag\ \overline{v}\ \fatsemi\ \mathcal{E} \\ \hline res.\ \exists\ bid. \\ \qquad res = \mathsf{ValBlock}\ (\mathsf{Generative}\ (\mathsf{Some}\ bid))\ tag\ \overline{v} \end{array}$$

WP-MATCH
$$\frac{\begin{array}{c}\mathsf{eval\text{-}match}\ hdr.\mathsf{tag}\ hdr.\mathsf{size}\ (\mathsf{SubjectLoc}\ \ell)\ bdr_{\mathit{fb}}\ e_{\mathit{fb}}\ \overline{br} = \mathsf{Some}\ e \\ \triangleright\ \ell \mapsto_{\mathsf{h}} hdr \qquad \triangleright\ \mathsf{wp}_{\mathcal{E}}\ e\ \left\{\ \Phi\ \right\}\end{array}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{Match}\ \ell\ bdr_{\mathit{fb}}\ e_{\mathit{fb}}\ \overline{br}\ \left\{\ \Phi\ \right\}}$$

WP-TAG
$$\frac{\triangleright\ \ell \mapsto_{\mathsf{h}} hdr \qquad \triangleright\ \Phi\ (\mathsf{encode\text{-}tag}\ hdr.\mathsf{tag})}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{GetTag}\ \ell\ \left\{\ \Phi\ \right\}}$$

WP-SIZE
$$\frac{\triangleright\ \ell \mapsto_{\mathsf{h}} hdr \qquad \triangleright\ \Phi\ hdr.\mathsf{size}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{GetSize}\ \ell\ \left\{\ \Phi\ \right\}}$$

WP-LOAD
$$\begin{array}{l} \qquad \triangleright(\ell + \mathit{fld}) \overset{q}{\mapsto} v \\ \hline \qquad \mathsf{Load}\ \ell\ \mathit{fld}\ \fatsemi\ \mathcal{E} \\ \hline res.\ res = v * \\ \qquad (\ell + \mathit{fld}) \overset{q}{\mapsto} v \end{array}$$

WP-STORE
$$\begin{array}{l} \qquad \triangleright(\ell + \mathit{fld}) \mapsto w \\ \hline \qquad \mathsf{Store}\ \ell\ \mathit{fld}\ v\ \fatsemi\ \mathcal{E} \\ \hline ().\ (\ell + \mathit{fld}) \mapsto v \end{array}$$

WP-XCHG
$$\begin{array}{l} \qquad \triangleright(\ell + \mathit{fld}) \mapsto w \\ \hline \mathsf{Xchg}\ (\mathsf{ValTuple}\ [\ell; \mathit{fld}])\ v\ \fatsemi\ \mathcal{E} \\ \hline \qquad res.\ res = w * \\ \qquad\qquad (\ell + \mathit{fld}) \mapsto v \end{array}$$

WP-CAS
$$\frac{\begin{array}{c}\triangleright(\ell + \mathit{fld}) \mapsto v \\ \triangleright\ \Big(\big(v \not\approx v_1 \mathrel{-\!\!*} (\ell + \mathit{fld}) \mapsto v \mathrel{-\!\!*} \Phi\ \mathsf{false}\big) \land \left(v \approx v_1 \mathrel{-\!\!*} (\ell + \mathit{fld}) \mapsto v_2 \mathrel{-\!\!*} \Phi\ \mathsf{true}\right)\Big)\end{array}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{CAS}\ (\mathsf{ValTuple}\ [\ell; \mathit{fld}])\ v_1\ v_2\ \left\{\ \Phi\ \right\}}$$

Figure 4.14: Program logic (excerpt) (1/2)

$$\frac{\triangleright(\ell + \mathit{fld}) \mapsto n_1}{\dfrac{\mathsf{FAA}\ (\mathsf{ValTuple}\ [\ell; \mathit{fld}])\ n_2 \ ⨟\ \mathcal{E}}{\mathit{res}.\ \mathit{res} = n_1 \ast (\ell + \mathit{fld}) \mapsto n_1 + n_2}}\ \textsc{wp-faa}$$

$$\frac{\triangleright\big(\forall\ \mathit{tid}\ v.\ \mathit{tid} \mapsto_{\mathsf{l}} v \twoheadrightarrow \mathsf{wp}\ e \ ⨟\ \mathit{tid}\ \{\_.\ \mathsf{True}\}\big) \qquad \triangleright\Phi\ ()}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{Fork}\ e\ \{\Phi\}}\ \textsc{wp-fork}$$

$$\frac{\triangleright\ \mathit{tid} \overset{q}{\mapsto}_{\mathsf{l}} v}{\dfrac{\mathsf{GetLocal} \ ⨟\ \mathit{tid} \ ⨟\ \mathcal{E}}{\mathit{res}.\ \mathit{res} = v \ast \mathit{tid} \overset{q}{\mapsto}_{\mathsf{l}} v}}\ \textsc{wp-get-local}$$

$$\frac{\triangleright\ \mathit{tid} \mapsto_{\mathsf{l}} w}{\dfrac{\mathsf{SetLocal}\ v \ ⨟\ \mathit{tid} \ ⨟\ \mathcal{E}}{().\ \mathit{tid} \mapsto_{\mathsf{l}} v}}\ \textsc{wp-set-local}$$

Figure 4.14: Program logic (excerpt) (2/2)

Most rules are straightforward; we use sequential specifications (see Section 3.4) and weakest preconditions (see Section 3.5) possibly annotated with a domain identifier. The assertion $\ell \overset{q}{\mapsto} v$ represents the fractional ownership of location ℓ and the knowledge that it contains value v; when the fraction q is 1, it represents the full ownership of ℓ. Similarly, the assertion $\mathit{tid} \overset{q}{\mapsto}_{\mathsf{l}} v$ represents the fractional ownership of the domain-local storage of domain tid and the knowledge that it contains v. The persistent assertion $\ell \mapsto_{\mathsf{h}} \mathit{hdr}$ represents the knowledge that location ℓ has header hdr. Finally, the assertion $\mathsf{meta\text{-}token}\ \ell\ \mathcal{E}$ is part of the meta theory[13] that allows to persistently associate meta data to locations:

$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad \mathsf{meta\text{-}token}\ \ell\ \mathcal{E}_2}{\mathsf{meta\text{-}token}\ \ell\ \mathcal{E}_1 \ast \mathsf{meta\text{-}token}\ \ell\ (\mathcal{E}_2 \setminus \mathcal{E}_1)}\ \textsc{meta-token-difference}$$

$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad \mathsf{meta\text{-}token}\ \ell\ \mathcal{E}}{\dot{\Rrightarrow}\ \mathsf{meta}\ \ell\ \mathcal{E}_1\ x}\ \textsc{meta-set}$$

$$\frac{\mathsf{meta}\ \ell\ \mathcal{E}\ x_1 \qquad \mathsf{meta}\ \ell\ \mathcal{E}\ x_2}{x_1 = x_2}\ \textsc{meta-agree}$$

We use this mechanism extensively to avoid exposing ghost names in specifications.

Our program logic is sound in the following sense:

$$\frac{\mathsf{state\text{-}wf}\ \sigma \qquad \forall\ v.\ 0 \mapsto_{\mathsf{l}} v \twoheadrightarrow \mathsf{wp}\ e \ ⨟\ 0\ \{\_.\ \mathsf{True}\}}{\mathsf{safe}\ ([e], \sigma)}\ \textsc{wp-adequacy}$$

In words, if the user can prove a weakest precondition for e in Iris, then e is safe to execute, *i.e.* cannot get suck, in any well-formed state (where some global variables have been initialized).

[13] `https://gitlab.mpi-sws.org/iris/iris/-/blob/master/iris/base_logic/lib/gen_heap.v`

4.2.7 Proof mode

We mechanized ZooLang and the program logic in the Rocq proof assistant, on top of Iris [Iris development team, 2025a]. Our mechanization includes tactics integrating into the Iris proof mode [Krebbers et al., 2018] to apply the reasoning rules and supports Diaframe [Mulder et al., 2022; Mulder and Krebbers, 2023], enabling proof automation.

4.3 Related work

Non-automated verification. In non-automated verification, the verified program is translated, manually or in an automated way, into a representation living inside a proof assistant where users write and prove specifications.

Translating into the native language of the proof assistant, such as Gallina for Rocq, is challenging as it is hard to faithfully preserve the semantics of the source language, *e.g.* non-terminating functions. Monadic translations should support it, but faithfully encoding all impure behaviors is challenging and tools typically provide a best-effort translation [Claret, 2024; Spector-Zabusky et al., 2018] that is only approximately sound.

The representation may be embedded, meaning the semantics of the language is formalized in the proof assistant. This is the path taken by some recent works [Chajed et al., 2019; Gondelman et al., 2023; Charguéraud, 2023] harnessing the power of separation logic. In particular, (1) CFML [Charguéraud, 2023], (2) Osiris [Seassau et al., 2025] and (3) DRFCaml [Georges et al., 2025] target OCaml.

(1) CFML does not support concurrency and is not based on Iris.

(2) Osiris is based on Iris but does not support concurrency. Its design philosophy is more perfectionist than pragmatic, especially in its treatment of evaluation order, at the cost of a complex program logic. The relatively small number of verified examples suggests that it is not yet ready for practical verification at scale.

(3) DRFCaml is based on Iris and does support concurrency. It is mostly an extension of HeapLang with features (modalities and stack regions) entirely orthogonal to our work; in particular, it also assumes a sequentially consistent memory model. The crucial difference is that it forbids data races on non-atomic locations, which makes it compatible with OCaml 5 thanks to the DRF property [Dolan et al., 2018] but is too restrictive to verify legal concurrent programs, including some that we verified.

Semi-automated verification. In semi-automated verification, the verified program is annotated by the user to guide the verification tool: preconditions, postconditions, invariants, *etc*. Given this input, the verification tool generates proof obligations that are mostly automatically discharged. One may further distinguish two types of semi-automated systems: *foundational* and *non-foundational*.

In *non-foundational* automated verification, the tool and external solvers it may rely on are part of the trusted computing base. It is the most common approach and has been widely applied in the literature [Swamy et al., 2013; Müller et al., 2017; Jacobs et al., 2011; Denis et al., 2022; Astrauskas et al., 2022; Filliâtre and Paskevich, 2013; Lattuada et al., 2023; Pulte et al., 2023], including to OCaml by Cameleer [Pereira and Ravara, 2021], which uses the Gospel specification language [Charguéraud et al., 2019] and Why3 [Filliâtre and Paskevich, 2013].

In *foundational* automated verification, proofs are checked by a proof assistant so the automation does not have to be trusted. To our knowledge, it has been applied to

C [Sammler et al., 2021] and Rust [Gäher et al., 2024].

Physical equality. There is some literature in proof-assistant research on reflecting physical equality from the implementation language into the proof assistant, for optimization purposes: for example, exposing OCaml's physical equality as a predicate in Rocq lets us implement some memoization and sharing techniques in Rocq libraries. However, axiomatizing physical equality in the proof assistant is difficult and can result in inconsistencies.

The earlier discussions of this question that we know come from Jourdan's thesis [Jourdan, 2016] (chapter 9), also presented more succintly in [Braibant et al., 2014]. This work introduces the Jourdan condition: physical equality implies equality of values. Boulmé [2021] extends the treatment of physical equality in Rocq, integrating it in an "extraction monad" to control it more safely. There is also a discussion on similar optimizations in Lean in [Selsam et al., 2020].

The correctness of the axiomatization of physical equality depends on the type of the values being compared: axiomatizations are typically polymorphic on any type `A`, but their correctness depends on the specific `A` being considered. For example, it is easy to correctly characterize physical equality on natural numbers and other non-dependent types arising in Rocq verification projects. One difficulty in HeapLang and ZooLang is that they are untyped languages; in particular, their representation of `0` and `false` have the same type. However, our remark that physical equality (in OCaml) does not necessarily coincide with definitional equality (in Rocq) also applies to other Rocq types: our examples with an existential `Any` constructor (see Section 4.2.3) can be reproduced with Σ-types.

4.4 Future work

Relaxed memory model. Currently, the most important limitation of ZooLang is that it assumes a sequentially consistent memory model, whereas OCaml 5 has a relaxed memory model (see Section 2.2). As a result, our semantics does not capture all observable behaviors and therefore all correctness results are compromised. This choice has a pragmatic justification: we wanted to ensure that we could scale up verification of concurrent algorithms in the simpler setting of sequential consistency before moving to relaxed memory.

It should be noted that moving to relaxed memory is much simpler than for other languages like C because the OCaml 5 memory model is comparatively not very relaxed. Indeed, Mével et al. [2020] propose an Iris-based program logic for Multicore OCaml [Sivaramakrishnan et al., 2020] which Mével and Jourdan [2021] use to verify a fine-grained concurrent queue; they show that it is possible to adapt specifications and proofs in non-trivial but relatively straightforward way. This suggests that the transition is feasible and would not throw away our work; we plan to do it in the future.

Language features. ZooLang currently lacks many features that we also plan to support in the future: exceptions, algebraic effects [Sivaramakrishnan et al., 2021], modules, functors, threads[14]. Overall, this was not a significant limitation except for `Parabs` (see Chapter 10). Algebraic effects have been formalized by de Vilhena and Pottier [2021], who

[14] `https://ocaml.org/manual/5.4/api/Thread.html`

propose an Iris-based program logic; accordingly, it should not be difficult to introduce them in ZooLang.

Bounded integers. As most Iris languages, ZooLang features *unbounded* integers, which are unsound but much more pleasant to reason about than *bounded* machine integers. Moreover, as noted by Carbonneaux et al. [2022], programmers often make assumptions about integers that are difficult to formalize. At the very least, introducing bounded integers would result in a lot of noise. Further investigation is probably needed; we leave it for future work.

Chapter 5

Prophecy variables

In 2020, Jung et al. introduced *prophecy variables* in Iris. Essentially, prophecy variables — or *prophets*, as we will call them in this thesis — can be used to predict the future of the program execution and reason about it. They are key to handle *future-dependent linearization points* [Dongol and Derrick, 2014]: linearization points that may or may not occur at a given location in the code depending on a future observation. We will encounter several of them in Chapter 9 and Chapter 11.

ZooLang supports prophets through the `Proph` and `Resolve` expressions — as in HeapLang, the canonical Iris language. In OCaml, four primitives are recognized by `ocaml2zoo`:

- `Zoo.proph ()` is translated to:
  `Proph`;
- `Zoo.resolve_with` e_1 e_2 e_3 is translated to:
  `Resolve` e_1 e_2 e_3;
- `Zoo.resolve_silent` e_1 e_2 is translated to:
  `Resolve Skip` e_1 e_2 where `Skip` $\triangleq$ `(fun: <> => ()) ()`;
- `Zoo.resolve` e_1 e_2 is translated to:
  `let: "@tmp" :=` e_2 `in Resolve Skip` e_1 `"@tmp" ;; "@tmp"`.

To reason about prophets in the logic, we build four abstraction layers above the semantics of `Proph` and `Resolve` (given in Section 4.2.5). The first two layers come from Jung et al. [2020] while the last two are contributions of this thesis.

5.1 Primitive prophet

The first layer consists of *primitive prophets*. These prophets are primitive in the sense that they simply reflect the semantics of `Proph` and `Resolve` in the program logic. The corresponding reasoning rules are given in Figure 5.1.

The assertion model *pid* *prophs* represents the exclusive ownership of the prophet with identifier *pid*; *prophs* is the list of prophecies that must still be resolved.

WP-PROPH says that `Proph` allocates a new prophet with some unknown prophecies to be resolved. WP-RESOLVE says that `Resolve` e *pid* v *atomically* resolves the next prophecy of prophet *pid*: we learn that the prophecies before resolution *prophs* is non-empty and its head is the pair (res, v) where *res* is the evaluation of e.

$$\frac{\text{model}\ pid\ props_1 \qquad \text{model}\ pid\ props_2}{\text{False}}\ \textsc{prophet-model-exclusive}$$

$$\frac{\text{True}}{\text{Proph}\ ⨾\ \mathcal{E}}\bigg/\ pid.\ \exists\ props.\ \text{model}\ pid\ props\quad \textsc{wp-proph}$$

$$\frac{\text{to-val}\ e = \text{None} \qquad \text{model}\ pid\ props \qquad \text{wp}_{\mathcal{E}}\ e \left\{ \begin{array}{l} res.\ \forall\ props'.\\ \quad props = (res, v) :: props' \mathrel{-\!\!*} \\ \quad \text{model}\ pid\ props' \mathrel{-\!\!*} \\ \quad \Phi\ res \end{array} \right\}}{\text{wp}_{\mathcal{E}}\ \text{Resolve}\ e\ pid\ v\ \{\ \Phi\ \}}\ \textsc{wp-resolve}$$

(with side condition: atomic e)

Figure 5.1: Reasoning rules for primitive prophets

5.2 Typed prophet

The second layer consists of *typed prophets*. They are very similar to primitive prophets except prophecies are now typed. For convenience, we further distinguish two kinds of typed prophets: *normal* and *strong*. The difference is that normal prophets only predict the values provided to `Resolve` while strong prophets also predict the evaluations of the backing expressions as primitive prophets do. The corresponding reasoning rules are given in Figure 5.2.

The rules are essentially the same as before. The prophet must provide a type τ along with two functions of-val and to-val. to-val converts an inhabitant of τ to a value; TYPED-PROPHET-RESOLVE-SPEC and TYPED-STRONG-PROPHET-RESOLVE-SPEC rely on it to enforce that the prophecies are well-typed. of-val attempts to convert a value to τ; it is used internally. of-val and to-val must be compatible: of-val (to-val *proph*) = Some *proph*.

5.3 Wise prophet

The third layer consists of *wise prophets*. These prophets *remember* past prophecies. The corresponding reasoning rules are given in Figure 5.3.

We introduce four predicates: model, full, snapshot and lb.

model *pid* γ *past* *prophs* represents the exclusive ownership of the prophet with identifier *pid*; γ is the logical name of the prophet; *past* is the list of prophecies resolved so far; *prophs* is the list of prophecies that must still be resolved.

full γ *prophs* represents the list of all (resolved or not) prophecies associated to the prophet with name γ, as stated by WISE-PROPHET-FULL-VALID. It is persistent.

snapshot γ *past* *prophs* represents a persistent snapshot of the state of the prophet with name γ at some point in the past. WISE-PROPHET-SNAPSHOT-VALID allows to relate the current state of model to the past state of snapshot.

lb γ *lb* represents a persistent lower bound on the non-resolved prophecies of the

TYPED-PROPHET-MODEL-EXCLUSIVE

$$\frac{\mathsf{model}\ \mathit{pid}\ \mathit{prophs1} \qquad \mathsf{model}\ \mathit{pid}\ \mathit{prophs2}}{\mathsf{False}}$$

TYPED-PROPHET-PROPH-SPEC

$$\frac{\mathsf{True}}{\dfrac{\mathsf{Proph}}{\mathit{pid}.\ \exists\ \mathit{prophs}.\ \mathsf{model}\ \mathit{pid}\ \mathit{prophs}}}$$

TYPED-PROPHET-RESOLVE-SPEC

$$\frac{\begin{array}{c}\mathsf{atomic}\ e \qquad \mathsf{to\text{-}val}\ e = \mathsf{None} \qquad v = \mathit{prophet}.\mathsf{to\text{-}val}\ \mathit{proph} \\ \mathsf{model}\ \mathit{pid}\ \mathit{prophs} \qquad \mathsf{wp}_{\mathcal{E}}\ e \left\{ \begin{array}{l} w.\ \forall\ \mathit{prophs}'. \\ \quad \mathit{prophs} = \mathit{proph} :: \mathit{prophs}' \twoheadrightarrow \\ \quad \mathsf{model}\ \mathit{pid}\ \mathit{prophs}' \twoheadrightarrow \\ \quad \Phi\ w \end{array} \right\}\end{array}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{Resolve}\ e\ \mathit{pid}\ v\ \{\ \Phi\ \}}$$

TYPED-STRONG-PROPHET-RESOLVE-SPEC

$$\frac{\begin{array}{c}\mathsf{atomic}\ e \\ \mathsf{to\text{-}val}\ e = \mathsf{None} \qquad \mathsf{model}\ \mathit{pid}\ \mathit{prophs} \qquad \mathsf{wp}_{\mathcal{E}}\ e \left\{ \begin{array}{l} w.\ \exists\ \mathit{proph}. \\ \quad (w, v) = \mathit{prophet}.\mathsf{to\text{-}val}\ \mathit{proph} \ast \\ \quad\quad \forall\ \mathit{prophs}'. \\ \quad\quad \mathit{prophs} = \mathit{proph} :: \mathit{prophs}' \twoheadrightarrow \\ \quad\quad \mathsf{model}\ \mathit{pid}\ \mathit{prophs}' \twoheadrightarrow \\ \quad\quad \Phi\ w \end{array} \right\}\end{array}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{Resolve}\ e\ \mathit{pid}\ v\ \{\ \Phi\ \}}$$

Figure 5.2: Reasoning rules for typed prophets

$$\text{persistent}\ (\mathsf{full}\ \gamma\ \mathit{prophs}) \qquad \text{persistent}\ (\mathsf{snapshot}\ \gamma\ \mathit{past}\ \mathit{prophs}) \qquad \text{persistent}\ (\mathsf{lb}\ \gamma\ \mathit{lb})$$

WISE-PROPHET-MODEL-EXCLUSIVE
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma_1\ \mathit{past}_1\ \mathit{prophs1} \qquad \mathsf{model}\ \mathit{pid}\ \gamma_2\ \mathit{past}_2\ \mathit{prophs2}}{\mathsf{False}}$$

WISE-PROPHET-FULL-GET
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs}}{\mathsf{full}\ \gamma\ (\mathit{past} +\!\!+\ \mathit{prophs})}$$

WISE-PROPHET-FULL-VALID
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs}_1 \qquad \mathsf{full}\ \gamma\ \mathit{prophs}_2}{\mathit{prophs}_2 = \mathit{past} +\!\!+\ \mathit{prophs}_1}$$

WISE-PROPHET-FULL-AGREE
$$\frac{\mathsf{full}\ \gamma\ \mathit{prophs}_1 \qquad \mathsf{full}\ \gamma\ \mathit{prophs}_2}{\mathit{prophs}_1 = \mathit{prophs}_2}$$

WISE-PROPHET-SNAPSHOT-GET
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs}}{\mathsf{snapshot}\ \gamma\ \mathit{past}\ \mathit{prophs}}$$

WISE-PROPHET-SNAPSHOT-VALID
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{past}_1\ \mathit{prophs}_1 \qquad \mathsf{snapshot}\ \gamma\ \mathit{past}_2\ \mathit{prophs}_2}{\exists\ \mathit{past}_3.\ \mathit{past}_1 = \mathit{past}_2 +\!\!+\ \mathit{past}_3 * \mathit{prophs}_2 = \mathit{past}_3 +\!\!+\ \mathit{prophs}_1}$$

WISE-PROPHET-LB-GET
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs}}{\mathsf{lb}\ \gamma\ \mathit{prophs}}$$

WISE-PROPHET-LB-VALID
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs} \qquad \mathsf{lb}\ \gamma\ \mathit{lb}}{\exists\ \mathit{past}_1\ \mathit{past}_2.\ \mathit{past} = \mathit{past}_1 +\!\!+\ \mathit{past}_2 * \mathit{lb} = \mathit{past}_2 +\!\!+\ \mathit{prophs}}$$

WISE-PROPHET-PROPH-SPEC
$$\frac{\dfrac{\mathsf{True}}{\mathsf{Proph}}}{\mathit{pid}.\ \exists\ \gamma\ \mathit{prophs}.\ \mathsf{model}\ \mathit{pid}\ \gamma\ []\ \mathit{prophs}}$$

WISE-PROPHET-RESOLVE-SPEC
$$\frac{\begin{array}{c}\mathsf{atomic}\ e \qquad \mathsf{to\text{-}val}\ e = \mathsf{None} \qquad v = \mathit{prophet}.\mathsf{to\text{-}val}\ \mathit{proph} \\ \mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs} \qquad \mathsf{wp}_{\mathcal{E}}\ e \left\{ \begin{array}{l} w.\ \forall\ \mathit{prophs}'. \\ \quad \mathit{prophs} = \mathit{proph} :: \mathit{prophs}' \twoheadrightarrow \\ \quad \mathsf{model}\ \mathit{pid}\ \gamma\ (\mathit{past} +\!\!+\ [\mathit{proph}])\ \mathit{prophs}' \twoheadrightarrow \\ \quad \Phi\ w \end{array} \right\}\end{array}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{Resolve}\ e\ \mathit{pid}\ v\ \{\ \Phi\ \}}$$

WISE-STRONG-PROPHET-RESOLVE-SPEC
$$\frac{\begin{array}{c}\mathsf{atomic}\ e \qquad \mathsf{to\text{-}val}\ e = \mathsf{None} \\ \mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{past}\ \mathit{prophs} \qquad \mathsf{wp}_{\mathcal{E}}\ e \left\{ \begin{array}{l} w.\ \exists\ \mathit{proph}. \\ \quad (w, v) = \mathit{prophet}.\mathsf{to\text{-}val}\ \mathit{proph} * \\ \quad\quad \forall\ \mathit{prophs}'. \\ \quad\quad \mathit{prophs} = \mathit{proph} :: \mathit{prophs}' \twoheadrightarrow \\ \quad\quad \mathsf{model}\ \mathit{pid}\ \gamma\ (\mathit{past} +\!\!+\ [\mathit{proph}])\ \mathit{prophs}' \twoheadrightarrow \\ \quad\quad \Phi\ w \end{array} \right\}\end{array}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{Resolve}\ e\ \mathit{pid}\ v\ \{\ \Phi\ \}}$$

Figure 5.3: Reasoning rules for wise prophets

prophet with name γ. In particular, as stated by WISE-PROPHET-LB-VALID, the list of currently non-resolved prophecies carried by model is always a suffix of lb.

WISE-PROPHET-RESOLVE-SPEC and WISE-STRONG-PROPHET-RESOLVE-SPEC are the same as before, except we also update the list of resolved prophecies after resolution.

5.4 Multiplexed prophet

The fourth layer consists of *multiplexed prophets*. Essentially, they allow to combine different prophets, each operating at a fixed index. They were made to handle the case when a single prophet is used to make independent predictions, as in Section 9.7. The corresponding reasoning rules are given in Figure 5.4.

The predicates and rules are basically the same as before, except that (1) model now carries sequences of lists of prophecies — one past and one future per index — and (2) full, snapshot and lb are parameterized with an index.

Importantly, the third argument provided to `Resolve` in WISE-PROPHETS-RESOLVE-SPEC and WISE-STRONG-PROPHETS-RESOLVE-SPEC must be a pair of an index and a prophecy value. Resolution happens only at the given index, meaning the prophecies at other indices are unchanged.

Note that we could generalize this abstraction to non-integer keys. In other words, we could replace sequences with functions of type $X \to \tau$, where τ is the prophecy type, and indices with inhabitants of X. In practice, however, we never needed such generalization.

5.5 Limitation

As noted by Jung et al., prophecy variables suffer from an important limitation: prophecy resolution is not modular because it requires a *physically* atomic backing expression. Most of the time, in concurrent algorithms, this is not a problem because prophecy resolution is used only with primitive memory operations.

However, in Section 9.4, we encounter the case where the backing expression is *logically* atomic, in the sense that it admits an atomic specification, but not physically atomic. More precisely, the backing expression is an operation on an infinite array that consists in acquiring a lock and updating the array while holding this lock. Therefore, exposing the implementation does not help; `Resolve` cannot be used *externally*. Yet, it can be used *internally*. In other words, it is possible to implement the operation so that it internally performs prophecy resolution. This leads to a complex specification, as we describe in Section 6.11.

5.6 Erasure

5.6.1 Erasure in HeapLang

Jung et al. [2020] proved that prophecy variables can be erased in HeapLang, in the sense that eliminating prophecy-related expressions and values preserves safety. Con-

$$\mathsf{persistent}\ (\mathsf{full}\ \gamma\ i\ \mathit{prophs}) \qquad \mathsf{persistent}\ (\mathsf{snapshot}\ \gamma\ i\ \mathit{past}\ \mathit{prophs}) \qquad \mathsf{persistent}\ (\mathsf{lb}\ \gamma\ i\ \mathit{lb})$$

WISE-PROPHETS-MODEL-EXCLUSIVE
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma_1\ \mathit{pasts}_1\ \mathit{prophss1} \qquad \mathsf{model}\ \mathit{pid}\ \gamma_2\ \mathit{pasts}_2\ \mathit{prophss2}}{\mathsf{False}}$$

WISE-PROPHETS-FULL-GET
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss}}{\mathsf{full}\ \gamma\ i\ (\mathit{pasts}\ i + \mathit{prophss}\ i)}$$

WISE-PROPHETS-FULL-VALID
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss} \qquad \mathsf{full}\ \gamma\ i\ \mathit{prophs}}{\mathit{prophs} = \mathit{pasts}\ i + \mathit{prophss}\ i}$$

WISE-PROPHETS-FULL-AGREE
$$\frac{\mathsf{full}\ \gamma\ i\ \mathit{prophs}_1 \qquad \mathsf{full}\ \gamma\ i\ \mathit{prophs}_2}{\mathit{prophs}_1 = \mathit{prophs}_2}$$

WISE-PROPHETS-SNAPSHOT-GET
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss}}{\mathsf{snapshot}\ \gamma\ (\mathit{pasts}\ i)\ (\mathit{prophss}\ i)}$$

WISE-PROPHETS-SNAPSHOT-VALID
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss} \qquad \mathsf{snapshot}\ \gamma\ i\ (\mathit{pasts}\ i)\ (\mathit{prophss}\ i)}{\exists\ \mathit{past}'.\ \mathit{pasts}\ i = \mathit{past} + \mathit{past}' * \mathit{prophs} = \mathit{past}' + \mathit{prophss}\ i}$$

WISE-PROPHETS-LB-GET
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss}}{\mathsf{lb}\ \gamma\ i\ (\mathit{prophss}\ i)}$$

WISE-PROPHETS-LB-VALID
$$\frac{\mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss} \qquad \mathsf{lb}\ \gamma\ i\ \mathit{lb}}{\exists\ \mathit{past}_1\ \mathit{past}_2.\ \mathit{pasts}\ i = \mathit{past}_1 + \mathit{past}_2 * \mathit{lb} = \mathit{past}_2 + \mathit{prophss}\ i}$$

WISE-PROPHETS-PROPH-SPEC
$$\frac{\dfrac{\mathsf{True}}{\mathsf{Proph}}}{\mathit{pid}.\ \exists\ \gamma\ \mathit{prophss}.\ \mathsf{model}\ \mathit{pid}\ \gamma\ (\lambda\ \_.\ [])\ \mathit{prophss}}$$

WISE-PROPHETS-RESOLVE-SPEC
$$\frac{\begin{array}{c}\mathsf{atomic}\ e \\ \mathsf{to\text{-}val}\ e = \mathsf{None} \qquad v = \mathit{prophet}.\mathsf{to\text{-}val}\ \mathit{proph} \qquad \mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss} \\ \mathsf{wp}_{\mathcal{E}}\ e \left\{ \begin{array}{l} w.\ \forall\ \mathit{prophs}. \\ \quad \mathit{prophss}\ i = \mathit{proph} :: \mathit{prophs} \mathrel{-\!\!*} \\ \quad \mathsf{model}\ \mathit{pid}\ \gamma\ (\mathsf{alter}\ (\cdot + [\mathit{proph}])\ i\ \mathit{pasts})\ (\mathit{prophss}\ [i \mapsto \mathit{prophs}]) \mathrel{-\!\!*} \\ \quad \Phi\ w \end{array} \right\}\end{array}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{Resolve}\ e\ \mathit{pid}\ (i, v)\ \{\ \Phi\ \}}$$

WISE-STRONG-PROPHETS-RESOLVE-SPEC
$$\frac{\begin{array}{c}\mathsf{atomic}\ e \qquad \mathsf{to\text{-}val}\ e = \mathsf{None} \qquad \mathsf{model}\ \mathit{pid}\ \gamma\ \mathit{pasts}\ \mathit{prophss} \\ \mathsf{wp}_{\mathcal{E}}\ e \left\{ \begin{array}{l} w.\ \exists\ \mathit{proph}. \\ \quad (w, v) = \mathit{prophet}.\mathsf{to\text{-}val}\ \mathit{proph} * \\ \quad\ \forall\ \mathit{prophs}. \\ \quad\ \mathit{prophss}\ i = \mathit{proph} :: \mathit{prophs} \mathrel{-\!\!*} \\ \quad\ \mathsf{model}\ \mathit{pid}\ \gamma\ (\mathsf{alter}\ (\cdot + [\mathit{proph}])\ i\ \mathit{pasts})\ (\mathit{prophss}\ [i \mapsto \mathit{prophs}]) \mathrel{-\!\!*} \\ \quad\ \Phi\ w \end{array} \right\}\end{array}}{\mathsf{wp}_{\mathcal{E}}\ \mathsf{Resolve}\ e\ \mathit{pid}\ (i, v)\ \{\ \Phi\ \}}$$

Figure 5.4: Reasoning rules for multiplexed prophets

cretely, erasure proceeds structurally with the following base cases:

$$\begin{array}{lcl} \mathsf{erase}\ \mathsf{Proph} & \triangleq & \mathsf{ValFun}\ \mathsf{BinderAnon}\ \mathsf{LitPoison} \\ \mathsf{erase}\ (\mathsf{Resolve}\ e_1\ e_2\ e_3) & \triangleq & \mathsf{Proj}_0\ (\mathsf{Proj}_0\ (\mathsf{Tuple}\ [\mathsf{erase}\ e_1; \mathsf{erase}\ e_2; \mathsf{erase}\ e_3])) \\ \mathsf{erase\text{-}literal}\ (\mathsf{LitProph}\ pid) & \triangleq & \mathsf{LitPoison} \end{array}$$

A delicate point is the interaction between prophecy variables and physical equality. Indeed, erasure is not injective — it makes things "more equal". Consequently, physical comparisons that always fail in the original program may succeed in the erased program, thereby breaking safety.

To cope with this point, HeapLang restricts the semantics of physical equality so that prophecies ($\mathsf{LitProph}\ pid$), poison ($\mathsf{LitPoison}$) and functions ($\mathsf{ValRecs}\ i\ \overline{rec}$) cannot be compared to any value, either directly or indirectly (through immutable blocks). This makes erasure injective and allows proving safety preservation.

5.6.2 Erasure in ZooLang

Unfortunately, we found this restriction to be impractical; for example, in Chapter 8, we need to compare (generative) immutable blocks containing functions. Thus, in ZooLang, physical equality is not restricted but comes with a special semantics (see Section 4.2.3). As a result, the problem of erasure injectivity remains, but this time with respect to value similarity.

For prophecy values, the problem can easily be solved by forbidding the original program to use poison. For functional values, it can also be solved by providing no guarantee for function bodies. For non-generative immutable blocks, there is no problem because the semantics is very weak. For generative immutable blocks, there really is a problem since value similarity guarantees that the block fields are equal in Rocq; therefore, injectivity with respect to similarity requires injectivity with respect to Rocq equality. We envision two solutions.

Solution 1. A straightforward but somewhat unsatisfactory solution is to introduce a special construct to erase $\mathsf{Resolve}$ expressions. Crucially, as $\mathsf{LitPoison}$, this construct would be forbidden in the original program. Although superficial, this solution suffices to show safety preservation for a *prophecy-inert* program, which is arguably the most important.

Solution 2. Another, more complex solution is to decompose erasure into two steps: (1) erase generative blocks, replacing them with immutable blocks; (2) erase prophecy variables normally thanks to injectivity with respect to value similarity. Interestingly, the first step would also formally justify our high-level semantics by providing a low-level semantics it translates into. We leave it for future work.

Chapter 6

Standard data structures

To save users from reinventing the wheel, Zoo comes with a library of verified standard data structures — more or less a subset of the OCaml standard library. Most of these data structures[1] are completely reimplemented in Zoo and axiom-free, including the `Array`[2] module. We claim that the proven specifications are modular and practical. In fact, most data structures have already been used to verify more complex ones.

In this chapter, we present the most important parts, including those that will be needed in the following chapters. The full library can be found in the accompanying mechanization.

6.1 List

We provide a verified implementation of (more or less) a subset of `Stdlib.List`[3]. Especially, we developed an extensive collection of flexible specifications for iterators (`iter`, `map`, `fold_left`, `fold_right`).

6.2 Array

We provide a verified implementation of (more or less) a subset of `Stdlib.Array`[4]. Similarly to `Stdlib.List`, we developed a collection of flexible specifications for iterators, including atomic specifications.

Remarkably, our formalization features different (fractional) predicates to express the ownership of either an entire array, a slice or even a circular slice — we used it to verify algorithms involving circular arrays, *e.g.* Chase-Lev working-stealing deque [Chase and Lev, 2005] (see Section 9.7).

[1] For practical reasons, to make them completely opaque, we chose to axiomatize a few functions from the `Domain` and `Random` modules. They could trivially be realized in Zoo.

[2] Our implementation of the `Array` module is compatible with the standard one. In particular, it uses the same low-level value representation.

[3] `https://ocaml.org/manual/5.3/api/List.html`

[4] `https://ocaml.org/manual/5.3/api/Array.html`

$$\textsc{random-state-create-spec}\quad \dfrac{\dfrac{\mathsf{True}}{\texttt{create ()}}}{t.\ \mathsf{model}\ t}$$

$$\textsc{random-state-int-spec}\quad \dfrac{\dfrac{0 < ub \ast \mathsf{model}\ t}{\texttt{int}\ t\ ub}}{n.\ 0 \leq n < ub \ast \mathsf{model}\ t}$$

$$\textsc{random-state-int-in-range-spec}\quad \dfrac{\dfrac{lb < ub \ast \mathsf{model}\ t}{\texttt{int\_in\_range}\ t\ lb\ ub}}{n.\ lb \leq n < ub \ast \mathsf{model}\ t}$$

Figure 6.1: `Random_state`: Specification (excerpt)

6.3 Dynamic array

We verified two implementations of a subset of `Stdlib.Dynarray`[5] (introduced in OCaml 5.2): (1) an efficient but *unsafe* version and (2) a less efficient but *safe* version. We explain these notions in Chapter 12.

6.4 Random generator

We provide an axiomatization of a subset of `Stdlib.Random`[6] and `Stdlib.Random.State`[7]. For instance, the specification of our module `Random_state` is given in Figure 6.1. The assertion model t represents the ownership of the pseudorandom number generator t; it is required and returned by the `int` and `int_in_range` functions in an imperative fashion.

6.5 Random round

We provide a verified `Random_round` module that allows iterating over $[\![0; sz)$ in a random order, for a given sz. This device is used internally by the parallel task scheduler of Chapter 10 to randomly pick a domain to steal from during *load balancing*. Its specification is given is Figure 6.2.

The assertion model $t\ sz\ prevs$ represents the ownership of t, where sz is the size of the iterated interval and $prevs$ the list of already visited elements. The main operation is `next`, which randomly chooses a non-visited element and adds it to $prevs$ (RANDOM-ROUND-NEXT-SPEC). At any time, iteration can be restarted using `reset` (RANDOM-ROUND-RESET-SPEC).

We also provide a simpler specification where model only maintains the number of non-visited elements; it is the one used in Chapter 10.

[5]`https://ocaml.org/manual/5.3/api/Dynarray.html`
[6]`https://ocaml.org/manual/5.3/api/Random.html`
[7]`https://ocaml.org/manual/5.3/api/Random.State.html`

$$\textsc{random-round-create-spec}\quad \dfrac{\dfrac{0 \le sz}{\texttt{create}\ sz}}{t.\ \textsf{model}\ t\ sz\ []}$$

$$\textsc{random-round-next-spec}\quad \dfrac{\dfrac{\textsf{length}\ prevs \neq sz \ast \quad \textsf{model}\ t\ sz\ prevs}{\texttt{next}\ t}}{n.\ 0 \le n < sz \ast \quad n \notin prevs \ast \quad \textsf{model}\ t\ sz\ (prevs +\!\!+ [n])}$$

$$\textsc{random-round-reset-spec}\quad \dfrac{\dfrac{\textsf{model}\ t\ sz\ prevs}{\texttt{reset}\ t}}{\texttt{()}.\ \textsf{model}\ t\ sz\ []}$$

Figure 6.2: `Random_round`: Specification

Although we did not need it, the specification we presented could be improved. In particular, it should be possible to determine the order in which the elements are chosen in advance using a prophecy variable (see Chapter 5). This order would materialize as an additional parameter of model and be consumed into *prevs* during iteration.

6.6 Domain

We reimplemented and verified a subset of `Stdlib.Domain`[8] — except for a few minor functions that we axiomatized. Its specification is given in Figure 6.3.

The assertion model t Ψ represents the ownership of domain t, or more accurately the right to call `join` to obtain Ψ v for some v (DOMAIN-JOIN-SPEC). As `join` consumes model, it can be called only once. This restriction is justified by the fact that a child domain is typically joined only by its parent domain in a fork-join fashion — this is the case, for example, in parallel schedulers. Alternatively, we could have used the same mechanism as in Section 6.10 to separate domain termination and output, thereby allowing calling `join` multiple times.

The rest of the specification deals with *domain-local storage*[9] (DLS). This part is interesting because DLS *keys* are generated dynamically.

The persistent assertion key *key* Ψ represents the knowledge that *key* is a valid DLS key whose initializer produces values satisfying Ψ. It can be obtained by calling `local_new` (DOMAIN-LOCAL-NEW-SPEC).

local-init *tid key* asserts that the DLS key *key* of domain *tid* has been logically initialized — but not necessarily physically initialized. It can be obtained through the rule DOMAIN-LOCAL-GET-KEY.

The assertion local *tid keys* represents the ownership of the local storage attached to domain *tid*. *keys* is the set of logically initialized DLS keys; when a new domain is spawned, it is empty (DOMAIN-SPAWN-SPEC).

The assertion local-pointsto *tid key q v* represents the fractional ownership of the DLS key *key* of domain *tid* and the knowledge that it currently contains v. It can be

[8] `https://ocaml.org/manual/5.3/api/Domain.html`
[9] `https://ocaml.org/manual/5.3/api/Domain.DLS.html`

$$\mathsf{persistent}\ (\mathsf{key}\ key\ \Psi)$$

DOMAIN-LOCAL-GET-KEY

$$\frac{key \notin keys \qquad \mathsf{local}\ tid\ keys \qquad \mathsf{key}\ key\ \Psi}{\dot{\Rrightarrow}\ \mathsf{local}\ tid\ (keys \uplus \{key\}) * \mathsf{local\text{-}init}\ tid\ key}$$

DOMAIN-LOCAL-POINTSTO-AGREE

$$\frac{\begin{array}{l}\mathsf{local\text{-}pointsto}\ tid\ key\ q_1\ v_1\\ \mathsf{local\text{-}pointsto}\ tid\ key\ q_2\ v_2\end{array}}{v_1 = v_2}$$

DOMAIN-LOCAL-POINTSTO-EXCLUSIVE

$$\frac{\begin{array}{l}\mathsf{local\text{-}pointsto}\ tid\ key\ 1\ v_1\\ \mathsf{local\text{-}pointsto}\ tid\ key\ q_2\ v_2\end{array}}{\mathsf{False}}$$

DOMAIN-SPAWN-SPEC

$$\begin{array}{c}\forall\ tid.\ \mathsf{local}\ tid\ \emptyset \twoheadrightarrow \mathsf{wp}\ fn\ ()\ ⨟\ tid\ \{\ \Psi\ \}\\ \hline \texttt{spawn}\ fn\\ \hline t.\ \mathsf{model}\ t\ \Psi\end{array}$$

DOMAIN-JOIN-SPEC

$$\begin{array}{c}\mathsf{model}\ t\ \Psi\\ \hline \texttt{join}\ t\\ \hline \Psi\end{array}$$

DOMAIN-LOCAL-NEW-SPEC

$$\begin{array}{c}\{\ \mathsf{True}\ \}\ fn\ ()\ \{\ \Psi\ \} *\\ \mathop{\Huge *}_{key \in keys}\ \exists\ \mathrm{X}.\ \mathsf{key}\ key\ \mathrm{X}\\ \hline \texttt{local\_new}\ fn\\ \hline key.\ \mathsf{key}\ key\ \Psi *\\ key \notin keys\end{array}$$

DOMAIN-LOCAL-GET-SPEC-INIT

$$\begin{array}{c}\mathsf{local}\ tid\ keys *\\ \mathsf{key}\ key\ \Psi *\\ \mathsf{local\text{-}init}\ tid\ key\\ \hline \texttt{local\_get}\ key\ ⨟\ tid\\ \hline \begin{array}{l}v.\ \mathsf{local}\ tid\ keys *\\ \quad \mathsf{local\text{-}pointsto}\ tid\ key\ 1\ v *\\ \quad \Psi\ v\end{array}\end{array}$$

DOMAIN-LOCAL-GET-SPEC-POINTSTO

$$\begin{array}{c}\mathsf{local}\ tid\ keys *\\ \mathsf{local\text{-}pointsto}\ tid\ key\ q\ v\\ \hline \texttt{local\_get}\ key\ ⨟\ tid\\ \hline \begin{array}{l}res.\ res = v *\\ \quad \mathsf{local}\ tid\ keys *\\ \quad \mathsf{local\text{-}pointsto}\ tid\ key\ q\ v\end{array}\end{array}$$

DOMAIN-LOCAL-GET-SPEC-POINTSTOPRED

$$\begin{array}{c}\mathsf{local}\ tid\ keys *\\ \mathsf{local\text{-}pointstopred}\ tid\ key\ \Psi\\ \hline \texttt{local\_get}\ key\ ⨟\ tid\\ \hline \begin{array}{l}v.\ \mathsf{local}\ tid\ keys *\\ \quad \mathsf{local\text{-}pointsto}\ tid\ key\ 1\ v *\\ \quad \Psi\ v\end{array}\end{array}$$

DOMAIN-LOCAL-SET-SPEC-INIT

$$\begin{array}{c}\mathsf{local}\ tid\ keys *\\ \mathsf{key}\ key\ \Psi *\\ \mathsf{local\text{-}init}\ tid\ key\\ \hline \texttt{local\_set}\ key\ v\ ⨟\ tid\\ \hline \begin{array}{l}().\ \mathsf{local}\ tid\ keys *\\ \quad \mathsf{local\text{-}pointsto}\ tid\ key\ 1\ v\end{array}\end{array}$$

DOMAIN-LOCAL-SET-SPEC-POINTSTO

$$\begin{array}{c}\mathsf{local}\ tid\ keys *\\ \mathsf{local\text{-}pointsto}\ tid\ key\ 1\ w\\ \hline \texttt{local\_set}\ key\ v\ ⨟\ tid\\ \hline \begin{array}{l}().\ \mathsf{local}\ tid\ keys *\\ \quad \mathsf{local\text{-}pointsto}\ tid\ key\ 1\ v\end{array}\end{array}$$

DOMAIN-LOCAL-SET-SPEC-POINTSTOPRED

$$\begin{array}{c}\mathsf{local}\ tid\ keys *\\ \mathsf{local\text{-}pointstopred}\ tid\ key\ \Psi\\ \hline \texttt{local\_set}\ key\ v\ ⨟\ tid\\ \hline \begin{array}{l}().\ \mathsf{local}\ tid\ keys *\\ \quad \mathsf{local\text{-}pointsto}\ tid\ key\ 1\ v\end{array}\end{array}$$

Figure 6.3: `Domain`: Specification (excerpt)

used to read (DOMAIN-LOCAL-GET-SPEC-POINTSTO) and write (DOMAIN-LOCAL-SET-SPEC-POINTSTO) to *key* similarly to the points-to predicate for normal storage. It is obtained after reading (DOMAIN-LOCAL-GET-SPEC-INIT) or writing to (DOMAIN-LOCAL-SET-SPEC-INIT) *key* for the first time, which requires local-init. In summary, the user has to first logically initialize a key and then access it once before obtaining the corresponding local-pointsto and thereby fine-grained control over the key.

In practice, this two-step procedure is inconvenient because, *e.g.*, a function that accesses a DLS key may be called in a context where the status of the key is not determined. In other words, such a function should expect either local-init or local-pointsto. To address this issue, we define the local-pointstopred predicate:

$$\mathsf{local\text{-}pointstopred}\ tid\ key\ \Psi \triangleq \bigvee \begin{bmatrix} \mathsf{local\text{-}init}\ tid\ key * \mathsf{key}\ key\ \Psi \\ \exists\, v.\, \mathsf{local\text{-}pointsto}\ tid\ key\ 1\ v * \Psi\ v \end{bmatrix}$$

Essentially, local-pointstopred *tid key* Ψ represents the full ownership of the DLS key *key* of domain *tid* and the knowledge that it contains a value satisfying Ψ. Alternatively, local-pointstopred may be seen as a degenerated full local-pointsto. In terms of specifications, it behaves exactly like local-init (DOMAIN-LOCAL-GET-SPEC-POINTSTOPRED, DOMAIN-LOCAL-SET-SPEC-POINTSTOPRED).

So far, we kept quiet about one major limitation of our specification: the *freshness condition* of DOMAIN-LOCAL-GET-KEY. Indeed, this rule requires the given key to be different from all the already initialized keys. DOMAIN-LOCAL-NEW-SPEC provides a basic way to differentiate keys: when a key is created, we learn that it is different from any *given* preexisting key. Crucially, this is the only way to do so, which significantly limits the modularity of the approach.

Remarkably, though, this approach is flexible enough to handle interesting cases. For example, consider the parallel LU decomposition[10] implemented in `Domainslib`. The scheduler, which may internally use DLS, is given tasks that rely on a global DLS key. The verification could proceed as follows: (0) we assume that no DLS key has been logically initialized in the current domain, materialized by local *tid* $\emptyset$; (1) the global key is created, producing key; (2) when the scheduler is created, it first allocates its own DLS keys that are proven to be different from the global key; (3) then, for each spawned domain, the scheduler logically initializes all the keys — which is possible because we know they are distinct —, reserves the obtained local-pointstopred of the global key for the tasks and keeps the rest.

6.7 Mutex

We provide a verified implementation of `Stdlib.Mutex`[11]. The specification, given in Figure 6.4, is mostly standard.

The persistent assertion inv t P represents the knowledge that t is a valid mutex protecting the resource P. It is returned by `create` (MUTEX-CREATE-SPEC) and required by all operations.

[10] `https://github.com/ocaml-multicore/domainslib/blob/main/test/LU_decomposition_multicore.ml`

[11] `https://ocaml.org/manual/5.3/api/Mutex.html`

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ P)$$

$$\dfrac{\substack{\textsc{mutex-init-to-inv}\\ \mathsf{init}\ t \qquad \triangleright P}}{\Rrightarrow \mathsf{inv}\ t\ P} \qquad \dfrac{\substack{\textsc{mutex-locked-exclusive}\\ \mathsf{locked}\ t \qquad \mathsf{locked}\ t}}{\mathsf{False}}$$

$$\textsc{mutex-create-spec}\quad \begin{array}{c} P \\ \hline \texttt{create ()} \\ \hline t.\ \mathsf{inv}\ t\ P \end{array} \qquad \textsc{mutex-create-spec-init}\quad \begin{array}{c} \mathsf{True} \\ \hline \texttt{create ()} \\ \hline t.\ \mathsf{init}\ t \end{array} \qquad \textsc{mutex-lock-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ P \\ \hline \texttt{lock}\ t \\ \hline ().\ \mathsf{locked}\ t * P \end{array}$$

$$\textsc{mutex-unlock-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ \Phi * \mathsf{locked}\ t * P \\ \hline \texttt{unlock}\ t \\ \hline ().\ \mathsf{True} \end{array} \qquad \textsc{mutex-synchronize-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ P \\ \hline \texttt{synchronize}\ t \\ \hline ().\ \mathsf{True} \end{array}$$

$$\textsc{mutex-protect-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ P * \left(\begin{array}{l}\mathsf{locked}\ t \twoheadrightarrow P \twoheadrightarrow \\ \mathsf{wp}\ \mathit{fn}\ ()\ \{\, v.\ \mathsf{locked}\ t * P * \Psi\ v \,\}\end{array}\right) \\ \hline \texttt{protect}\ t\ \mathit{fn} \\ \hline \Psi \end{array}$$

Figure 6.4: `Mutex`: Specification

The assertion init t represents the ownership of an *uninitialized* mutex t, *i.e.* a mutex whose protected resource is not yet determined. It is also returned by `create` (MUTEX-CREATE-SPEC-INIT) and can be converted to inv through MUTEX-INIT-TO-INV. As a result, it allows delayed initialization. This is needed when the protected resource depends on resources available only after the mutex creation, *e.g.* in our implementation of infinite arrays (see Section 6.11).

The exclusive assertion locked t represents the temporary acquisition of the mutex by the current domain. It is returned by `lock` (MUTEX-LOCK-SPEC) — marking the beginning of the critical section — and required by `unlock` (MUTEX-UNLOCK-SPEC) — marking the end.

The `synchronize` operation simply locks and unlocks the mutex, thereby performing synchronization (see Section 2.2). In the sequentially consistent memory model that we adopted, it is basically useless and its specification (MUTEX-SYNCHRONIZE-SPEC) is trivial. In a relaxed memory model, however, the specification should be expressive enough to support transferring memory views.

6.8 Semaphore

We provide a verified implementation of `Stdlib.Semaphore`[12], a generalization of `Stdlib.Mutex` allowing more than one domain in the critical section. Its specification is similar.

6.9 Condition

We provide a verified implementation of `Stdlib.Condition`[13], *i.e. condition variables*. Essentially, a condition variable allows blocking a domain until someone sends a *notification* or a *spurious wakeup* occurs. The specification is given in Figure 6.5.

The persistent assertion inv t represents the knowledge that t is a valid condition variable. Its is returned by `create` (CONDITION-CREATE-SPEC) and required by all operations.

`notify` (CONDITION-NOTIFY-SPEC) and `notify_all` (CONDITION-NOTIFY-ALL-SPEC) send a notification to respectively one and all waiting domains, if any.

A domain can wait for a notification using `wait` t mtx while holding the mutex mtx (CONDITION-WAIT-SPEC); after the call, mtx is held.

`wait_until` t mtx $pred$ works similarly but also waits until the predicate $pred$ returns `true` (CONDITION-WAIT-UNTIL-SPEC). It is a higher-order operation; the precondition requires the ability to call $pred$ multiple times while holding mtx. For flexibility, the specification is parameterized with a predicate Ψ; Ψ false represents the resources needed and maintained by $pred$ in addition to those protected by mtx; Ψ true represents the final resources, obtained when $pred$ returns `true`.

$$\mathsf{persistent}\ (\mathsf{inv}\ t)$$

$$\textsc{condition-create-spec}\quad \frac{\dfrac{\mathsf{True}}{\texttt{create ()}}}{t.\ \mathsf{inv}\ t}$$

$$\textsc{condition-notify-spec}\quad \frac{\dfrac{\mathsf{inv}\ t}{\texttt{notify}\ t}}{().\ \mathsf{True}}$$

$$\textsc{condition-notify-all-spec}\quad \frac{\dfrac{\mathsf{inv}\ t}{\texttt{notify\_all}\ t}}{().\ \mathsf{True}}$$

$$\textsc{condition-wait-spec}\quad \frac{\dfrac{\begin{array}{l}\mathsf{inv}\ t\ *\\ \mathsf{mutex.inv}\ mtx\ P\ *\\ \mathsf{mutex.locked}\ mtx\ *\\ P\end{array}}{\texttt{wait}\ t\ mtx}}{\begin{array}{l}().\ \mathsf{mutex.locked}\ mtx\ *\\ \quad P\end{array}}$$

$$\textsc{condition-wait-until-spec}\quad \frac{\dfrac{\begin{array}{l}\mathsf{inv}\ t\ *\\ \mathsf{mutex.inv}\ mtx\ P\ *\\ \mathsf{mutex.locked}\ mtx\ *\\ P\ *\\ \Psi\ \mathsf{false}\ *\\ \left(\dfrac{\dfrac{\begin{array}{l}\mathsf{mutex.locked}\ mtx\ *\\ P\ *\\ \Psi\ \mathsf{false}\end{array}}{pred\ ()}}{\begin{array}{l}b.\ \mathsf{mutex.locked}\ mtx\ *\\ \quad (\textbf{if}\ b\ \textbf{then}\ \mathsf{True}\ \textbf{else}\ P)\ *\\ \quad \Psi\ b\end{array}}\right)\end{array}}{\texttt{wait\_until}\ t\ mtx\ pred}}{\begin{array}{l}().\ \mathsf{mutex.locked}\ mtx\ *\\ \quad \Psi\ \mathsf{true}\end{array}}$$

Figure 6.5: `Condition`: Specification (excerpt)

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ \Psi\ \Xi\ \Omega) \qquad\qquad \mathsf{persistent}\ (\mathsf{result}\ t\ v)$$

$$\textsc{ivar-producer-exclusive}\ \frac{\mathsf{producer}\ t \quad \mathsf{producer}\ t}{\mathsf{False}}$$

$$\textsc{ivar-consumer-divide}\ \frac{\mathsf{inv}\ t\ \Psi\ \Xi\ \Omega \quad \mathsf{consumer}\ t\ \mathrm{X} \quad \forall\ v.\ \mathrm{X}\ v \twoheadrightarrow \mathop{\Large\ast}_{\mathrm{X} \in \mathrm{X}s} \mathrm{X}\ v}{\Rrightarrow \mathop{\Large\ast}_{\mathrm{X} \in \mathrm{X}s} \mathsf{consumer}\ t\ \mathrm{X}}$$

$$\textsc{ivar-result-agree}\ \frac{\mathsf{result}\ t\ v_1 \quad \mathsf{result}\ t\ v_2}{v_1 = v_2}$$

$$\textsc{ivar-producer-result}\ \frac{\mathsf{producer}\ t \quad \mathsf{result}\ t\ v}{\mathsf{False}}$$

$$\textsc{ivar-inv-result}\ \frac{\mathsf{inv}\ t\ \Psi\ \Xi\ \Omega \quad \mathsf{result}\ t\ v}{\Rrightarrow \triangleright \square\ \Xi\ v}$$

$$\textsc{ivar-inv-result-consumer}\ \frac{\mathsf{inv}\ t\ \Psi\ \Xi\ \Omega \quad \mathsf{result}\ t\ v \quad \mathsf{consumer}\ t\ \mathrm{X}}{\Rrightarrow \triangleright^2\ \mathrm{X}\ v}$$

$$\textsc{ivar-create-spec}\ \frac{\mathsf{True}}{\texttt{create ()}} \quad t.\ \mathsf{inv}\ t\ \Psi\ \Xi\ \Omega \ast \mathsf{producer}\ t \ast \mathsf{consumer}\ t\ \Psi$$

$$\textsc{ivar-try-get-spec}\ \frac{\mathsf{inv}\ t\ \Psi\ \Xi\ \Omega}{\texttt{try\_get}\ t} \quad o.\ \mathbf{match}\ o\ \mathbf{with}\ \mid \mathsf{None} \Rightarrow \mathsf{True}\ \mid \mathsf{Some}\ v \Rightarrow £\,2 \ast \mathsf{result}\ t\ v\ \mathbf{end}$$

$$\textsc{ivar-try-get-spec-result}\ \frac{\mathsf{inv}\ t\ \Psi\ \Xi\ \Omega \ast \mathsf{result}\ t\ v}{\texttt{try\_get}\ t} \quad res.\ res = \mathsf{Some}\ v \ast £\,2$$

$$\textsc{ivar-get-spec}\ \frac{\mathsf{inv}\ t\ \Psi\ \Xi\ \Omega \ast \mathsf{result}\ t\ v}{\texttt{get}\ t} \quad res.\ res = v \ast £\,2$$

$$\textsc{ivar-wait-spec}\ \frac{\mathsf{inv}\ t\ \Psi\ \Xi\ \Omega \ast \triangleright \Omega\ t\ waiter}{\texttt{wait}\ t\ waiter} \quad o.\ \mathbf{match}\ o\ \mathbf{with}\ \mid \mathsf{None} \Rightarrow \mathsf{True}\ \mid \mathsf{Some}\ v \Rightarrow £\,2 \ast \mathsf{result}\ t\ v \ast \Omega\ t\ waiter\ \mathbf{end}$$

$$\textsc{ivar-set-spec}\ \frac{\mathsf{inv}\ t\ \Psi\ \Xi\ \Omega \ast \mathsf{producer}\ t \ast \triangleright \Psi\ v \ast \triangleright \square\ \Xi\ v}{\texttt{set}\ t\ v} \quad res.\ \exists\ waiters.\ res = \mathsf{list.to\text{-}val}\ waiters \ast \mathsf{result}\ t\ v \ast \mathop{\Large\ast}_{waiter \in waiters} \Omega\ t\ waiter$$

Figure 6.6: `Ivar`: Specification (excerpt)

6.10 Write-once variable

We provide three verified versions of concurrent write-once variables — also known as *ivars* — which are typically used to implement *futures*/*promises*. Basically, an ivar represents a *shared deferred value*; it is initially undetermined and can be determined only once. Although the three versions are close, they offer slightly different functionalities.

The first version is the simplest. It is used in our implementation of `Stdlib.Domain` (see Section 6.6). It supports basic operations: testing whether an ivar has been determined (`is_set`), querying the value (`try_get`) and setting the value (`set`). It does not feature any waiting mechanism. Its simplicity allows for an efficient implementation — essentially, an atomic reference.

The second version features a blocking waiting mechanism (`get`). Internally, the implementation relies on a *non-atomic* reference coupled with a mutex and a condition variable.

For the sake of performance, the test and query operations does not synchronize through the mutex, which involves a data race with the writer. In the sequentially consistent memory model that we adopted, this is not a problem. In the relaxed memory model of OCaml 5 (see Section 2.2), data races do not trigger *undefined behavior*, but there is still the problem of memory synchronization: if a domain reads the value without synchronizing with the writer, this domain does not necessarily see all the updates performed by the writer.

To make the specification somewhat correct in both memory models, we introduced a special assertion synchronized t attesting that the current domain "synchronized" with ivar t; in our sequentially consistent memory model, this assertion is trivial.

The third version features a non-blocking waiting mechanism (`wait`). It is used to implement futures in Chapter 10. Its specification is given in Figure 6.6.

The persistent assertion inv t Ψ Ξ Ω represents the knowledge that t is a valid ivar such that: (1) Ψ is the *non-persistent output predicate* satisfied by the produced value; (2) Ξ is the *persistent output predicate* satisfied by the produced value; (3) Ω is the *waiter predicate* satisfied by the waiters.

The persistent assertion result t v represents the knowledge that ivar t has been determined to value v. One can exploit this knowledge to read the value by calling `try_get` (IVAR-TRY-GET-SPEC-RESULT) or `get` (IVAR-IVAR-GET-SPEC). Using IVAR-INV-RESULT, it can also be combined with inv to obtain the persistent output predicate.

The exclusive assertion producer t represents the right to determine ivar t (IVAR-SET-SPEC). Doing so requires to give the two output predicates and yields a list of waiters satisfying the waiter predicate. These waiters correspond to those submitted by `wait` before the ivar was determined; afterwards, no more waiters can be added.

The assertion consumer t X represents the right to consume X once ivar t has been determined. Indeed, using IVAR-INV-RESULT-CONSUMER, it can be combined with inv and result to obtain X. When t is created, this assertion is produced with the full non-persistent predicate (IVAR-CREATE-SPEC); then, it can be divided into several parts (IVAR-CONSUMER-DIVIDE).

[12] `https://ocaml.org/manual/5.3/api/Semaphore.Counting.html`
[13] `https://ocaml.org/manual/5.3/api/Condition.html`

One notable aspect of this specification is that determination of the ivar — as indicated by result— is separated from the division of the output predicates — as achieved by consumer.

One slight drawback is the presence of later modalities (see Section 3.6) in IVAR-INV-RESULT and IVAR-INV-RESULT-CONSUMER. This is due to a well-known restriction on Iris's higher-order ghost state [Jung et al., 2018b]: occurrences of iProp must be guarded. To easily get rid of these later modalities, most operations emit *later credits* [Spies et al., 2022], which allows *logical* elimination — as opposed to *physical* elimination through program steps:

$$\frac{\pounds\,1 \qquad \triangleright P}{\mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} P}$$

6.11 Infinite array

We provide a verified `Inf_array` module implementing infinite arrays. We use it in Section 9.4 and Section 9.7 to simplify complex data structures, especially to abstract away from the handling of finite arrays, and focus on the core aspects. Internally, it consists of a finite array protected by a mutex; all operations acquire the mutex. Its specification is given in Figure 6.7.

The persistent assertion inv t represents the knowledge that t is a valid infinite array. It is returned by `create` (INF-ARRAY-CREATE-SPEC) and required by all operations.

The assertion model t vs represents the ownership of t and the knowledge that t contains vs, where vs is a sequence of values. When the array is created, vs is initialized to $\lambda\,\_.\,default$, where *default* is provided by the user (INF-ARRAY-CREATE-SPEC).

The assertion model' t vs_l vs_r is an alternative to model that is sometimes more convenient — including in Section 9.7. It represents the ownership of t and the knowledge that it contains $vs_l +\!\!+\, vs_r$, where vs_l is a list and vs_r a sequence of values. It can be obtained from model using INF-ARRAY-MODEL-TO-MODEL'.

To access array cells individually, the module provides the same operations as `Stdlib.Atomic`[14] (see Section 2.3.2.1): `get`, `set`, `xchg`, `cas` and `faa`. We provide atomic specifications (see Section 3.7) for these operations (INF-ARRAY-GET-SPEC, INF-ARRAY-SET-SPEC, INF-ARRAY-XCHG-SPEC). For example, INF-ARRAY-XCHG-SPEC states that, given a valid array t, `xchg` t i v atomically writes v to the i-th cell of t and returns the former value.

In Section 5.5, we mentioned an important limitation of prophets: prophecy resolution requires a *physically* atomic expression. In particular, using `Resolve` (`xchg` t i v) pid $v_{resolve}$ to resolve pid does not work because `xchg` t i v is not physically atomic; it is only *logically* atomic. In Section 9.4, though, we will need to perform prophecy resolution during `xchg`.

We propose an expedient: internal prophecy resolution. Concretely, we provide alternative versions of `xchg` and `cas`, `xchg_resolve` and `cas_resolve`, that internally resolve a given prophet. We designed their specifications with two constraints in mind, imposed by our use case: (1) the actual resolution should be left to the user, as it depends on the nature of the prophet; (2) it should be possible to access invariants during resolution.

For example, the specification of `xchg_resolve` is INF-ARRAY-XCHG-RESOLVE-SPEC. The third premise says that, atomically, we must be able to: (1) access model; (2) given back the updated model, resolve pid against an arbitrary logically atomic expression; (3)

[14] `https://ocaml.org/manual/5.3/api/Atomic.html`

$$\mathsf{persistent}\ (\mathsf{inv}\ t)$$

$$\dfrac{\forall\ i\ v.\ vs_l\ [i] = \mathsf{Some}\ v \rightarrow vs\ i = v \qquad \mathsf{model}\ t\ vs}{\mathsf{model'}\ t\ vs_l\ (\lambda\, i.\ vs\ (\mathsf{length}\ vs_l + i))}\ \textsc{inf-array-model-to-model'}$$

$$\textsc{inf-array-create-spec}\quad \dfrac{\mathsf{True}}{\dfrac{\texttt{create}\ \mathit{default}}{t.\ \mathsf{inv}\ t * \mathsf{model}\ t\ (\lambda\ \_.\ \mathit{default})}}$$

$$\textsc{inf-array-get-spec}\quad \dfrac{0 \le i * \mathsf{inv}\ t \;\big|\; vs.\ \mathsf{model}\ t\ vs}{\dfrac{\texttt{get}\ t\ i}{\mathsf{model}\ t\ vs \;\big|\; res.\ res = vs\ i}}$$

$$\textsc{inf-array-get-spec'}\quad \dfrac{0 \le i * \mathsf{inv}\ t \;\big|\; vs_l\ vs_r.\ \mathsf{model'}\ t\ vs_l\ vs_r}{\dfrac{\texttt{get}\ t\ i}{\mathsf{model'}\ t\ vs_l\ vs_r \;\big|\; res.\ \textbf{match}\ vs_l\ [i]\ \textbf{with} \mid \mathsf{None} \Rightarrow res = vs_r\ (i - \mathsf{length}\ vs_l) \mid \mathsf{Some}\ v \Rightarrow res = v\ \textbf{end}}}$$

$$\textsc{inf-array-set-spec}\quad \dfrac{0 \le i * \mathsf{inv}\ t \;\big|\; vs.\ \mathsf{model}\ t\ vs}{\dfrac{\texttt{set}\ t\ i\ v}{\mathsf{model}\ t\ (vs\ [i \mapsto v]) \;\big|\; ().\ \mathsf{True}}}$$

$$\textsc{inf-array-set-spec'}\quad \dfrac{0 \le i * \mathsf{inv}\ t \;\big|\; vs_l\ vs_r.\ \mathsf{model'}\ t\ vs_l\ vs_r}{\dfrac{\texttt{set}\ t\ i\ v}{\textbf{if}\ \mathsf{decide}\ (i < \mathsf{length}\ vs_l)\ \textbf{then}\ \mathsf{model'}\ t\ (vs_l\ [i \mapsto v])\ vs_r\ \textbf{else}\ \mathsf{model'}\ t\ vs_l\ (vs_r\ [i - \mathsf{length}\ vs_l \mapsto v]) \;\big|\; ().\ \mathsf{True}}}$$

$$\textsc{inf-array-xchg-spec}\quad \dfrac{0 \le i * \mathsf{inv}\ t \;\big|\; vs.\ \mathsf{model}\ t\ vs}{\dfrac{\texttt{xchg}\ t\ i\ v}{\mathsf{model}\ t\ (vs\ [i \mapsto v]) \;\big|\; res.\ res = vs\ i}}$$

$$\textsc{inf-array-xchg-resolve-spec}\quad \dfrac{0 \le i \qquad \mathsf{inv}\ t \qquad {}^{\top}{\Rrightarrow}^{\mathcal{E}}\ \exists\ vs.\ \mathsf{model}\ t\ vs * \forall\ e.\ e \xrightarrow{\mathsf{pure}} () \twoheadrightarrow \mathsf{model}\ t\ (vs\ [i \mapsto v]) \twoheadrightarrow \mathsf{wp}_{\mathcal{E}}\ \texttt{Resolve}\ e\ pid\ v_{resolve}\ \{\ \_.\ {}^{\mathcal{E}}{\Rrightarrow}^{\top}\ \Phi\ (vs\ i)\ \}}{\mathsf{wp}\ \texttt{xchg\_resolve}\ t\ i\ v\ pid\ v_{resolve}\ \{\ \Phi\ \}}$$

Figure 6.7: `Inf_array`: Specification (excerpt)

prove the postcondition Φ. Crucially, as in the definition of atomic specifications (see Section 3.7), Φ is chosen by the user; the only way to prove the conclusion is to consume the premise at some point. This premise could be generalized to match the expressivity of atomic updates (see Section 3.7), especially the retry loop.

6.12 Future work

Our standard library currently features basic imperative data structures: array, dynamic array, stack, queue, double-ended queue. In the future, we would like to verify more complex data structures, striving for a complete cover of the OCaml standard library. In particular, it would be interesting to integrate the verified hash table of Pottier [2017].

Chapter 7

Persistent data structures

In this chapter, we use Zoo to verify *persistent* data structures.

Definition. A data structure is said to be persistent when any update operation preserves the previous version of the data structure. There are many ways to implement these data structures and make them efficient — see, for instance, the lectures of Xavier Leroy at Collège de France[1]. *Purely functional* implementations are fully immutable; they typically rely on sharing substructures between versions. *Imperative* implementations rely on mutable state under the hood, reshaping and rebalancing the underlying structure that versions refer to.

Use cases. Persistent data structures are typically used in contexts where "going back to a previous version" is a desired functionality: version control systems, saves in games, backtracking algorithms, dynamic bindings [Baker, 1978].

7.1 Purely functional data structures

We verified two basic functional data structures: persistent stacks and persistent queues. For example, the specification of persistent queues is given in Figure 7.1. All versions are persistent in the Iris sense and update operations (`push` and `pop`) return new, independent versions.

Currently, verification of functional programs relies on the regular ZooLang translation, *i.e.* on a deeply embedded representation. However, we found this approach is cumbersome. In the future, it would be desirable to be able to verify them directly in Rocq, through a translation to Gallina. Similarly to Hacspec [Haselwarter et al., 2024], this new translation would come with a generated proof of equivalence with the ZooLang representation.

7.2 Persistent array

Persistent arrays can be naively implemented by copying imperative arrays. However, this is a performance disaster when the number of versions is large. We verified an efficient, imperative implementation based on the idea of Baker [1991], that we discovered through Conchon and Filliâtre [2008]. It is a good example of how a seemingly persistent

[1] `https://xavierleroy.org/CdF/2022-2023/`

$$\text{persistent}\ (\text{model}\ t\ vs)$$

PQUEUE-MODEL-NIL

$$\text{model}\ \texttt{empty}\ []$$

PQUEUE-IS-EMPTY-SPEC

$$\frac{\dfrac{\text{model}\ t\ vs}{\texttt{is\_empty}\ t}}{res.\ res = \text{decide}\ (vs = [])}$$

PQUEUE-PUSH-SPEC

$$\frac{\dfrac{\text{model}\ t\ vs}{\texttt{push}\ t\ v}}{t'.\ \text{model}\ t'\ (vs +\!\!+ [v])}$$

PQUEUE-POP-SPEC

$$\frac{\dfrac{\text{model}\ t\ vs}{\texttt{pop}\ t}}{\begin{array}{l} o.\ \textbf{match}\ o\ \textbf{with} \\ \quad \mid \text{None} \Rightarrow \\ \qquad vs = [] \\ \quad \mid \text{Some}\ p \Rightarrow \\ \qquad \exists\ v\ vs'\ t'. \\ \qquad vs = v :: vs' * \\ \qquad p = (v, t') * \\ \qquad \text{model}\ t'\ vs' \\ \quad \textbf{end} \end{array}}$$

Figure 7.1: Pqueue: Specification

interface can hide a mutable world. In Section 7.3 and Section 7.4, we reshape and develop this implementation.

7.2.1 Specification

The specifications is given in Figure 7.2.

The persistent assertion model $t\ \gamma\ vs$ represents the knowledge that t is a valid version containing values vs. γ is a logical identifier that can be interpreted as the underlying mutable state shared by the different versions.

The exclusive assertion inv $\tau\ \gamma$ represents the ownership of the mutable state attached to γ. It is required and returned by all operations, allowing them to internally access and modify the state. τ is a persistent predicate over values that represents the type of the elements.

`make` *equal sz v* creates a new state identified by γ along with an initial version t (PARRAY-1-MAKE-SPEC) initialized with v. The user must provide a type τ and an equality function *equal* — typically, physical or structural equality — satisfying equal-model τ, which is defined as follows:

$$\text{equal-model}\ \tau\ \mathit{equal} \triangleq \{\, \tau\ v_1 * \tau\ v_2 \,\}\ \mathit{equal}\ v_1\ v_2\ \{\, b.\ \textbf{if}\ b\ \textbf{then}\ v_1 = v_2\ \textbf{else}\ \text{True} \,\}$$

This function is used to short-cut the `set` operation: in the case where the value to write is equal to the current value (at a given index in the input version) according to *equal*, `set` simply returns the input version. equal-model allows the operations to call *equal* as many times as they wish, but only on values in type τ; when it returns true, the compared values must be equal in Rocq.

$$\text{persistent}\ (\text{model}\ t\ \gamma\ vs)$$

$$\textsc{parray-1-inv-exclusive}\quad \frac{\text{inv}\ \tau\ \gamma_1 \qquad \text{inv}\ \tau\ \gamma_2}{\text{False}}$$

$$\textsc{parray-1-make-spec}\quad \frac{\begin{array}{l}\text{equal-model}\ \tau\ \mathit{equal}\ * \\ \tau\ v\end{array}}{\texttt{make}\ \mathit{equal}\ \mathit{sz}\ v} \ \begin{array}{l} t.\ \exists\ \gamma. \\ \quad \text{inv}\ \tau\ \gamma\ * \\ \quad \text{model}\ t\ \gamma\ (\text{replicate}\ \mathit{sz}\ v)\end{array}$$

$$\textsc{parray-1-get-spec}\quad \frac{\begin{array}{l} vs\,[i] = \text{Some}\ v\ * \\ \text{inv}\ \tau\ \gamma\ * \\ \text{model}\ t\ \gamma\ vs\end{array}}{\texttt{get}\ t\ i} \ \begin{array}{l} \mathit{res}.\ \mathit{res} = v\ * \\ \quad \text{inv}\ \tau\ \gamma\end{array}$$

$$\textsc{parray-1-set-spec}\quad \frac{\begin{array}{l} 0 \le i < \text{length}\ vs\ * \\ \text{inv}\ \tau\ \gamma\ * \\ \text{model}\ t\ \gamma\ vs\ * \\ \tau\ v\end{array}}{\texttt{set}\ t\ i\ v} \ \begin{array}{l} t'.\ \text{inv}\ \tau\ \gamma\ * \\ \quad \text{model}\ t'\ \gamma\ (vs\,[i \mapsto v])\end{array}$$

Figure 7.2: `Parray_1`: Specification

`get` t i reads the value at index i in version t (PARRAY-1-GET-SPEC). `set` t i v returns a version — it may be the same — with the same elements as t except v has been written at index i.

7.2.2 Implementation

The idea of Baker, quoting Conchon and Filliâtre [2008], is "to use an imperative array for the newest version of the persistent array and indirections for old versions". Let us explain this in detail.

Concretely, the implementation relies on the following OCaml types:

```
type 'a descr =
  | Root of
    { equal: 'a -> 'a -> bool;
      data: 'a array;
    }
  | Diff of
    { index: int;
      value: 'a;
      parent: 'a t;
    }
and 'a t =
  'a descr ref
```

A version (`'a t`) is a reference to an internal descriptor (`'a descr`), which is either marked as the `Root` node or a `Diff` node. As suggested by these terms, the versions form a tree in memory, called the *version tree*. The `Root` version — there is only one such version — carries the equality function provided by the user along with an imperative array that contains the values associated to the version via model. A `Diff` version t carries an index i, a value v and a parent version t' such that the values of t is the values of t' where i has been set to v; in other words, to restore t from t', it suffices to apply the patch

```
let a = make equal sz v_a in      (a)
let b = set a i v_b in            (b)
let c = set b i v_c in            (c)
let _ = get b i in                (d)
let d = set b i v_d in            (e)
let _ = get a i in                (f)
...
```

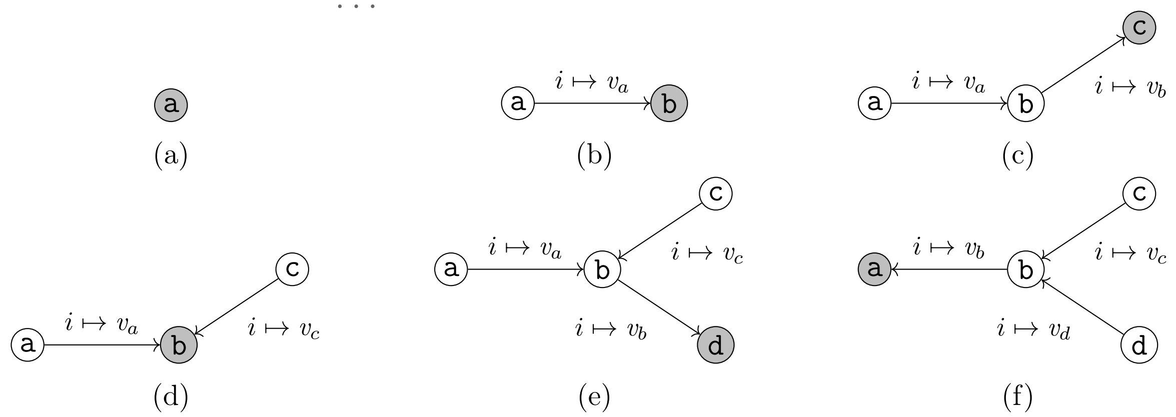


Figure 7.3: Parray_1: Version trees

$i \mapsto v$. This structure allows to restore the values of any version of the tree by applying the patches along the path from the `Root` version to the target version in that order; this operation is called *rerooting*.

For example, consider the program of Figure 7.3, which creates a persistent array and performs multiple `get` and `set` at the same index i — for simplicity. Let us describe what happens in this program in terms of the version tree. (a) We create a new version tree with only one version `a` initialized with v_a. (b) We create a new version `b` which becomes the new `Root` while `a` becomes a `Diff` carrying $i \mapsto v_a$. (c) Similarly, we create a new version `c`. (d) We reroot to `b`, reversing the edge between `c` and `b`. (e) We create a new version `d`. (f) We reroot to `a`, reversing the edges between `d` and `b` and between `b` and `a`.

7.2.3 Ghost state

The definition of the predicates inv and model is given in Figure 7.5. We describe it bit by bit.

nodes theory (Figure 7.4). This theory allows constructing a monotonic mapping that associates to a version its list of elements.

The assertion nodes-auth γ *nodes* represents the ownership of the full mapping *nodes*. It is stored in inv.

The persistent assertion nodes-elem γ *node vs* represents the knowledge that *node* has values *vs* (NODES-ELEM-LOOKUP). model is directly defined on top of it (t is a value while *node* is a memory location).

`Diff` version. The assertion node-model γ *node vs* represents the ownership of the `Diff` node *node* with values *vs*. Its main function is to relate *node* to its parent using the property stated in Section 7.2.2. It refers to the values of the parent through nodes-elem.

$$\frac{\textsf{nodes-auth}\ \gamma\ \mathit{nodes} \qquad \textsf{nodes-elem}\ \gamma\ \mathit{node}\ v}{\mathit{nodes}\,[\mathit{node}] = \textsf{Some}\ v}\ \textsc{nodes-elem-lookup}$$

$$\frac{\textsf{nodes-elem}\ \gamma\ \mathit{nodes}\ \mathit{vs}_1 \qquad \textsf{nodes-elem}\ \gamma\ \mathit{nodes}\ \mathit{vs}_2}{\mathit{vs}_1 = \mathit{vs}_2}\ \textsc{nodes-elem-agree}$$

$$\frac{\mathit{nodes}\,[\mathit{node}] = \textsf{None} \qquad \textsf{nodes-auth}\ \gamma\ \mathit{nodes}}{\dot{\Rrightarrow}\ \textsf{nodes-auth}\ \gamma\ (\mathit{nodes}\,[\mathit{node} \mapsto \mathit{vs}]) * \textsf{nodes-elem}\ \gamma\ \mathit{node}\ \mathit{vs}}\ \textsc{nodes-insert}$$

Figure 7.4: `Parray_1`: nodes theory

$$\begin{aligned}
&\textsf{node-model}\ \gamma\ \mathit{node}\ \mathit{vs} \triangleq\\
&\quad \exists\ i\ v\ \mathit{node}'\ \mathit{vs}'.\\
&\quad \mathit{node} \mapsto \text{`}\texttt{Diff}(i, v, \mathit{node}') *\\
&\quad \tau\ v *\\
&\quad \textsf{nodes-elem}\ \gamma\ \mathit{node}'\ \mathit{vs}' *\\
&\quad \textsf{length}\ \mathit{vs} = \gamma.\textsf{size} *\\
&\quad i < \gamma.\textsf{size} *\\
&\quad \mathit{vs} = \mathit{vs}'\,[i \mapsto v]
\end{aligned}$$

$$\begin{aligned}
&\textsf{model}\ t\ \gamma\ \mathit{vs} \triangleq\\
&\quad \exists\ \mathit{node}.\\
&\quad t = \mathit{node} *\\
&\quad \textsf{nodes-elem}\ \gamma\ \mathit{node}\ \mathit{vs}
\end{aligned}$$

$$\begin{aligned}
&\textsf{inv}\ \gamma\ \tau \triangleq\\
&\quad \exists\ \mathit{root}\ \mathit{vs}_{\mathit{root}}\ \mathit{nodes}.\\
&\quad \textsf{equal-model}\ \gamma.\textsf{equal} *\\
&\quad \mathit{root} \mapsto \text{`}\texttt{Root}(\gamma.\textsf{equal}, \gamma.\textsf{data}) *\\
&\quad \textsf{array.model}\ \gamma.\textsf{data}\ 1\ \mathit{vs}_{\mathit{root}} *\\
&\quad \textsf{nodes-elem}\ \gamma\ \mathit{root}\ \mathit{vs}_{\mathit{root}} *\\
&\quad \textsf{lenth}\ \mathit{vs}_{\mathit{root}} = \gamma.\textsf{size} *\\
&\quad \Big(\mathop{\Large\ast}_{v\,\in\,\mathit{vs}_{\mathit{root}}} \tau\ v\Big) *\\
&\quad \textsf{nodes-auth}\ \gamma\ \mathit{nodes} *\\
&\quad \Big(\mathop{\Large\ast}_{\mathit{node} \mapsto \mathit{vs}\,\in\,\textsf{delete}\ \mathit{root}\ \mathit{nodes}} \textsf{node-model}\ \gamma\ \mathit{node}\ \mathit{vs}\Big)
\end{aligned}$$

Figure 7.5: `Parray_1`: Predicates definition

Version graph. In inv, all the nodes are gathered using an iterated separating conjunction. One striking thing is the fact that there is clearly a graph of nodes but no notion of tree. Indeed, we did not formalize the tree property because it is not needed to prove the specification. However, this also means termination is not straightforward – but our logic only guarantees partial correction anyway. Note that we do formalize the tree property in Section 7.4.4.

7.3 Snapshottable array

A more general way for a data structure to support persistency is to make a it *snapshottable*. An imperative data structure is snapshottable if it possible to take and restore *snapshots* of its state. Contrary to a functional interface where every version is persistent and updates generate new versions, the user explicitly chooses which versions are persistent by taking snapshots and updates modify the current state. One may easily construct a persistent variant of a snapshottable data structure by systematically taking a snapshot after an update.

We verified a snapshottable version of the persistent arrays of Section 7.2. We could have presented this version first and deduced a persistent variant through the canonical construction mentioned above, but the resulting persistent implementation would be suboptimal and we wanted to explain the idea of Baker in a simple and familiar setting.

In Section 7.4, we will see that the snapshottable interface enables an interesting optimization. It also has the advantage of featuring a shared object — the array itself, to which snapshots refer —, which allows centralizing the resources, including the user-provided equality function and the underlying imperative array that previously had to be transported during rerooting.

7.3.1 Specification

The specification is given in Figure 7.6.

The exclusive assertion model τ t vs represents the ownership of the array and the knowledge that it currently contains values vs. It is returned by `make` (PARRAY-2-MAKE-SPEC) and used by all operations in an imperative fashion, including `get` (PARRAY-2-GET-SPEC) and `set` (PARRAY-2-SET-SPEC).

The persistent assertion snapshot s t vs represents the knowledge that s is a valid snapshot of array t at version vs. It can be obtained through `capture` (PARRAY-2-CAPTURE-SPEC) and used to restore vs through `restore` (PARRAY-2-RESTORE-SPEC).

7.3.2 Implementation

The implementation relies on the exact same technique that we presented in Section 7.2.2. The `get` and `set` operations no longer need to reroot, as they directly access the current version, which is always the root of the version tree. `capture` simply stores the current root and `restore` reroots to the captured version.

7.3.3 Ghost state

The ghost state is very similar to that of Section 7.2.3: model holds the same resources as parray-1.inv, snapshot the same as parray-1.model. We refer to the mechanization for

$$\mathsf{persistent}\ (\mathsf{snapshot}\ s\ t\ vs)$$

Parray-2-model-exclusive

$$\frac{\mathsf{model}\ \tau\ t\ vs_1 \qquad \mathsf{model}\ \tau\ t\ vs_1}{\mathsf{False}}$$

Parray-2-make-spec

$$\begin{array}{c} \mathsf{equal\text{-}model}\ \tau\ equal\ * \\ \tau\ v \\ \hline \texttt{make}\ equal\ sz\ v \\ \hline t.\ \mathsf{model}\ \tau\ t\ (\mathsf{replicate}\ sz\ v) \end{array}$$

Parray-2-get-spec

$$\begin{array}{c} vs\,[i] = \mathsf{Some}\ v\ * \\ \mathsf{model}\ \tau\ t\ vs \\ \hline \texttt{get}\ t\ i \\ \hline res.\ res = v\ * \\ \mathsf{model}\ \tau\ t\ vs \end{array}$$

Parray-2-set-spec

$$\begin{array}{c} 0 \leq i < \mathsf{length}\ vs\ * \\ \mathsf{model}\ \tau\ t\ vs\ * \\ \tau\ v \\ \hline \texttt{set}\ t\ i\ v \\ \hline ().\ \mathsf{model}\ \tau\ t\ (vs\,[i \mapsto v]) \end{array}$$

Parray-2-capture-spec

$$\begin{array}{c} \mathsf{model}\ \tau\ t\ vs \\ \hline \texttt{capture}\ t \\ \hline s.\ \mathsf{model}\ \tau\ t\ vs\ * \\ \mathsf{snapshot}\ s\ t\ vs \end{array}$$

Parray-2-restore-spec

$$\begin{array}{c} \mathsf{model}\ \tau\ t\ vs\ * \\ \mathsf{snapshot}\ s\ t\ vs' \\ \hline \texttt{restore}\ t\ s \\ \hline ().\ \mathsf{model}\ \tau\ t\ vs' \end{array}$$

Figure 7.6: `Parray_2`: Specification

details.

7.4 Snapshottable store

We verified an implementation of snapshottable *heterogeneous stores* developed by Basile Clément and Gabriel Scherer [Allain et al., 2024], available through the `Store`[2] library.

A heterogeneous store is a bag of mutable references not necessarily of the same type. This abstraction can be used to easily add snapshots to complex imperative data structures — we show one example in Section 7.5. They were motivated by applications in backtracking algorithms, including in the `Inferno`[3] library and the Alt-Ergo[4] SMT solver.

The implementation is based on the idea of Baker enhanced with an important optimization that we present in Section 7.4.2: *record elision*. The resulting algorithm is fairly short but subtle. As a matter of fact, during the development, Clément and Scherer heavily relied on the `Monolith` [Pottier, 2021] fuzz-testing library. When they reached a fixed point, they found that it was quite difficult to convince oneself of its correctness, and so they suggested we verify it.

7.4.1 Specification

The specification is given on Figure 7.7. It is very similar to that of snapshottable arrays presented in Section 7.3.1, except lists are naturally generalized to maps. Further-

[2] `https://gitlab.com/basile.clement/store/`
[3] `https://gitlab.inria.fr/fpottier/inferno`
[4] `https://alt-ergo.ocamlpro.com/`

$$\mathsf{persistent}\ (\mathsf{snapshot}\ s\ t\ \sigma)$$

$$\textsc{pstore-model-exclusive}\quad \dfrac{\mathsf{model}\ t\ \sigma_1 \qquad \mathsf{model}\ t\ \sigma_2}{\mathsf{False}}$$

$$\textsc{pstore-create-spec}\quad \begin{array}{c} \mathsf{True} \\ \hline \texttt{create}\ () \\ \hline t.\ \mathsf{model}\ t\ \emptyset \end{array}$$

$$\textsc{pstore-ref-spec}\quad \begin{array}{c} \mathsf{model}\ t\ \sigma \\ \hline \texttt{ref}\ t\ v \\ \hline r.\ \sigma\,[r] = \mathsf{None} * \\ \mathsf{model}\ t\ (\sigma\,[r \mapsto v]) \end{array}$$

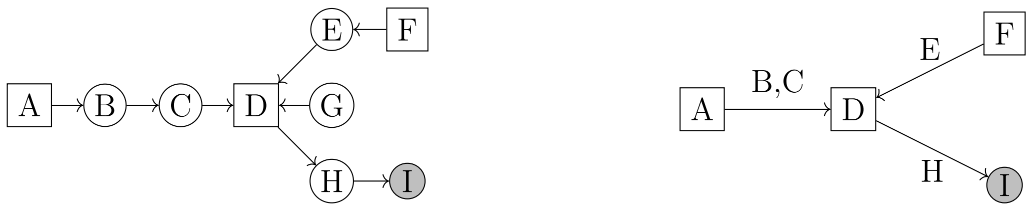
$$\textsc{pstore-get-spec}\quad \begin{array}{c} \sigma\,[r] = \mathsf{Some}\ v * \\ \mathsf{model}\ t\ \sigma \\ \hline \texttt{get}\ t\ r \\ \hline \mathit{res}.\ \mathit{res} = v * \\ \mathsf{model}\ t\ \sigma \end{array}$$

$$\textsc{pstore-set-spec}\quad \begin{array}{c} r \in \mathsf{dom}\ \sigma * \\ \mathsf{model}\ t\ \sigma \\ \hline \texttt{set}\ t\ r\ v \\ \hline ().\ \mathsf{model}\ t\ (\sigma\,[r \mapsto v]) \end{array}$$

$$\textsc{pstore-capture-spec}\quad \begin{array}{c} \mathsf{model}\ t\ \sigma \\ \hline \texttt{capture}\ t \\ \hline s.\ \mathsf{model}\ t\ \sigma * \\ \mathsf{snapshot}\ s\ t\ \sigma \end{array}$$

$$\textsc{pstore-restore-spec}\quad \begin{array}{c} \mathsf{model}\ t\ \sigma * \\ \mathsf{snapshot}\ s\ t\ \sigma' \\ \hline \texttt{restore}\ t\ s \\ \hline ().\ \mathsf{model}\ t\ \sigma' \end{array}$$

Figure 7.7: `Pstore`: Specification



(a) (b)

Figure 7.8: `Pstore`: Version tree and its corresponding subtree of captured nodes (squares represent captured nodes, circles non-captured nodes)

more, contrary to arrays, the domain of the store is unbounded; new references can be created using the `ref` operation (PSTORE-REF-SPEC).

7.4.2 Implementation

Snapshottable stores can be implemented by simply adapting the algorithm of Section 7.3, replacing the imperative array by references. As it is, this implementation suffers from one significant overhead compared to plain references: while the `get` operation is fast (one memory read), the `set` operation incurs additional costs (in the slow path) due to the systematic creation of a new node in the version tree. Basile Clément and Gabriel Scherer designed an optimization called *record elision* that makes `set` much faster (roughly as fast as for plain references) in the common case where the `capture` and `restore` operations are infrequent.

Tree of captured nodes. To explain how record elision works, we first need to take a closer look at the structure of the version tree. In the presence of snapshots, we can distinguish *captured* nodes and *non-captured* nodes. Captured nodes form a subtree whose

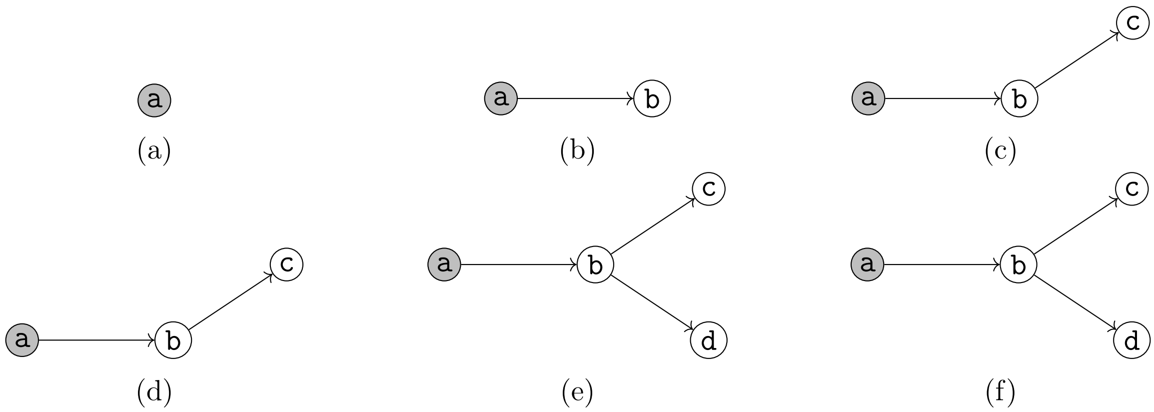


Figure 7.9: Pstore: Historic trees corresponding to the version trees of Figure 7.3

edges are chains of non-captured nodes. For example, the version tree of Figure 7.8a corresponds to the subtree of Figure 7.8b. The root of the subtree coincides with the last captured node; we call it the *base node*.

Remarkably, this structure "forgets" about useless non-captured nodes like G that represent aborted paths — incidentally, these nodes are not superficially kept alive and can therefore be garbage-collected.

We track separately the *final chain*, defined as the chain of non-captured nodes connecting the base node (D) to the root of the version tree (I).

Record elision. Thanks to this structure, the idea of record elision can be simply explained. To avoid systematically creating a new node in `set`, we want to detect the case where the reference to write into is already registered in the final chain, in the sense that some node in the final chain is already responsible for restoring the previous value. In this case, creating a new node would be redundant, so we can elide the record.

Logically, performing the write without recording is harder to reason about because it changes the mappings of existing nodes. More precisely, when record elision takes place during `set` r v, the write propagates in the final chain, from the root (included) to the last (and only) `Diff` node on reference r (excluded). In other words, for each of these nodes, the new logical mapping is $\sigma\,[r \mapsto v]$, where σ is the current mapping.

Consequently, contrary to the algorithm without record elision, the final chain is not persistent, in the sense that the mappings attached to its nodes may change. However, the subtree of captured nodes is entirely persistent, which allows rerooting as before. When the root is captured for the first time — meaning the final chain is non-empty —, the final chain is "frozen", *i.e.* made persistent.

Note that this optimization is possible because the imperative interface makes snapshots explicit. It is not available in the persistent interface.

Having laid out record elision, we now face the question of how to realize it. Indeed, in the implementation of Section 7.3, there is no way to detect whether a reference is already registered in the final chain.

Basic realization. A simple approach consists in storing the base node, which can be used to compute the final chain. Even better, we can store the entire final chain, which avoids recomputing it. Unfortunately, this implementation does not perform well for two reasons: (1) the cost of iterating the final chain is linear in the number of distinct modified references since the last `restore`, which may be unacceptable; (2) the liveness properties are not as good as before because the final chain is kept alive, which prevents it from being garbage-collected.

Node identifiers. A more sophisticated approach consists in assigning each captured node a unique identifier and annotating each reference with the identifier of the base node at the time of the last write. To determine whether record elision is possible, it suffices to compare the identifier of the reference with the identifier of the current base node; if, and only if, they are the same, the reference is already registered in the final chain and therefore elision can be performed.

For this to work, a few adjustments are necessary: (1) keep track of the identifier of the base node in the store; (2) keep track of the identifiers of captured nodes, either directly in the nodes or in the snapshots; (3) store identifiers of references in `Diff` nodes to restore them correctly during rerooting.

Historic tree. So far, we only considered the version tree as it is represented in memory, with the root being the current version. We can also look at nodes from the perspective of the *historic tree*, whose root is the initial node in the history of the store — we call the node the *origin*. For example, the historic trees corresponding to the version trees of Figure 7.3 are given in Figure 7.9.

This tree is particularly interesting because it is always defined and monotonic: it grows over time, as new nodes are inserted, but its root never changes. Furthermore, we make the following remark: according to the structure of the version tree, reversing the path from the root to some target node *is equivalent* to (1) reversing the path from the root to the origin and (2) reversing the path from the origin to the target node. This has significant implications: at any point in time, the only contributing nodes to the history of references are exactly those in the path from the origin to the root.

Generations. This new insight suggests an alternative realization: replacing unique node identifiers with depths in the historic tree. Indeed, a node is not uniquely identified by its depth in the historic tree but it is sufficient since there is only one contributing node per depth. To determine whether record elision is possible, it suffices to compare the depth of the reference with the depth of the current base node.

This realization was discovered by Basile Clément and Gabriel Scherer. In the actual implementation, depths are called *generations* and count only captured nodes.

7.4.3 Proof insights

Global generations. Perhaps surprisingly, proving the specification of Figure 7.7 is non-trivial and extremely tedious. As a matter of fact, Alexandre Moine and Gabriel Scherer attempted to formalize the reasoning of Section 7.4.2 in Rocq but ran into a wall. The main difficulty lies in the formalization of the two trees (version tree and historic tree) and their relationship. In short, while the reasoning was *local* without record elision, it becomes *global* with record elision.

Local generations. Our own, simultaneous attempt succeeded thanks to a key insight: it is possible to formalize generations *in a local way*, without making the historic tree

$$\mathsf{persistent}\ (\mathsf{snapshot}\ s\ t\ \sigma)$$

$$\textsc{pstore-raw-model-valid}\quad \frac{\mathsf{model}\ t\ \sigma_0\ \sigma}{\mathsf{dom}\ \sigma \subseteq \mathsf{dom}\ \sigma_0}$$

$$\textsc{pstore-raw-model-exclusive}\quad \frac{\mathsf{model}\ t\ \sigma_{01}\ \sigma_1 \qquad \mathsf{model}\ t\ \sigma_{02}\ \sigma_2}{\mathsf{False}}$$

$$\textsc{pstore-raw-create-spec}\quad \begin{array}{c} \mathsf{True} \\ \hline \texttt{create ()} \\ \hline t.\ \mathsf{model}\ t\ \emptyset\ \emptyset \end{array}$$

$$\textsc{pstore-raw-ref-spec}\quad \begin{array}{c} \mathsf{model}\ t\ \sigma_0\ \sigma \\ \hline \texttt{ref}\ t\ v \\ \hline r.\ \sigma_0\,[r] = \mathsf{None} \ast \\ \mathsf{model}\ t\ (\sigma_0\,[r \mapsto v])\ \sigma \end{array}$$

$$\textsc{pstore-raw-get-spec}\quad \begin{array}{c} (\sigma_0 \cup \sigma)\,[r] = \mathsf{Some}\ v \ast \\ \mathsf{model}\ t\ \sigma_0\ \sigma \\ \hline \texttt{get}\ t\ r \\ \hline res.\ res = v \ast \\ \mathsf{model}\ t\ \sigma_0\ \sigma \end{array}$$

$$\textsc{pstore-raw-set-spec}\quad \begin{array}{c} r \in \mathsf{dom}\ \sigma_0 \ast \\ \mathsf{model}\ t\ \sigma_0\ \sigma \\ \hline \texttt{set}\ t\ r\ v \\ \hline ().\ \mathsf{model}\ t\ \sigma_0\ (\sigma\,[r \mapsto v]) \end{array}$$

$$\textsc{pstore-raw-capture-spec}\quad \begin{array}{c} \mathsf{model}\ t\ \sigma_0\ \sigma \\ \hline \texttt{capture}\ t \\ \hline s.\ \mathsf{model}\ t\ \sigma_0\ \sigma \ast \\ \mathsf{snapshot}\ s\ t\ \sigma \end{array}$$

$$\textsc{pstore-raw-restore-spec}\quad \begin{array}{c} \mathsf{model}\ t\ \sigma_0\ \sigma \ast \\ \mathsf{snapshot}\ s\ t\ \sigma' \\ \hline \texttt{restore}\ t\ s \\ \hline ().\ \mathsf{model}\ t\ \sigma_0\ \sigma' \end{array}$$

Figure 7.10: `Pstore`: More general specification with ground mapping σ_0

explicit. As a result, most of the reasoning remains local; global reasoning is still needed for the version tree but remains manageable.

The idea is the following: given essentially the same rerooting structure as before where mappings also contain generations, we require (1) the generations of the mapping of a captured node to be bounded by the node generation and (2) the generation of the next potential captured node to be strictly greater than the generation of the base node.

Low-level interface. Another difficulty is dynamic reference creation. In the specification of Figure 7.7, each reference is local to the branch in which it was created; restoring another branch discards the reference. This requires small adjustments in the formalization of the rerooting logic, as done by Alexandre Moine in his proof without record elision.

We opted for an alternative, arguably more satisfactory way: instead of directly proving the specification of Figure 7.7, we derived it from a more general specification, given in Figure 7.10. This specification is closer to the actual implementation in the sense that references are not local to a branch. When a reference is created, it is published globally in a *ground mapping* σ_0 (PSTORE-RAW-REF-SPEC, PSTORE-RAW-MODEL-VALID); any branch can access the reference (PSTORE-RAW-RESTORE-SPEC).

$$
\begin{array}{l}
\mathsf{model}\ t\ \sigma_0\ \sigma \triangleq \\
\quad \exists\ \ell\ \gamma\ \varsigma\ g\ \mathit{root}\ \mathit{base}\ \mathit{descr}\ \delta s\ \mathit{cnodes}\ \epsilon s. \\
\quad t = \ell\ * \\
\quad \sigma = \varsigma.\mathsf{val}\ * \\
\quad \mathsf{meta}\ \ell\ \top\ \gamma\ * \\
\quad \ell \mapsto \{\ \mathsf{gen}\colon g;\ \mathsf{root}\colon \mathit{root}\ \}\ * \\
\quad \mathit{root} \mapsto \S\mathtt{Root}\ * \\
\quad \left(\mathop{\Large\ast}_{r \mapsto \mathit{data} \in \varsigma/\sigma_0} r \mapsto \mathit{data} \right)\ * \\
\quad \mathsf{cnode\text{-}model}\ \gamma\ \sigma_0\ \mathit{base}\ \mathit{descr}\ \{\ \mathsf{parent}\colon \mathit{root};\ \mathsf{label}\colon \delta s\ \}\ \varsigma\ * \\
\quad \mathit{descr}.\mathsf{gen} < g\ * \\
\quad (\forall\ r \in \delta s.\mathsf{ref}.\ \exists\ \mathit{data}.\ \varsigma\,[r] = \mathsf{Some}\ \mathit{data} \wedge \mathit{data}.\mathsf{gen} = g)\ * \\
\quad \mathsf{cnodes\text{-}auth}\ \gamma\ \mathit{cnodes}\ * \\
\quad \mathit{cnodes}\,[\mathit{base}] = \mathsf{Some}\ \mathit{descr}\ * \\
\quad \mathsf{treemap\text{-}rooted}\ \epsilon s\ \mathit{base}\ * \\
\quad \left(\begin{array}{l} \mathop{\Large\ast}_{\mathit{cnode} \mapsto \mathit{descr},\, \epsilon \,\in\, \mathsf{delete}\ \mathit{base}\ \mathit{cnodes},\ \epsilon s} \\ \exists\ \mathit{descr}'. \\ \mathit{cnodes}\,[\epsilon.\mathsf{parent}] = \mathsf{Some}\ \mathit{descr}'\ * \\ \mathsf{cnode\text{-}model}\ \gamma\ \sigma_0\ \mathit{cnode}\ \mathit{descr}\ \epsilon\ \mathit{descr}'.\mathsf{store} \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{snapshot}\ s\ t\ \sigma \triangleq \\
\quad \exists\ \ell\ \gamma\ g\ \mathit{cnode}\ \mathit{descr}. \\
\quad t = \ell\ * \\
\quad s = (t, g, \mathit{cnode})\ * \\
\quad \sigma = \mathit{descr}.\mathsf{store}.\mathsf{val}\ * \\
\quad \mathit{descr}.\mathsf{gen} \le g\ * \\
\quad \mathsf{meta}\ \ell\ \top\ \gamma\ * \\
\quad \mathsf{cnodes\text{-}elem}\ \gamma\ \mathit{cnode}\ \mathit{descr}
\end{array}
$$

Figure 7.11: `Pstore`: Predicates definition (simplified)

7.4.4 Ghost state

A simplified definition of the predicates model and snapshot is given in Figure 7.11; we omitted the definition of cnode-model. The structure is roughly similar to that of Figure 7.5. In particular, the cnodes theory is the counterpart of the nodes theory, cnode-model is the counterpart of node-model, the iterated conjunction over ς/σ_0 (the extension of σ_0 with ς) is the counterpart of array.model.

In yellow, we highlighted the parts formalizing the tree of captured nodes, represented as a mapping ϵs from nodes to edges, where an edge consists of a parent node and a chain. In particular, the pure assertion treemap-rooted ϵs *base* states that ϵs represents a tree rooted in *base*; all the tree logic is contained in the treemap theory.

In blue, we highlighted the parts formalizing the second part of the generation logic described in Section 7.4.3. The first part resides in the cnode-model predicate.

7.4.5 Future work

Simpler proof. We developed the invariant based on our understanding of the algorithm; in particular, the subtree of captured nodes is explicit. However, this representation makes rerooting reasoning very tedious. On second thought, since the subtree is entirely persistent, it should be possible to go back to the normal representation where captured and non-captured nodes are treated the same. Crucially, though, the final chain — the non-persistent part — should still be separated. In practice, this would make the proof of `capture` slightly more complex but drastically simplify the proof of `restore`; the proofs of `get` and `set` would be essentially unaffected.

Semi-persistent interface. In the full algorithm of Basile Clément and Gabriel Scherer, two interfaces coexist and can be used simultaneously: the *persistent* interface that we verified and the *semi-persistent* [Conchon and Filliâtre, 2008] interface. In the semi-persistent interface, only the ancestors of the current version are preserved; restoring a version invalidates all the versions that came after. In practice, this interface is sufficient for most backtracking problems.

On its own, the semi-persistent interface is not difficult to formalize — much simpler than the persistent interface. The difficulty lies in the combination of the two interfaces. As a matter of fact, even informally specifying the resulting interface is non-trivial, including for its authors; maintenance it is also a problem. An interesting and challenging future work would consist in figuring out a specification and proving it.

Custom data structures. Recently, Clément et al. [2025] extended persistent stores to support not only references but also *custom data structures* such as dynamic arrays and hash tables. In particular, they define classes of storable data structures as generic interfaces. It would be interesting and possibly not very difficult to verify this extension.

7.5 Snapshottable union-find

We verified a snapshottable *union-find* data structure built on top of snapshottable stores. The union-find data structure is a well-known data structure that can be used to represent disjoint sets or, equivalently, an equivalence relation. For example, it is at the core of ML type inference, which proceeds by repeated unification between type variables. Making it snapshottable allows backtracking, which is often needed in type inference — for example, to type GADTs.

7.5.1 Specification

The specification is given in Figure 7.12.

The exclusive assertion model t *reprs* represents the ownership of instance t and the knowledge that its current state is *reprs*, a mapping that associates to an element the representative of its equivalence class. It is returned by `create` (PUF-CREATE-SPEC) and used by all operations in an imperative fashion.

The persistent assertion snapshot s t *reprs* represents the knowledge that s is a snapshot of t that can be used to `restore` state *reprs* (PUF-RESTORE-SPEC). It can be obtained through `capture` (PUF-CAPTURE-SPEC).

The operations `make` and `union` allows the user to incrementally define an equivalence relation. `make` creates a new element in a new equivalence class. `union` merges two equivalence classes — the union-condition predicate ensures that (1) elements are preserved, (2) only the two merged classes are affected and (3) one of the two representatives was chosen to be the representative of the resulting class.

`repr` returns the representative of an element. `equiv` checks whether two elements are equivalent, *i.e.* are in the same equivalence class, *i.e.* have the same representative.

7.5.2 Implementation

Essentially, the implementation consists of a standard union-find algorithm where normal references are replaced with `Pstore` references to support snapshots. In short, it

$$\mathsf{persistent}\ (\mathsf{snapshot}\ s\ t\ \mathit{reprs})$$

$$\textsc{puf-model-valid}\quad \frac{\mathit{reprs}\,[\mathit{elt}] = \mathsf{Some}\ \mathit{repr} \qquad \mathsf{model}\ t\ \mathit{reprs}}{\mathit{reprs}\,[\mathit{repr}] = \mathsf{Some}\ \mathit{repr}}$$

$$\textsc{puf-model-exclusive}\quad \frac{\mathsf{model}\ t\ \mathit{reprs}_1 \qquad \mathsf{model}\ t\ \mathit{reprs}_2}{\mathsf{False}}$$

$$\textsc{puf-create-spec}\quad \frac{\mathsf{True}}{\texttt{create ()}} \quad t.\ \mathsf{model}\ t\ \emptyset$$

$$\textsc{puf-make-spec}\quad \frac{\mathsf{model}\ t\ \mathit{reprs}}{\texttt{make}\ t} \quad \mathit{elt}.\ \mathsf{model}\ t\ (\mathit{reprs}\,[\mathit{elt} \mapsto \mathit{elt}])$$

$$\textsc{puf-repr-spec}\quad \frac{\mathit{reprs}\,[\mathit{elt}] = \mathsf{Some}\ \mathit{repr} * \mathsf{model}\ t\ \mathit{reprs}}{\texttt{repr}\ t\ \mathit{elt}} \quad \mathit{res}.\ \mathit{res} = \mathit{repr} * \mathsf{model}\ t\ \mathit{reprs}$$

$$\textsc{puf-equiv-spec}\quad \frac{\mathit{reprs}\,[\mathit{elt}_1] = \mathsf{Some}\ \mathit{repr}_1 * \mathit{reprs}\,[\mathit{elt}_2] = \mathsf{Some}\ \mathit{repr}_2 * \mathsf{model}\ t\ \mathit{reprs}}{\texttt{equiv}\ t\ \mathit{elt}_1\ \mathit{elt}_2} \quad \mathit{res}.\ \mathit{res} = \mathsf{decide}\ (\mathit{repr}_1 = \mathit{repr}_2) * \mathsf{model}\ t\ \mathit{reprs}$$

$$\textsc{puf-union-spec}\quad \frac{\mathit{reprs}\,[\mathit{elt}_1] = \mathsf{Some}\ \mathit{repr}_1 * \mathit{reprs}\,[\mathit{elt}_2] = \mathsf{Some}\ \mathit{repr}_2 * \mathsf{model}\ t\ \mathit{reprs}}{\texttt{union}\ t\ \mathit{elt}_1\ \mathit{elt}_2} \quad ().\ \exists\, \mathit{reprs}'.\ \mathsf{model}\ t\ \mathit{reprs}' * \mathsf{union\text{-}condition}\ \mathit{reprs}\ \mathit{repr}_1\ \mathit{repr}_2\ \mathit{reprs}'$$

$$\textsc{puf-capture-spec}\quad \frac{\mathsf{model}\ t\ \mathit{reprs}}{\texttt{capture}\ t} \quad s.\ \mathsf{model}\ t\ \mathit{reprs} * \mathsf{snapshot}\ s\ t\ \mathit{reprs}$$

$$\textsc{puf-restore-spec}\quad \frac{\mathsf{model}\ t\ \mathit{reprs} * \mathsf{snapshot}\ s\ t\ \mathit{reprs}'}{\texttt{restore}\ t\ s} \quad ().\ \mathsf{model}\ t\ \mathit{reprs}'$$

$$\mathsf{union\text{-}condition}\ \mathit{reprs}\ \mathit{repr}_1\ \mathit{repr}_2\ \mathit{reprs}' \triangleq \bigwedge \left[\begin{array}{l}
\mathsf{dom}\ \mathit{reprs} = \mathsf{dom}\ \mathit{reprs}' \\
\\
\forall\, \mathit{elt}\ \mathit{repr}. \\
\mathit{reprs}\,[\mathit{elt}] = \mathsf{Some}\ \mathit{repr} \to \\
\mathit{repr} \neq \mathit{repr}_1 \to \\
\mathit{repr} \neq \mathit{repr}_2 \to \\
\mathit{reprs}'\,[\mathit{elt}] = \mathsf{Some}\ \mathit{repr} \\
\\
\exists\, \mathit{repr}_{12}. \\
(\mathit{repr}_{12} = \mathit{repr}_1 \lor \mathit{repr}_{12} = \mathit{repr}_2) \land \\
\quad \forall\, \mathit{elt}\ \mathit{repr}. \\
\quad \mathit{reprs}\,[\mathit{elt}] = \mathsf{Some}\ \mathit{repr} \to \\
\quad \mathit{repr} = \mathit{repr}_1 \lor \mathit{repr} = \mathit{repr}_2 \to \\
\quad \mathit{reprs}'\,[\mathit{elt}] = \mathsf{Some}\ \mathit{repr}_{12}
\end{array}\right]$$

Figure 7.12: `Puf`: Specification

$$\mathsf{model}\ t\ \mathit{reprs} \triangleq \exists\ \mathit{descrs}.\ \mathsf{pstore\text{-}2.model}\ t\ \mathit{descrs} * \mathsf{consistent}\ \mathit{reprs}\ \mathit{descrs}$$

$$\mathsf{snapshot}\ s\ t\ \mathit{reprs} \triangleq \exists\ \mathit{descrs}.\ \mathsf{pstore\text{-}2.snapshot}\ s\ t\ \mathit{descrs} * \mathsf{consistent}\ \mathit{reprs}\ \mathit{descrs}$$

Figure 7.13: `Puf`: Predicates definition

maintains a forest of parent pointer trees where each tree represents an equivalence class. Path compression is performed during `repr` and the standard rank heuristic is used in `union`.

7.5.3 Ghost state

The definition of model and snapshot is given in Figure 7.11; we omitted the definition of consistent. It is remarkably simple: each predicate relies on the counterpart `Pstore` predicate, asserting that the `Pstore` state is consistent with the actual state through the pure consistent predicate. Most of the reasoning is contained in the consistent theory and therefore takes place outside Iris.

7.6 Related work

Conchon and Filliâtre [2007] implement persistent arrays and persistent union-find in OCaml and verify them in Rocq. They use a shallow embedding of OCaml in Rocq with an explicit heap and express specifications using dependent types. This approach leads to verbose specifications. On the contrary, we benefit from separation logic and provide simpler specifications.

Moine et al. [2022] propose the only formal verification of a transient data structure that we are aware of. They verify both functional correctness and time complexity of a transient stack in separation logic, using CFML [Charguéraud, 2010]. They represent the shared mutable state between snapshots using a dedicated assertion. Thanks to Iris ghost state, we do not need such an assertion: our specifications are simpler.

Chapter 8

Rcfd: Parallelism-safe file descriptor

As mentioned in Section 4.2.3, the Rcfd module from the Eio library is particularly interesting in several respects. Not only does it justify the introduction of generative constructors in OCaml, but it also demonstrates the use of Iris for expressing realistic concurrent protocols.

8.1 Specification

The Rcfd module provides a parallelism-safe wrapper around a file descriptor (FD) relying internally on reference-counting. Interestingly, it is used in Eio in two different ways, more precisely two different ownership regimes: (1) in the *free regime*, any domain can try to access or close the FD; (2) in the *strict regime*, any domain can try to access the FD but only the owner domain can close it — and is responsible for closing it. Actually, in both regimes, "closing" the wrapper only flags it as closed but does not immediately close the wrapped FD; it will be closed only once it is possible, meaning all ongoing accesses are done. To verify all uses, the specification of Rcfd, given in Figure 8.1, supports both regimes.

The persistent assertion inv t owned fd Ψ represents the knowledge that t is a valid inhabitant of `Rcfd.t` wrapping the FD fd; it is required by all operations. The boolean owned corresponds to the ownership regime: loose when owner is false and strict when it is true. Ψ is an arbitrary *fractional* predicate controlled by t in the following sense: (1) when t is created using `make`, the user has to provide the full predicate Ψ 1 (RCFD-MAKE-SPEC); (2) when trying to access fd using `use`, the user may temporary get a fraction of Ψ if t has not been flagged as closed (RCFD-USE-SPEC).

More precisely, to call `use`, the user has to supply a *closed* function, that is called if the FD has been flagged as closed, and an *open* function, that is called if the FD is still open. Consequently, the specification RCFD-USE-SPEC requires a weakest precondition for both functions. To connect these weakest preconditions to the postcondition, the user can choose an arbitrary predicate X parameterized by a boolean indicating whether *closed* (false) or *open* (true) was called.

The exclusive assertion owner t represents the ownership of t in the strict regime. It is returned by `make` if the user chooses this regime (RCFD-MAKE-SPEC).

`close` t flags t as closed, if it is not already. RCFD-CLOSE-SPEC requires (1) owner t in the strict regime and (2) proving that the full ownership of Ψ entails the full ownership of fd (RCFD-CLOSE-SPEC), which is necessary to call `Unix.close`. It yields closing t, a persistent assertion attesting that t has been flagged as closed.

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ \mathit{owned}\ \mathit{fd}\ \Psi) \qquad\qquad \mathsf{persistent}\ (\mathsf{closing}\ t)$$

$$\frac{\mathsf{owner}\ t \qquad \mathsf{owner}\ t}{\mathsf{False}}\ \textsc{rcfd-owner-exclusive} \qquad\qquad \frac{\mathsf{owner}\ t \qquad \mathsf{closing}\ t}{\mathsf{False}}\ \textsc{rcfd-owner-closing}$$

$$\textsc{rcfd-make-spec}\quad \begin{array}{l} \Psi\ 1 \\ \hline \texttt{make}\ \mathit{fd} \\ \hline t.\ \mathsf{inv}\ t\ \mathit{owned}\ \mathit{fd}\ \Psi\ * \\ \quad \textbf{if}\ \mathit{owned}\ \textbf{then}\ \mathsf{owner}\ t\ \textbf{else}\ \mathsf{True} \end{array}$$

$$\textsc{rcfd-use-spec}\quad \begin{array}{l} \mathsf{inv}\ t\ \mathit{owned}\ \mathit{fd}\ \Psi\ * \\ \mathsf{wp}\ \mathit{closed}\ ()\ \{\, \mathrm{X}\ \mathsf{false}\,\}\ * \\ (\forall\, q.\ \Psi\ q \mathrel{-\!\!*} \mathsf{wp}\ \mathit{open}\ \mathit{fd}\ \{\, \mathit{res}.\ \Psi\ q * \mathrm{X}\ \mathsf{true}\ \mathit{res}\,\}) \\ \hline \texttt{use}\ t\ \mathit{closed}\ \mathit{open} \\ \hline \mathit{res}.\ \exists\, b.\ \mathrm{X}\ b\ \mathit{res} \end{array}$$

$$\textsc{rcfd-use-spec-owner}\quad \begin{array}{l} \mathsf{inv}\ t\ \mathit{owned}\ \mathit{fd}\ \Psi\ * \\ \mathsf{owner}\ t\ * \\ (\forall\, q.\ \Psi\ q \mathrel{-\!\!*} \mathsf{wp}\ \mathit{open}\ \mathit{fd}\ \{\, \mathit{res}.\ \Psi\ q * \mathrm{X}\ \mathit{res}\,\}) \\ \hline \texttt{use}\ t\ \mathit{closed}\ \mathit{open} \\ \hline \mathit{res}.\ \mathrm{X}\ \mathit{res} \end{array}$$

$$\textsc{rcfd-use-spec-closing}\quad \begin{array}{l} \mathsf{inv}\ t\ \mathit{owned}\ \mathit{fd}\ \Psi\ * \\ \mathsf{closing}\ t\ * \\ \mathsf{wp}\ \mathit{closed}\ ()\ \{\, \mathrm{X}\,\} \\ \hline \texttt{use}\ t\ \mathit{closed}\ \mathit{open} \\ \hline \mathit{res}.\ \mathrm{X}\ \mathit{res} \end{array}$$

$$\textsc{rcfd-close-spec}\quad \begin{array}{l} \mathsf{inv}\ t\ \mathit{owned}\ \mathit{fd}\ \Psi\ * \\ (\textbf{if}\ \mathit{owned}\ \textbf{then}\ \mathsf{owner}\ t\ \textbf{else}\ \mathsf{True})\ * \\ (\Psi\ 1 \mathrel{-\!\!*} \exists\, \mathit{chars}.\ \mathsf{unix.fd}\ \mathit{fd}\ 1\ \mathit{chars}) \\ \hline \texttt{close}\ t \\ \hline b.\ \mathsf{closing}\ t\ * \\ \quad \textbf{if}\ \mathit{owned}\ \textbf{then}\ b = \mathsf{true}\ \textbf{else}\ \mathsf{True} \end{array}$$

$$\textsc{rcfd-close-spec-closing}\quad \begin{array}{l} \mathsf{inv}\ t\ \mathsf{false}\ \mathit{fd}\ \Psi\ * \\ \mathsf{closing}\ t \\ \hline \texttt{close}\ t \\ \hline \mathsf{false}.\ \mathsf{True} \end{array}$$

$$\textsc{rcfd-remove-spec}\quad \begin{array}{l} \mathsf{inv}\ t\ \mathit{owned}\ \mathit{fd}\ \Psi\ * \\ \textbf{if}\ \mathit{owned}\ \textbf{then}\ \mathsf{owner}\ t\ \textbf{else}\ \mathsf{True} \\ \hline \texttt{remove}\ t \\ \hline o.\ \mathsf{closing}\ t\ * \\ \quad \textbf{if}\ \mathit{owned}\ \textbf{then} \\ \quad\quad o = \mathsf{Some}\ \mathit{fd} * \Psi\ 1 \\ \quad \textbf{else} \\ \quad\quad \textbf{match}\ o\ \textbf{with} \\ \quad\quad \mid \mathsf{None} \Rightarrow \\ \quad\quad\quad \mathsf{True} \\ \quad\quad \mid \mathsf{Some}\ \mathit{fd\_} \Rightarrow \\ \quad\quad\quad \mathit{fd\_} = \mathit{fd} * \Psi\ 1 \\ \quad\quad \textbf{end} \end{array}$$

$$\textsc{rcfd-remove-spec-closing}\quad \begin{array}{l} \mathsf{inv}\ t\ \mathsf{false}\ \mathit{fd}\ \Psi\ * \\ \mathsf{closing}\ t \\ \hline \texttt{remove}\ t \\ \hline \mathsf{None}.\ \mathsf{True} \end{array}$$

Figure 8.1: Specification (excerpt)

Alternatively, instead of closing the FD, `remove` tries to retrieve the full ownership of Ψ (RCFD-REMOVE-SPEC). To achieve this, the operation exploits the same mechanism as `close` — flagging t as closed — but also waits until all `use` calls are done.

8.2 Protocol

Thomas Leonard, the author of `Rcfd`, suggested verifying it to make sure the informal concurrent protocol he described in the OCaml interface was correct. This protocol introduces a notion of monotonic logical state — modeled in Iris using a specific resource algebra [Timany and Birkedal, 2021] — to describe the evolution of a FD. Originally, there were four logical states but we found that only three are necessary for the verification: Open, ClosingUsers and ClosingNoUsers.

In the Open state, the FD is available for use, meaning any domain can access it through `use`. Physically, this corresponds to the `Open` constructor.

When some domain flags the FD as closed through `close` or `remove`, the state transitions from Open to ClosingUsers. Crucially, there can only be one such domain. In this state, the FD is not really closed yet because of ongoing `use` operations. Physically, this logical transition corresponds to switching from the `Open` to the `Closing` constructor (see Section 4.2.3) using `Atomic.Loc.compare_and_set`.

Once all `use` operations have finished, when the reference-count reaches zero, it is time to actually "close" the FD by calling the function carried by the `Closing` constructor. This has to be done only once. The "closing" domain is the one that succeeds in updating the `Closing` constructor (to a new one carrying a no-op function) using `Atomic.Loc.compare_and_set`. At this point, the state transitions from ClosingUsers to ClosingNoUsers and the wrapper no longer owns the FD.

8.3 Generative contructors

In Section 4.2.3, we examined the implementation of `close` to justify the introduction of *generative constructors*; in particular, the `Open` constructor has to be generative. In doing so, we overlooked part of it. We now consider the full implementation, given in Figure 8.2.

In the `then` branch of the outermost conditional, the "closing" domain tries to update the state again using `Atomic.Loc.compare_and_set`. If, and only if, it succeeds, it actually closes the FD by calling `Unix.close`. One might ask whether this is safe since another domain could have seen the `Closing` it just published and called the associated "closing" function, that is `Unix.close`. The reason is twofold: (1) during the first `Atomic.Loc.compare_and_set`, the domain transfers the resource needed to call `Unix.close` (see RCFD-CLOSE-SPEC) to `t`; (2) during the second `Atomic.Loc.compare_and_set`, it retrieves this resource, which is still there because the observed state is physically the same and *therefore the "closing" functions are the same*. However, we have seen in Section 4.2.3 that normal constructors do not enjoy such a property. As a result, the `Closing` constructor also has to be generative.

```
let closed =
  Closing (fun () -> ())
let close t =
  match t.state with
  | Closing _ ->
      false
  | Open fd as state ->
      let close () = Unix.close fd in
      let new_state = Closing close in
      if Atomic.Loc.compare_and_set [%atomic.loc t.state] state new_state then (
        if t.ops == 0
        && Atomic.Loc.compare_and_set [%atomic.loc t.state] new_state closed
        then
          close () ;
        true
      ) else (
        false
      )
```

Figure 8.2: `close` implementation

$$\frac{\textsf{tokens-auth}\ \gamma\ \Psi\ ops}{0 \leq ops}\ \textsc{tokens-auth-valid}$$

$$\frac{\textsf{tokens-auth}\ \gamma\ \Psi\ 0}{\Psi\ 1}\ \textsc{tokens-auth-consume}$$

$$\frac{\textsf{tokens-auth}\ \gamma\ \Psi\ ops}{\dot{\Rrightarrow}\ \exists\, q.\ \textsf{tokens-auth}\ \gamma\ \Psi\ (ops + 1) * \textsf{tokens-frag}\ \gamma\ q * \Psi\ q}\ \textsc{tokens-update-alloc}$$

$$\frac{\textsf{tokens-auth}\ \gamma\ \Psi\ ops \quad \textsf{tokens-frag}\ \gamma\ q \quad \Psi\ q}{\dot{\Rrightarrow}\ \textsf{tokens-auth}\ \gamma\ \Psi\ (ops - 1)}\ \textsc{tokens-update-dealloc}$$

Figure 8.3: tokens theory

$$\mathsf{persistent}\ (\mathsf{lstate\text{-}lb}\ \gamma\ \mathit{lstate})$$

$$\textsc{lstate-lb-get}\quad \frac{\mathsf{lstate\text{-}auth}\ \gamma\ \mathit{lstate}}{\mathsf{lstate\text{-}lb}\ \gamma\ \mathit{lstate}}$$

$$\textsc{lstate-lb-mono}\quad \frac{\mathit{lstate}' \rightsquigarrow \mathit{lstate} \qquad \mathsf{lstate\text{-}lb}\ \gamma\ \mathit{lstate}}{\mathsf{lstate\text{-}lb}\ \gamma\ \mathit{lstate}'}$$

$$\textsc{lstate-valid}\quad \frac{\mathsf{lstate\text{-}auth}\ \gamma\ \mathit{lstate} \qquad \mathsf{lstate\text{-}lb}\ \gamma\ \mathit{lstate}'}{\mathit{lstate}' \rightsquigarrow^{*} \mathit{lstate}}$$

$$\textsc{lstate-valid-closing-users}\quad \frac{\mathsf{lstate\text{-}auth}\ \gamma\ \mathit{lstate} \qquad \mathsf{lstate\text{-}lb}\ \gamma\ \mathsf{ClosingUsers}}{\mathit{lstate} \neq \mathsf{Open}}$$

$$\textsc{lstate-valid-closing-no-users}\quad \frac{\mathsf{lstate\text{-}auth}\ \gamma\ \mathit{lstate} \qquad \mathsf{lstate\text{-}lb}\ \gamma\ \mathsf{ClosingNoUsers}}{\mathit{lstate} = \mathsf{ClosingNoUsers}}$$

$$\textsc{lstate-update-close-users}\quad \frac{\mathsf{lstate\text{-}auth}\ \gamma\ \mathsf{Open} \qquad \textbf{if}\ \gamma.\mathsf{owned}\ \textbf{then}\ \mathsf{owner}\ \gamma\ \textbf{else}\ \mathsf{True}}{\mathrel{|\!\dot{\Rrightarrow}}\ \mathsf{lstate\text{-}auth}\ \gamma\ \mathsf{ClosingUsers}}$$

$$\textsc{lstate-update-close-no-users}\quad \frac{\mathsf{lstate\text{-}auth}\ \gamma\ \mathsf{ClosingUsers}}{\mathrel{|\!\dot{\Rrightarrow}}\ \mathsf{lstate\text{-}auth}\ \gamma\ \mathsf{ClosingNoUsers}}$$

Figure 8.4: lstate theory

$$\textsc{owner-exclusive}\quad \frac{\mathsf{owner}\ \gamma \qquad \mathsf{owner}\ \gamma}{\mathsf{False}}$$

$$\textsc{owner-lstate-auth}\quad \frac{\mathsf{owner}\ \gamma \qquad \mathsf{lstate\text{-}auth}\ \gamma\ \mathit{lstate}}{\mathit{lstate} = \mathsf{Open}}$$

$$\textsc{owner-lstate-lb}\quad \frac{\mathsf{owner}\ \gamma \qquad \mathsf{lstate\text{-}lb}\ \gamma\ \mathsf{ClosingUsers}}{\mathsf{False}}$$

Figure 8.5: owner theory

$$
\begin{aligned}
&\mathsf{inv\text{-}lstate\text{-}open}\ \gamma\ \Psi\ \mathit{state}\ \mathit{ops} \triangleq\\
&\quad \mathsf{tokens\text{-}auth}\ \gamma\ \Psi\ \mathit{ops} *\\
&\quad \mathit{state} = \texttt{`Open@}\gamma.\mathsf{open}\,\texttt{[}\gamma.\mathsf{fd}\texttt{]}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{inv\text{-}lstate\text{-}closing\text{-}users}\ \gamma\ \Psi\ \mathit{state}\ \mathit{ops} \triangleq\\
&\quad \exists\, \mathit{fn}.\\
&\quad \mathsf{tokens\text{-}auth}\ \gamma\ \Psi\ \mathit{ops} *\\
&\quad \mathit{state} = \texttt{`Closing[}\mathit{fn}\texttt{]} *\\
&\quad 0 < \mathit{ops} *\\
&\quad \left(\Psi\ 1 \twoheadrightarrow \mathsf{wp}\ \mathit{fn}\ ()\ \left\{\ ().\ \mathsf{True}\ \right\}\right)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{inv\text{-}lstate\text{-}closing\text{-}no\text{-}users}\ \mathit{state} \triangleq\\
&\quad \exists\, \mathit{fn}.\\
&\quad \mathit{state} = \texttt{`Closing[}\mathit{fn}\texttt{]} *\\
&\quad \mathsf{wp}\ \mathit{fn}\ ()\ \left\{\ ().\ \mathsf{True}\ \right\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{inv\text{-}lstate}\ \gamma\ \Psi\ \mathit{state}\ \mathit{lstate}\ \mathit{ops} \triangleq\\
&\quad \textbf{match}\ \mathit{lstate}\ \textbf{with}\\
&\quad \mid \mathsf{Open} \Rightarrow\\
&\qquad \mathsf{inv\text{-}lstate\text{-}open}\ \gamma\ \Psi\ \mathit{state}\ \mathit{ops}\\
&\quad \mid \mathsf{ClosingUsers} \Rightarrow\\
&\qquad \mathsf{inv\text{-}lstate\text{-}closing\text{-}users}\ \gamma\ \Psi\ \mathit{state}\ \mathit{ops}\\
&\quad \mid \mathsf{ClosingNoUsers} \Rightarrow\\
&\qquad \mathsf{inv\text{-}lstate\text{-}closing\text{-}no\text{-}users}\ \mathit{state}\\
&\quad \textbf{end}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{inv\text{-}inner}\ \ell\ \gamma\ \Psi \triangleq\\
&\quad \exists\, \mathit{state}\ \mathit{lstate}\ \mathit{ops}.\\
&\quad \ell.\mathsf{ops} \mapsto \mathit{ops} *\\
&\quad \ell.\mathsf{state} \mapsto \mathit{state} *\\
&\quad \mathsf{lstate\text{-}auth}\ \gamma\ \mathit{lstate} *\\
&\quad \mathsf{inv\text{-}lstate}\ \gamma\ \Psi\ \mathit{state}\ \mathit{lstate}\ \mathit{ops}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{inv}\ t\ \mathit{owned}\ \mathit{fd}\ \Psi \triangleq\\
&\quad \exists\, \ell\ \gamma.\\
&\quad t = \ell *\\
&\quad \mathit{owned} = \gamma.\mathsf{owned} *\\
&\quad \mathit{fd} = \gamma.\mathsf{fd} *\\
&\quad \mathsf{meta}\ \ell\ \top\ \gamma *\\
&\quad \boxed{\mathsf{inv\text{-}inner}\ \ell\ \gamma\ \Psi}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{owner}\ t \triangleq\\
&\quad \exists\, \ell\ \gamma.\\
&\quad t = \ell *\\
&\quad \mathsf{meta}\ \ell\ \top\ \gamma *\\
&\quad \mathsf{owner}\ \gamma
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{closing}\ t \triangleq\\
&\quad \exists\, \ell\ \gamma.\\
&\quad t = \ell *\\
&\quad \mathsf{meta}\ \ell\ \top\ \gamma *\\
&\quad \mathsf{lstate\text{-}lb}\ \gamma\ \mathsf{ClosingUsers}
\end{aligned}
$$

Figure 8.6: Predicates definition

8.4 Ghost state

The definition of the predicates (inv, *owned* and closing), given in Figure 8.6, realize the informal protocol of Section 8.2. It involves three ghost theories.

tokens theory (Figure 8.3). This theory is responsible for the Ψ bookkeeping.

tokens-auth γ Ψ *ops* represents the Ψ stock; *ops* is the current number of borrowers. When there is no borrower, the stock can be consumed to obtain Ψ 1 (TOKENS-AUTH-CONSUME).

tokens-frag γ q represents a borrow of fraction q. To get a borrow, one can use TOKENS-UPDATE-ALLOC, which also yields a fraction of Ψ. To end a borrow, one can use TOKENS-UPDATE-DEALLOC, which symmetrically requires to give back the Ψ fraction.

tokens-auth appears in the Open and ClosingUsers states. In the ClosingNoUsers state, the FD has been "closed", meaning the stock has been consumed. tokens-frag does not appear in Figure 8.6; indeed, it is only used locally, especially in the proof of the `use` operation.

lstate theory (Figure 8.4). This theory, similarly to Timany and Birkedal [2021], is responsible for keeping track of the *monotonic* logical state.

lstate-auth γ *lstate* states that *lstate* is the current logical state. It can be updated using LSTATE-UPDATE-CLOSE-USERS and LSTATE-UPDATE-CLOSE-NO-USERS.

lstate-lb γ *lstate* represents a persistent lower bound on the logical state; in other words, it attests that the current logical state is at least lstate (LSTATE-VALID). It can be obtained by taking a snapshot of lstate-auth (LSTATE-LB-GET). For example, lstate-lb γ ClosingUsers rules out the Open state (LSTATE-VALID-CLOSING-USERS) while lstate-lb γ ClosingNoUsers rules out everything but the ClosingNoUsers state (LSTATE-VALID-CLOSING-NO-USERS).

lstate-auth is stored in the Iris invariant of inv to be shared between domains. lstate-lb is used in closing as a witness that the logical state is at least ClosingUsers, *i.e.* the FD has been flagged as closed.

owner theory (Figure 8.5). This theory is responsible for constraining the logical state in the strict ownership regime. It features a single exclusive assertion, owner γ, which acts like a key possessed by the owner through owner. As long as the owner holds the key, the logical state must be Open (OWNER-LSTATE-AUTH, OWNER-LSTATE-LB). To unlock the logical state and let it step beyond Open, the owner has to relinquish the key (LSTATE-UPDATE-CLOSE-USERS).

Chapter 9

`Saturn`: A library of standard lock-free data structures

We verified a collection of standard (mostly) lock-free data structures including stacks, queues (list-based, array-based and stack-based), bags and work-stealing deques. Most of them are taken from the `Saturn` [Karvonen and Morel, 2025b], `Eio` [Madhavapeddy and Leonard, 2025] and `Picos` [Karvonen, 2025c] libraries. These data structures are meant to be used as is or adapted to fit specific needs. To cover a wide range of use cases, we provide specialized variants: bounded or unbounded, single-producer (SP) or multi-producer (MP), single-consumer (SC) or multi-consumer (MC).

Given the sheer number of data structures, we do not detail all of them. We focus on the most interesting ones, especially those involving non-fixed linearization points [Dongol and Derrick, 2014].

9.1 Stacks

We verified three variants of the Treiber stack [Treiber, 1986]: (1) unbounded MPMC, (2) bounded MPMC, (3) closable unbounded MPMC. This last variant features a closing mechanism: at some point, some thread can decide to close the stack, retrieving the current content and preventing others from operating on it; we use it in Section 10.6 to represent a set of vertex successors in the context of a concurrent graph implementation.

As explained in Section 4.2.3, the three verified stacks use generative constructors to prevent sharing. One may ask whether it would be easier to use a mutable version of lists instead. From the programmer's perspective, this is unsatisfactory because (1) the compiler will typically emit warnings complaining that the mutability is not exploited and (2) it does not really reflect the intent, *i.e.* we want precise guarantees for physical equality, not modify the list. From the verification perspective, this is also unsatisfactory because the mutable representation is more complex to write and reason about: pointers and points-to assertions versus pure Rocq list.

Although verified stacks may seem like a not-so-new contribution, it is, as far as we know, the first verification of realistic OCaml implementations. For comparison, the exemplary concurrent stacks verified in Iris [Iris development team, 2025b] all suffer from the same flaw: they need to introduce indirections (pointers) to be able to use the compare-and-set primitive.

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ \iota)$$

$$\textsc{mpmc-queue-1-model-exclusive}\quad \frac{\mathsf{model}\ t\ vs_1 \qquad \mathsf{model}\ t\ vs_2}{\mathsf{False}}$$

$$\textsc{mpmc-queue-1-create-spec}\quad \frac{\dfrac{\mathsf{True}}{\mathtt{create}\ ()}}{t.\ \mathsf{inv}\ t\ \iota \ast \mathsf{model}\ t\ []}$$

$$\textsc{mpmc-queue-1-is-empty-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ \iota \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \mathtt{is\_empty}\ t\ ⨟\ \iota \\ \hline \mathsf{model}\ t\ vs \\ \hdashline res.\ res = \mathsf{decide}\ (vs = []) \end{array}$$

$$\textsc{mpmc-queue-1-push-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ \iota \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \mathtt{push}\ t\ v\ ⨟\ \iota \\ \hline \mathsf{model}\ t\ (vs +\!\!\!+ [v]) \\ \hdashline ().\ \mathsf{True} \end{array}$$

$$\textsc{mpmc-queue-1-pop-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ \iota \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \mathtt{pop}\ t\ ⨟\ \iota \\ \hline \mathsf{model}\ t\ (\mathsf{tail}\ vs) \\ \hdashline res.\ res = \mathsf{head}\ vs \end{array}$$

Figure 9.1: `Mpmc_queue_1`: Specification

9.2 List-based queues

List-based queues are represented using a list of nodes, each containing a value. The canonical list-based queue is the Michael-Scott queue [Michael and Scott, 1996], of which we verified four variants: unbounded MPMC, bounded MPMC, unbounded MPSC and unbounded SPMC. The MPMC variant is used in Sections 10.3 and 10.4.3; the SPMC is used in Section 9.6.

In the following, we focus on the MPMC variant.

9.2.1 Specification

The specification is given in Figure 9.1. The persistent assertion $\mathsf{inv}\ t\ \iota$ represents the knowledge that t is a valid queue. It is return by `create` (MPMC-QUEUE-1-CREATE-SPEC) and required by all operations. The exclusive assertion $\mathsf{model}\ t\ vs$ represents the ownership of queue t and the knowledge that it contains values vs. It is also returned by `create` and accessed atomically by all operations. `is_empty` (MPMC-QUEUE-1-IS-EMPTY-SPEC) atomically reads vs and returns whether it is empty. `push` (MPMC-QUEUE-1-PUSH-SPEC) and `pop` (MPMC-QUEUE-1-POP-SPEC) atomically update vs.

9.2.2 Implementation

In the Iris literature, Vindum and Birkedal [2021] established contextual refinement of the Michael-Scott queue while Mulder and Krebbers [2023] proved logical atomicity. However, the implementation we verified differs from the original one in several respects. As we explain in Section 9.2.3, this requires to redesign and extend the previous Iris

$$\text{persistent } (\text{saved-pred } \gamma \ \Psi)$$

SAVED-PRED-ALLOC

$$\dot{\Rrightarrow} \exists\, \gamma.\, \text{saved-pred } \gamma \ \Psi$$

SAVED-PRED-AGREE

$$\frac{\text{saved-pred } \gamma \ \Psi_1 \qquad \text{saved-pred } \gamma \ \Psi_2}{\triangleright(\Psi_1 \ x \equiv \Psi_2 \ x)}$$

Figure 9.2: Mpmc_queue_1: saved-pred theory

invariants.

Efficient representation. The Michael-Scott essentially consists of a singly linked list of nodes that only grows over time. The previously verified implementations, implemented in HeapLang, use a double indirection to represent the list [Vindum and Birkedal, 2021, Figure 2]. Similarly to the Treiber stack, this is made so as to be able to use the compare-and-set primitive of HeapLang.

In OCaml, this would correspond to introducing extra atomic references (`Atomic.t`) between the nodes. Using atomic record fields (see Section 2.3.2.2), we can represent the list more efficiently, without the extra indirection. However, there is one subtlety: in this new representation: we need to clear the outdated nodes so that their value is no longer reachable and can be garbage-collected, *i.e.* to prevent memory leak. Consequently, contrary to previously verified implementations, the nodes are mutable.

External linearization. Our work also revealed another interesting aspect that is not addressed in the literature, as far as we know. None of the previously verified implementations deal with the `is_empty` operation, that consists in reading the sentinel node and checking whether it has a successor. It it has no successor, it is necessarily the last node of the chain, hence the queue is empty. If it does have a successor, `is_empty` returns `false`, meaning we must have observed a non-empty queue. However, this last part is more tricky than it may seem. Indeed, it may happen that (1) we read the sentinel while the queue is empty, (2) other operations fill and empty again the queue so that the sentinel is outdated, (3) we read the successor of the former sentinel while the queue is still empty. The crucial point here is that `is_empty` is linearized when the first `push` operation filled the queue. In other words, the linearization point of `is_empty` is triggered by another operation; this is called an *external linearization point*.

9.2.3 Ghost state

The definition of inv and model is given in Figure 9.7. It relies on five simple ghost theories: saved-pred, history, front, model and waiters.

saved-pred theory (Figure 9.2). The persistent saved-pred γ Ψ represents the knowledge that the logical name γ is bound to the Iris predicate Ψ; it is a basic example of higher-order ghost state. Due to a restriction on the latter, two Iris predicates with the same name are only (extensionally) equal under the later modality (SAVED-PRED-AGREE).

$$\mathsf{persistent}\ (\mathsf{history\text{-}at}\ \gamma\ i\ \mathit{node})$$

$$\textsc{history-at-get}\ \frac{\mathit{hist}\,[i] = \mathsf{Some}\ \mathit{node} \qquad \mathsf{history\text{-}auth}\ \gamma\ \mathit{hist}}{\mathsf{history\text{-}at}\ \gamma\ i\ \mathit{node}}$$

$$\textsc{history-at-lookup}\ \frac{\mathsf{history\text{-}auth}\ \gamma\ \mathit{hist} \qquad \mathsf{history\text{-}at}\ \gamma\ i\ \mathit{node}}{\mathit{hist}\,[i] = \mathsf{Some}\ \mathit{node}}$$

$$\textsc{history-at-agree}\ \frac{\mathsf{history\text{-}at}\ \gamma\ i\ \mathit{node}_1 \qquad \mathsf{history\text{-}at}\ \gamma\ i\ \mathit{node}_2}{\mathit{node}_1 = \mathit{node}_2}$$

$$\textsc{history-update}\ \frac{\mathsf{history\text{-}auth}\ \gamma\ \mathit{hist}}{\dot{\Rrightarrow}\ \mathsf{history\text{-}auth}\ \gamma\ (\mathit{hist} +\!\!+ [\mathit{node}]) \ast \mathsf{history\text{-}at}\ \gamma\ (\mathsf{length}\ \mathit{hist})\ \mathit{node}}$$

Figure 9.3: `Mpmc_queue_1`: history theory

$$\mathsf{persistent}\ (\mathsf{front\text{-}lb}\ \gamma\ i)$$

$$\textsc{front-lb-get}\ \frac{\mathsf{front\text{-}auth}\ \gamma\ i}{\mathsf{front\text{-}lb}\ \gamma\ i}$$

$$\textsc{front-lb-le}\ \frac{i' \leq i \qquad \mathsf{front\text{-}lb}\ \gamma\ i}{\mathsf{front\text{-}lb}\ \gamma\ i'}$$

$$\textsc{front-lb-valid}\ \frac{\mathsf{front\text{-}auth}\ \gamma\ i_1 \qquad \mathsf{front\text{-}lb}\ \gamma\ i_2}{i_2 \leq i_1}$$

$$\textsc{front-update}\ \frac{i \leq i' \qquad \mathsf{front\text{-}auth}\ \gamma\ i}{\dot{\Rrightarrow}\ \mathsf{front\text{-}auth}\ \gamma\ i'}$$

Figure 9.4: `Mpmc_queue_1`: front theory

$$\textsc{model-1-exclusive}\ \frac{\mathsf{model}_1\ \gamma\ \mathit{vs}_1 \qquad \mathsf{model}_1\ \gamma\ \mathit{vs}_2}{\mathsf{False}}$$

$$\textsc{model-2-exclusive}\ \frac{\mathsf{model}_2\ \gamma\ \mathit{vs}_1 \qquad \mathsf{model}_2\ \gamma\ \mathit{vs}_2}{\mathsf{False}}$$

$$\textsc{model-agree}\ \frac{\mathsf{model}_1\ \gamma\ \mathit{vs}_1 \qquad \mathsf{model}_2\ \gamma\ \mathit{vs}_2}{\mathit{vs}_1 = \mathit{vs}_2}$$

$$\textsc{model-update}\ \frac{\mathsf{model}_1\ \gamma\ \mathit{vs}_1 \qquad \mathsf{model}_2\ \gamma\ \mathit{vs}_2}{\dot{\Rrightarrow}\ \mathsf{model}_1\ \gamma\ \mathit{vs} \ast \mathsf{model}_2\ \gamma\ \mathit{vs}}$$

Figure 9.5: `Mpmc_queue_1`: model theory

$$\textsc{waiters-insert}\ \frac{\mathsf{waiters\text{-}auth}\ \gamma\ \mathit{waiters}}{\dot{\Rrightarrow}\ \exists\ \mathit{waiter}.\ \mathsf{waiters\text{-}auth}\ \gamma\ (\mathit{waiters}\,[\mathit{waiter} \mapsto i]) \ast \mathsf{saved\text{-}pred}\ \mathit{waiter}\ \Psi \ast \mathsf{waiters\text{-}at}\ \gamma\ \mathit{waiter}\ i}$$

$$\textsc{waiters-delete}\ \frac{\mathsf{waiters\text{-}auth}\ \gamma\ \mathit{waiters} \qquad \mathsf{waiters\text{-}at}\ \gamma\ \mathit{waiter}\ i}{\dot{\Rrightarrow}\ \mathit{waiters}\,[\mathit{waiter}] = \mathsf{Some}\ i \ast \mathsf{waiters\text{-}auth}\ \gamma\ (\mathsf{delete}\ \mathit{waiter}\ \mathit{waiters})}$$

Figure 9.6: `Mpmc_queue_1`: waiters theory

$$
\begin{array}{l}
\mathsf{waiter\text{-}au}\ \gamma\ \Psi \triangleq \\
\quad \langle\, vs.\ \mathsf{model}_1\ \gamma\ vs \mid \mathsf{model}_1\ \gamma\ vs \Rrightarrow \Psi\ (\mathsf{decide}\ \ (vs = [\,]))\,\rangle_{\gamma.\mathsf{inv}} \\
\\
\mathsf{waiter\text{-}model}\ \gamma\ past\ waiter\ i \triangleq \\
\quad \exists\ \Psi. \\
\quad \mathsf{saved\text{-}pred}\ waiter\ \Psi\ * \\
\quad \textbf{if}\ \mathsf{decide}\ (i < \mathsf{length}\ past)\ \textbf{then} \\
\qquad \Psi\ \mathsf{false} \\
\quad \textbf{else} \\
\qquad \mathsf{waiter\text{-}au}\ \gamma\ \Psi \\
\\
\mathsf{inv\text{-}inner}\ \ell\ \gamma \triangleq \\
\quad \exists\ hist\ past\ front\ nodes\ back\ vs\ waiters. \\
\quad hist = past +\!\!+ [front] +\!\!+ nodes\ * \\
\quad back \in hist\ * \\
\quad \ell.\mathsf{front} \mapsto front\ * \\
\quad \ell.\mathsf{back} \mapsto back\ * \\
\quad \mathsf{xtchain}\ \ \{\,\mathsf{tag}\text{: }\S\mathtt{Node}\text{; }\mathsf{size}\text{: }2\,\}\ \ hist\ \S\mathtt{Null}\ * \\
\quad \left(\bigstar_{node, v \in nodes, vs}\ node.\mathsf{data} \mapsto v\right)\ * \\
\quad \mathsf{history\text{-}auth}\ \gamma\ hist\ * \\
\quad \mathsf{front\text{-}auth}\ \gamma\ (\mathsf{length}\ past)\ * \\
\quad \mathsf{model}_2\ \gamma\ vs\ * \\
\quad \mathsf{waiters\text{-}auth}\ \gamma\ waiters\ * \\
\quad \left(\bigstar_{waiter \mapsto i \in waiters}\ \mathsf{waiter\text{-}model}\ \gamma\ past\ waiter\ i\right) \\
\\
\mathsf{inv}\ t\ \iota \triangleq \\
\quad \exists\ \ell\ \gamma. \\
\quad t = \ell\ * \\
\quad \iota = \gamma.\mathsf{inv}\ * \\
\quad \mathsf{meta}\ \ell\ \top\ \gamma\ * \\
\quad \boxed{\mathsf{inv\text{-}inner}\ \ell\ \gamma}^{\gamma.\mathsf{inv}}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{model}\ t\ vs \triangleq \\
\quad \exists\ \ell\ \gamma. \\
\quad t = \ell\ * \\
\quad \mathsf{meta}\ \ell\ \top\ \gamma\ * \\
\quad \mathsf{model}_1\ \gamma\ vs
\end{array}
$$

Figure 9.7: `Mpmc_queue_1`: Predicates definition

history theory (Figure 9.3). This theory is responsible for keeping track of all the nodes involved in the history of the queue. The assertion history-auth γ *hist* represents the ownership of the full history *hist*, which can only grow (HISTORY-UPDATE). The persistent assertion history-at γ i *node* represents the knowledge that the i-th node of the queue is *node* (HISTORY-AT-LOOKUP).

front theory (Figure 9.4). This theory is responsible for enforcing the monotonicity of the front index, *i.e.* the index of the sentinel node, by tying it to the front-auth predicate (FRONT-UPDATE). The persistent assertion front-lb γ i represents the knowledge that i is a lower bound of the current front index (FRONT-LB-VALID).

model theory (Figure 9.5). This theory is responsible for keeping track of the logical content of the queue through two agreeing (MODEL-AGREE) parts model_1 and model_2. They are respectively stored in model and inv.

waiters theory (Figure 9.6). This theory is responsible for keeping track of the `is_empty` operations to be linearized, called the waiters. The assertion waiters-auth γ *waiters*, stored in inv, keeps track of all the waiters. The assertion waiters-at γ *waiter* represents the ownership of a waiter. When a `is_empty` operation reads the sentinel node, it registers itself as a waiter (WAITERS-INSERT) and stores the atomic update (see Section 3.8) materializing its linearization point into inv; then, when it reads the successor of the sentinel, it cancels the waiter (WAITERS-DELETE) and retrieves the atomic update (that may or may not have been triggered).

Node chain. To represent the mutable chain of nodes, we introduce the notion of *explicit chain* that allows decoupling the chain structure formed by the nodes and the content of the nodes. Concretely, the assertion xchain *dq* ℓs *dst* represents a chain linking locations ℓs and ending at value *dst*; *dq* is a discardable fraction [Vindum and Birkedal, 2021] that controls the ownership of the chain. In Figure 9.7, we use a variant of this assertion xtchain *hdr* ℓs *dst* that additionally requires ℓs to have header *hdr*.

This notion is very flexible as it is independent of the rest of the structure. As a matter of fact, we used it and its generalization to doubly linked list more broadly, to verify other algorithms. All the variants of Michael-Scott we verified rely on it. In particular, it was quite straightforward to extend the invariant of the bounded queue, where nodes carry more (mutable and immutable) information.

9.2.4 Future work

Hybrid queues. In the future, it would be interesting to build on this work to verify more complex hybrid queues (see Section 9.4), *i.e.* queues based on a list of (possibly circular) arrays.

Cooperative pointer reversal. Another generic way to implement a list-based queue is to rely on *cooperative pointer reversal*. In short, it consists in reversing the implementation of the Michael-Scott `push` operation: instead of first adding a new node to the end of the list and then updating the back pointer, the back pointer is updated first and then the node is added to the list — this last part may be performed cooperatively by another operation.

Vesa Karvonen proposed an MPMC queue[1] following this design in `Saturn`. Dmitry Vyukov also proposed an MPSC queue[2] based on the idea of reversing the `push` operation. It would be interesting to verify them.

9.3 Array-based queues

Array-based queues relies on an array of values, commonly operated as a ring buffer. As far as we know, three such queues have been verified in Iris: two bounded MPMC queues [Mével and Jourdan, 2021; Carbonneaux et al., 2022] and an unbounded MPMC queue [Vindum et al., 2022].

We verified two other array-based queues from `Saturn`: (1) a bounded SPSC queue, consisting of a circular array and two cached indices; (2) a relaxed MPMC queue (a bag) that can be seen as a simplified version of the queue verified by Vindum et al. [2022], where operations are similarly assigned a single-element queue but not ordered.

9.4 Towards hybrid queues: infinite-array-based queues

Some of the fastest queues proposed in the literature [Morrison and Afek, 2013; Yang and Mellor-Crummey, 2016; Ramalhete, 2016; Nikolaev, 2019; Romanov and Koval, 2023] are hybrid, *i.e.* employ a list of arrays. We implemented such a queue in OCaml.

When we tried to verify it, we encountered interesting problems that also occur in less realistic infinite-array-based queues. We claim that studying these idealized queues provides insights that are crucial for the (future) verification of the original hybrid queue. In this section, we present two verified infinite-array-based queues, leaving the extension to hybrid queues for future work.

9.4.1 First implementation: patient consumers

9.4.1.1 Specification

The specification of the first queue is given in Figure 9.8. If is similar to that of Figure 9.1, except `pop` always succeeds and it features an additional `size` operation.

9.4.1.2 Implementation

The implementation is extremely simple and can be seen as an idealized version of the queue verified by Vindum et al. [2022] — we realized this proximity after carrying out the verification. It relies on (1) two ticket dispensers, one for producers and one for consumers, incremented atomically using the fetch-and-add primitive, and (2) an infinite array (see Section 6.11) of slots. `push` t v takes a ticket from the producer dispenser and writes v into the corresponding cell of the infinite array. `pop` t takes a ticket from the consumer dispenser and waits until the corresponding producer has written its value.

[1] `https://github.com/ocaml-multicore/picos/pull/350`

[2] `https://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue`

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ \iota)$$

$$\textsc{inf-mpmc-queue-1-model-exclusive}\quad \frac{\mathsf{model}\ t\ vs_1 \qquad \mathsf{model}\ t\ vs_1}{\mathsf{False}}$$

$$\textsc{inf-mpmc-queue-1-create-spec}\quad \begin{array}{c} \mathsf{True} \\ \hline \mathtt{create}\ () \\ \hline t.\ \mathsf{inv}\ t\ \iota\ * \\ \quad \mathsf{model}\ t\ [] \end{array}$$

$$\textsc{inf-mpmc-queue-1-size-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ \iota \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \mathtt{size}\ t\ ⨾\ \iota \\ \hline \mathsf{model}\ t\ vs \\ \hdashline res.\ res = \mathsf{length}\ vs \end{array}$$

$$\textsc{inf-mpmc-queue-1-is-empty-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ \iota \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \mathtt{is\_empty}\ t\ ⨾\ \iota \\ \hline \mathsf{model}\ t\ vs \\ \hdashline res.\ res = \mathsf{decide}\ \ (vs = []) \end{array}$$

$$\textsc{inf-mpmc-queue-1-push-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ \iota \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \mathtt{push}\ t\ v\ ⨾\ \iota \\ \hline \mathsf{model}\ t\ (vs +\!\!+ [v]) \\ \hdashline ().\ \mathsf{True} \end{array}$$

$$\textsc{inf-mpmc-queue-1-pop-spec}\quad \begin{array}{c} \mathsf{inv}\ t\ \iota \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \mathtt{pop}\ t\ ⨾\ \iota \\ \hline v\ vs'.\ vs = v :: vs'\ * \\ \quad \mathsf{model}\ t\ vs' \\ \hdashline res.\ res = v \end{array}$$

Figure 9.8: `Inf_mpmc_queue_1`: Specification

External linearization. Given this implementation, the question is: when are `push` and `pop` linearized? The analysis is exactly the same as in Vindum et al. [2022].

A `push` operation is linearized at the point when it atomically takes a ticket. As a result, the logical content of the queue is updated before the pushed value is physically written in the array. Also, a producer may be linearized before another producer but write its value after.

The linearization point of `pop`, however, is non-fixed and may be external. If the consumer arrives after the corresponding producer, `pop` is linearized at the point when it atomically takes a ticket. If the consumer arrives before the producer, `pop` is linearized by the producer just after the linearization of the latter.

Future-depend linearization. The `size` successively reads the value of the producer dispenser, the consumer dispenser and the producer dispenser again; if the value has not changed, it returns the positive part of the difference; otherwise, it starts over. Interestingly, the linearization point is future-depend: `size` may or may not be linearized at the time it reads the consumer dispenser, depending on whether it later observes the same value for the producer dispenser. This pattern appears frequently in concurrent `size` operations.

9.4.1.3 Ghost state

Although the implementation is very short, it is quite challenging to verify. In particular, it involves non-trivial ghost state more or less similar to Vindum et al. [2022]. The external linearization point is handled using atomic updates (see Section 3.8) and the future-dependent linearization point using a local prophecy variable (see Chapter 5). We refer to the mechanization for details.

9.4.2 Second implementation: impatient consumers

9.4.2.1 Specification

The specification of the second queue is given in Figure 9.9. If is similar to that of Figure 9.8, except the specification of `size` and `is_empty` is slightly weaker.

9.4.2.2 Implementation

The implementation is also based on two ticket dispensers ordering the operations. However, consumers are now impatient: after taking a ticket, a consumer directly performs an atomic exchange, replacing the content of the corresponding slot with `Closed` and returning the former content; if the latter is `Value` v, `pop` returns v; otherwise, it starts over. Symmetrically, `push` t v takes a ticket and attempts to atomically update the content of the corresponding slot from `Empty` to `Value` v; if the update fails, meaning the consumer was quicker, the operation starts over.

Linearization. Although this may look like a benign optimization, it has dramatic consequences. To explain, let us ask the same question as before: when are `push` and `pop` linearized? Even if the producer arrives first, it is not certain to win the update and therefore cannot be linearized as before. Conversely, even if the consumer arrives last, it cannot be linearized right away since it might still win the update.

$$\text{persistent}\ (\text{inv}\ t\ \iota)$$

$$\textsc{inf-mpmc-queue-2-model-exclusive}\quad \frac{\text{model}\ t\ vs_1 \qquad \text{model}\ t\ vs_1}{\text{False}}$$

$$\textsc{inf-mpmc-queue-2-create-spec}\quad \frac{\frac{\text{True}}{\texttt{create}\ ()}}{t.\ \text{inv}\ t\ \iota * \text{model}\ t\ []}$$

$$\textsc{inf-mpmc-queue-2-size-spec}\quad \frac{\frac{\text{inv}\ t\ \iota}{vs.\ \text{model}\ t\ vs}}{\texttt{size}\ t\ ⨾\ \iota} \quad \frac{\text{model}\ t\ vs}{sz.\ \text{length}\ vs \leq sz}$$

$$\textsc{inf-mpmc-queue-2-is-empty-spec}\quad \frac{\frac{\text{inv}\ t\ \iota}{vs.\ \text{model}\ t\ vs}}{\texttt{is\_empty}\ t\ ⨾\ \iota} \quad \frac{\text{model}\ t\ vs}{b.\ \textbf{if}\ b\ \textbf{then}\ vs = []\ \textbf{else}\ \text{True}}$$

$$\textsc{inf-mpmc-queue-2-push-spec}\quad \frac{\frac{\text{inv}\ t\ \iota}{vs.\ \text{model}\ t\ vs}}{\texttt{push}\ t\ v\ ⨾\ \iota} \quad \frac{\text{model}\ t\ (vs +\!\!+ [v])}{().\ \text{True}}$$

$$\textsc{inf-mpmc-queue-2-pop-spec}\quad \frac{\frac{\text{inv}\ t\ \iota}{vs.\ \text{model}\ t\ vs}}{\texttt{pop}\ t\ ⨾\ \iota} \quad \frac{v\ vs'.\ vs = v :: vs' * \text{model}\ t\ vs'}{res.\ res = v}$$

Figure 9.9: `Inf_mpmc_queue_2`: Specification

$$\text{persistent } (\text{lstates-at } \gamma\ i\ \mathit{lstate}) \qquad \text{persistent } (\text{lstates-lb } \gamma\ i\ \mathit{lstate})$$

$$\frac{\begin{array}{c}\text{lstates-auth } \gamma\ \mathit{lstates} \\ \text{lstates-at } \gamma\ i\ \mathit{lstate}\end{array}}{\mathit{lstates}\,[i] = \text{Some } \mathit{lstate}}\ \textsc{lstates-at-lookup} \qquad \frac{\begin{array}{c}\mathit{lstates}\,[i] = \text{Some } \mathit{lstate} \\ \text{lstates-auth } \gamma\ \mathit{lstates}\end{array}}{\text{lstates-lb } \gamma\ i\ (\text{lstate-winner } \mathit{lstate})}\ \textsc{lstates-lb-get}$$

$$\frac{\begin{array}{c}\text{lstates-lb } \gamma\ i\ \mathit{lstate}_1 \\ \text{lstates-lb } \gamma\ i\ \mathit{lstate}_2\end{array}}{\text{lstate-winner } \mathit{lstate}_1 = \text{lstate-winner } \mathit{lstate}_2}\ \textsc{lstates-lb-agree}$$

$$\frac{\text{lstates-auth } \gamma\ \mathit{lstates}}{\begin{array}{l}\dot{\Rrightarrow}\ \text{lstates-auth } \gamma\ (\mathit{lstates} +\!\!+ [\mathit{lstate}]) * \\ \quad \text{lstates-lb } \gamma\ (\text{length } \mathit{lstates})\ (\text{lstate-winner } \mathit{lstate}) * \\ \quad \text{lstates-at } \gamma\ (\text{length } \mathit{lstates})\ \mathit{lstate}\end{array}}\ \textsc{lstates-update}$$

Figure 9.10: `Inf_mpmc_queue_2`: lstates theory

$$\text{persistent } (\text{producers-at } \gamma\ i\ \text{Discard})$$

$$\frac{\begin{array}{c}\text{producers-at } \gamma\ i\ \text{Own} \\ \text{producers-at } \gamma\ i\ \mathit{own}\end{array}}{\text{False}}\ \textsc{producers-at-exclusive} \qquad \frac{\text{producers-at } \gamma\ i\ \text{Own}}{\dot{\Rrightarrow}\ \text{producers-at } \gamma\ i\ \text{Discard}}\ \textsc{producers-at-discard} \qquad \frac{\text{producers-auth } \gamma\ i}{\begin{array}{l}\dot{\Rrightarrow}\ \text{producers-auth } \gamma\ (i+1) * \\ \quad \text{producers-at } \gamma\ i\ \text{Own}\end{array}}\ \textsc{producers-update}$$

Figure 9.11: `Inf_mpmc_queue_2`: producers theory

If the producer arrives first, either it (1) wins the update and is linearized when it takes a ticket, or (2) loses the update and starts over. If the producer arrives last, either it (1) wins the update and is linearized when it takes a ticket, linearizing the corresponding consumer at the same time, or (2) loses the update and starts over. The situation is symmetric for the consumer. Consequently, the linearization point of `push` is future-dependent and that of `pop` is both future-dependent and possibly external.

9.4.2.3 Ghost state

The definition of inv and model is given in Figure 9.13; we omit the definition of inv-lstate-left, inv-lstate-right and inv-slot. It relies on six ghost theories: model, history, lstates, producers and consumers.

Prophecy variable. To deal with the future-dependent linearization points, we use a shared multiplexed prophecy variable (see Section 5.4) stored into the queue. This prophecy variable predicts the per-index winner of the slot update. To distinguish operations, we use the same trick as Jung et al. [2020], *i.e.* physical identifiers.

$$\text{persistent}\ (\text{consumers-at}\ \gamma\ i\ \text{Discard}) \qquad \text{persistent}\ (\text{consumers-lb}\ \gamma\ i)$$

$$\frac{\text{consumers-at}\ \gamma\ i\ \text{Own} \quad \text{consumers-at}\ \gamma\ i\ \mathit{own}}{\text{False}}\ \textsc{consumers-at-exclusive}$$

$$\frac{\text{consumers-at}\ \gamma\ i\ \text{Own}}{\dot{\Rrightarrow}\ \text{consumers-at}\ \gamma\ i\ \text{Discard}}\ \textsc{consumers-at-discard}$$

$$\frac{\text{consumers-auth}\ \gamma\ i \quad \text{consumers-lb}\ \gamma\ j}{j \leq i}\ \textsc{consumers-lb-valid}$$

$$\frac{\text{consumers-auth}\ \gamma\ i}{\text{consumers-lb}\ \gamma\ i}\ \textsc{consumers-lb-get}$$

$$\frac{\text{consumers-auth}\ \gamma\ i}{\dot{\Rrightarrow}\ \text{consumers-auth}\ \gamma\ (i+1) * \text{consumers-at}\ \gamma\ i\ \text{Own}}\ \textsc{consumers-update}$$

Figure 9.12: `Inf_mpmc_queue_2`: consumers theory

$$
\begin{array}{l}
\text{inv-inner}\ \ell\ \gamma \triangleq \\
\quad \exists\ \mathit{front}\ \mathit{back}\ \mathit{hist}\ \mathit{slots}\ \mathit{lstates}\ \mathit{pasts}\ \mathit{prophss}. \\
\quad \ell.\text{front} \mapsto \mathit{front} * \\
\quad \ell.\text{back} \mapsto \mathit{back} * \\
\quad \text{inf-array.model}\ \gamma.\text{data}\ \mathit{slots} * \\
\quad \text{model}_2\ \gamma\ (\text{oflatten}\ (\text{drop}\ \mathit{front}\ \mathit{hist})) * \\
\quad \text{history-auth}\ \gamma\ \mathit{hist} * \\
\quad \text{length}\ \mathit{hist} = \mathit{back} * \\
\quad \text{lstates-auth}\ \gamma\ \mathit{lstates} * \\
\quad \text{length}\ \mathit{lstates} = \text{max}\ \mathit{front}\ \mathit{back} * \\
\quad \text{wise-prophets.model}\ \gamma.\text{proph}\ \gamma.\text{proph-name}\ \mathit{pasts}\ \mathit{prophss} * \\
\quad \text{producers-auth}\ \gamma\ \mathit{back} * \\
\quad \text{consumers-auth}\ \gamma\ \mathit{front} * \\
\quad \left(\mathop{\Huge *}_{i \mapsto \mathit{lstate} \in \text{take}\ \mathit{back}\ \mathit{lstates}} \text{inv-lstate-left}\ \gamma\ \mathit{back}\ i\ \mathit{lstate}\right) * \\
\quad \left(\mathop{\Huge *}_{k \mapsto \mathit{lstate} \in \text{drop}\ \mathit{back}\ \mathit{lstates}} \text{inv-lstate-right}\ \gamma\ (\mathit{back}+k)\ \mathit{lstate}\right) * \\
\quad (\forall\ i.\ \text{inv-slot}\ \gamma\ i\ (\mathit{slots}\ i)\ (\mathit{pasts}\ i))
\end{array}
\qquad
\begin{array}{l}
\text{model}\ t\ \mathit{vs} \triangleq \\
\quad \exists\ \ell\ \gamma. \\
\quad t = \ell * \\
\quad \text{meta}\ \ell\ \top\ \gamma * \\
\quad \text{model}_1\ \gamma\ \mathit{vs}
\end{array}
$$

$$
\begin{array}{l}
\text{inv}\ t\ \iota \triangleq \\
\quad \exists\ \ell\ \gamma. \\
\quad t = \ell * \\
\quad \iota = \gamma.\text{inv} * \\
\quad \text{meta}\ \ell\ \top\ \gamma * \\
\quad \ell.\text{data} \mapsto_{\square} \gamma.\text{data} * \\
\quad \ell.\text{proph} \mapsto_{\square} \gamma.\text{proph} * \\
\quad \text{inf-array.inv}\ \gamma.\text{data} * \\
\quad \boxed{\text{inv-inner}\ \ell\ \gamma}^{\gamma.\text{inv}}
\end{array}
$$

Figure 9.13: `Inf_mpmc_queue_2`: Predicates definition (excerpt)

To carry out the proof, we must resolve the prophecy variable atomically while updating the infinite array. For doing so, we crucially rely on the special update operations performing prophecy resolution internally, presented in Section 6.11.

model theory. This theory is similar to Figure 9.5. It connects inv and model.

history theory. This theory is similar to Figure 9.3. It keeps track of the pushed values.

lstates theory (Figure 9.10). This theory is the most important one. It is responsible for keeping track of the logical state of each slot: Producer indicates that the producer wins the update, Consumer indicates that the consumer wins the update, ProducerProducer indicates that the producer arrived first and wins the update, ProducerConsumer indicates that the producer arrived first and loses the update, and symmetrically for ConsumerConsumer and ConsumerProducer.

The assertion lstates-auth γ *lstates* represents the ownership of the logical states. The persistent assertion lstates-at γ i *lstate* represents the knowledge that the logical state of the i-th slot is *lstate* (LSTATES-AT-LOOKUP). The persistent assertion lstates-lb γ i *lstate* represents the knowledge that *lstate* is a lower bound on the logical state of the i-th slot; in practice, *lstate* is either Producer or Consumer, thereby indicating the winner.

When the first operation arrives, it predicts the winner and sets the logical state accordingly, thereby imposing its prediction to the other. One may wonder why the logical state contain so much information. Empirically, our attempts showed that this is needed to carry out the proofs, especially to deal with corner cases; it seems that decoupling the winner information from the rest through separate ghost state is not possible.

producers theory (Figure 9.11). This theory is responsible for the emission of producer tokens (PRODUCERS-UPDATE), which are initially exclusive (PRODUCERS-AT-EXCLUSIVE) but can be made persistent (PRODUCERS-AT-DISCARD).

consumers theory (Figure 9.12). Similarly, this theory is responsible for the emission of consumer tokens (CONSUMERS-UPDATE).

9.5 Stack-based queues

A standard way to implement a sequential queue is to use two stacks: producers push onto the *back stack* while consumers pop from the *front stack*, stealing and reversing the back stack when needed. Based on this simple idea, Vesa Karvonen developed a new lock-free concurrent queue. We verified three variants: an MPMC queue from `Picos`, a basic MPSC queue from `Saturn` and a closable MPSC queue from `Eio`.

Generative constructors. Similarly to the sequential implementation, the two stacks are mainly immutable. Both stacks are updated using compare-and-set, so we use generative constructors to reason about physical equality.

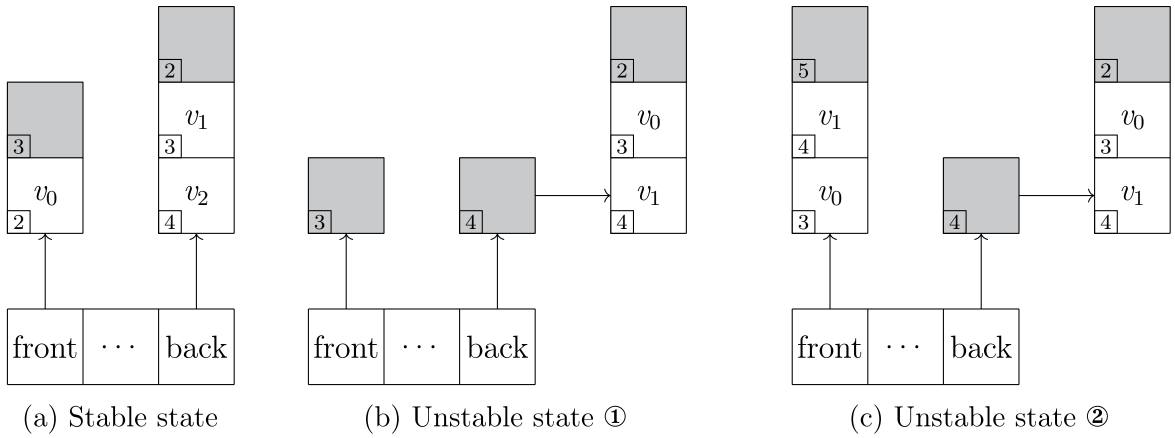


Figure 9.14: `Mpmc_queue_2`: States

Concurrent stack reversal. Similarly again, producers and consumers work concurrently on separate stacks, limiting interference. The key difference compared to the sequential version is that the algorithm has to deal with the concurrent back stack reversal in a lock-free way.

Figure 9.14 shows the three states involved in the reversal. Essentially, the concurrent protocol — and therefore the Iris invariant — includes a *destabilization* phase during which a new back stack pointing to the old one awaits to be *stabilized* (Figure 9.14b), which happens when the reversed old back stack becomes the new front stack (Figure 9.14c). To fully stabilize the structure, the link from the new back stack to the old is removed (Figure 9.14a). In practice, the synchronization is fairly tricky and relies on the indices of the elements.

9.6 Relaxed queue

We implemented and verified a relaxed queue 🐫 🐓 guaranteeing only per-producer FIFO ordering. It is based on an industrial-strength C++ queue [Desrochers, 2025]. What makes it interesting is its original interface, which differs from other queues we worked on.

9.6.1 Specification

The specification is given in Figure 9.15. It features four predicates: inv, model, producer and consumer.

The persistent assertion inv t ι represents the knowledge that t is a valid queue. It is returned by `create` (BAG-2-CREATE-SPEC) and require by all operations.

The exclusive assertion model t $zooValss$ represents the ownership of queue t and the knowledge that is contains the per-producer values vss. It is also returned by `create` (BAG-2-CREATE-SPEC) and accessed atomically by most operations.

The exclusive assertion producer t $prod$ ws represents the ownership of producer $prod$ attached to queue t; ws is an upper bound on the values of the sub-queue corresponding to $prod$ (BAG-2-PRODUCER-VALID). `create_producer` t creates a new producer for t (BAG-2-CREATE-PRODUCER-SPEC). `push` $prod$ v atomically pushes v into the sub-queue of producer $prod$ (BAG-2-PUSH-SPEC). `close_producer` $prod$ marks producer $prod$ as

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ \iota)$$

BAG-2-MODEL-EXCLUSIVE

$$\frac{\mathsf{model}\ t\ vss_1 \quad \mathsf{model}\ t\ vss_2}{\mathsf{False}}$$

BAG-2-PRODUCER-VALID

$$\frac{\iota \subseteq \mathcal{E} \quad \mathsf{inv}\ t\ \iota \quad \mathsf{model}\ t\ vss \quad \mathsf{producer}\ t\ prod\ ws}{\Rrightarrow_{\mathcal{E}} \exists\ vs.\ vss\,[prod] = \mathsf{Some}\ vs \ast \mathsf{suffix}\ vs\ ws}$$

BAG-2-PRODUCER-EXCLUSIVE

$$\frac{\mathsf{producer}\ t_1\ prod\ ws_1 \quad \mathsf{producer}\ t_2\ prod\ ws_2}{\mathsf{False}}$$

BAG-2-CONSUMER-EXCLUSIVE

$$\frac{\mathsf{consumer}\ t_1\ cons \quad \mathsf{consumer}\ t_2\ cons}{\mathsf{False}}$$

BAG-2-CREATE-SPEC

$$\frac{\mathsf{True}}{\texttt{create}\ ()} \quad t.\ \mathsf{inv}\ t\ \iota \ast \mathsf{model}\ t\ \emptyset$$

BAG-2-CREATE-PRODUCER-SPEC

$$\mathsf{inv}\ t\ \iota$$
$$vss.\ \mathsf{model}\ t\ vss$$
$$\texttt{create\_producer}\ t\ ;\ \iota$$
$$prod.\ \mathsf{model}\ t\ (vss\,[prod \mapsto [\,]])$$
$$res.\ res = prod \ast \mathsf{producer}\ t\ prod\ [\,]$$

BAG-2-PUSH-SPEC

$$\mathsf{inv}\ t\ \iota \ast \mathsf{producer}\ t\ prod\ ws$$
$$vss.\ \mathsf{model}\ t\ vss$$
$$\texttt{push}\ prod\ v\ ;\ \iota$$
$$vs.\ vs\,[prod] = \mathsf{Some}\ vs \ast \mathsf{model}\ t\ (vss\,[prod \mapsto vs +\!\!+ [v]])$$
$$().\ \mathsf{producer}\ t\ prod\ (vs +\!\!+ [v])$$

BAG-2-CLOSE-PRODUCER-SPEC

$$\mathsf{inv}\ t\ \iota \ast \mathsf{producer}\ t\ prod\ ws$$
$$\texttt{close\_producer}\ prod$$
$$().\ \mathsf{producer}\ t\ prod\ ws$$

BAG-2-POP-SPEC

$$\mathsf{inv}\ t\ \iota \ast \mathsf{consumer}\ t\ cons$$
$$vss.\ \mathsf{model}\ t\ vss$$
$$\texttt{pop}\ t\ cons\ ;\ \iota$$
$$o.\ \textbf{match}\ o\ \textbf{with} \mid \mathsf{None} \Rightarrow \mathsf{model}\ t\ vss \mid \mathsf{Some}\ v \Rightarrow \exists\ prod\ vs.\ vss\,[prod] = \mathsf{Some}\ (v :: vs) \ast \mathsf{model}\ t\ (vss\,[prod \mapsto vs])\ \textbf{end}$$
$$res.\ res = o \ast \mathsf{consumer}\ t\ cons$$

BAG-2-CREATE-CONSUMER-SPEC

$$\frac{\mathsf{inv}\ t\ \iota}{\texttt{create\_consumer}\ t} \quad cons.\ \mathsf{consumer}\ t\ cons$$

Figure 9.15: Bag_2: Specification

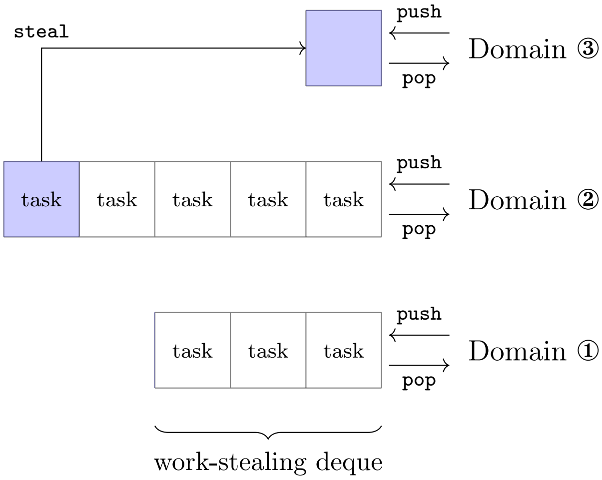


Figure 9.16: Work stealing

closed (BAG-2-CLOSE-PRODUCER-SPEC), which does not affect the current consumers but makes the corresponding sub-queue unavailable for future consumers.

The exclusive assertion consumer t *cons* represents the ownership of consumer *cons* attached to queue t. `create_consumer` t creates a new consumer (BAG-2-CREATE-CONSUMER-SPEC). `pop` t *cons* pops a value from t through consumer *cons*.

9.6.2 Implementation

Producers. A queue consists of a lock-free linked list of SPMC sub-queues (see Section 9.2), each corresponding to a single producer. A producer stores both its sub-queue and the linked list node where the sub-queue can be found. `push` simply calls the same operation on the sub-queue. `close_producer` removes the sub-queue from the linked list, preventing future consumers from accessing it.

Consumers. A consumer consists of an optional default sub-queue. The first time it is used through `pop`, it traverses the linked list looking for a non-empty sub-queue. If it finds one, the consumer records it as a default target for future `pop` operations; otherwise, the operation fails.

9.7 Work-stealing deques

Work-stealing. Randomized *work stealing* [Blumofe and Leiserson, 1999] is the standard strategy for parallel task scheduling. It has been implemented in many libraries, including Cilk [Blumofe et al., 1996; Frigo et al., 1998], TBB, OpenMP, `Taskflow` [Huang et al., 2022], `Tokio` and `Domainslib` [Multicore OCaml development team, 2025].

The idea of work-stealing, illustrated in Figure 9.16, is the following. Each domain owns a deque-like data structure, called *work-stealing deque*, to store its tasks. Locally, each domain treats its deque as a stack, operating at the back end. When a domain runs out of tasks, it becomes a thief: it tries to steal a task from the deque of another randomly selected “victim” domain, operating at the front end. Multiple thieves may concurrently attempt to steal tasks from a single deque.

Work-stealing deque. The most popular work-stealing deque algorithm is the Chase-Lev deque [Chase and Lev, 2005; Lê et al., 2013]; it is lock-free and unbounded. We verified the implementation from `Saturn` along with two other variants: a bounded variant, used in the `Moonpool` [Cruanes, 2025] and `Taskflow` [Huang et al., 2022] libraries, and an idealized infinite-array-based variant.

Remarkably, the three variants essentially share the same logical states. In particular, although they do not behave exactly the same way, the original and the idealized versions follow a similar concurrent protocol, involving external and future-dependent linearization.

9.7.1 Infinite work-stealing deque

9.7.1.1 Specification

The specification of the infinite-array-based version is given in Figure 9.17. It features three predicates: inv, model and owner.

The persistent assertion inv t ι represents the knowledge that t is a valid deque. It is returned by `create` (INF-WS-DEQUE-CREATE-SPEC) and required by all operations.

The exclusive assertion model t vs represents the ownership of the content of the deque vs. It it returned by `create` and accessed atomically by all operations.

The exclusive assertion owner t ws represents the owner of the deque; ws is an upper bound on the current content of the deque (INF-WS-DEQUE-OWNER-MODEL). It is returned by `create` and used by all private operation: `size` (INF-WS-DEQUE-SIZE-SPEC), `is_empty` (INF-WS-DEQUE-IS-EMPTY-SPEC), `push` (INF-WS-DEQUE-PUSH-SPEC) and `pop` (INF-WS-DEQUE-POP-SPEC). The only public operation is `steal` (INF-WS-DEQUE-STEAL-SPEC), which does not require owner.

Note that the public postconditions of the private operations are quite verbose. This is due to the fact that owner is passed to the operation and therefore cannot be combined with model through INF-WS-DEQUE-OWNER-MODEL to get information about the content of the deque; instead, we provide such information in the public postcondition. We need this expressivity in practice to verify a wrapper with better liveness properties.

9.7.1.2 Weak specification.

In parallel with this thesis, Choi [2023] also worked on the verification of the Chase-Lev work-stealing deque. However, we argue that the specification he proves, given in Figure 9.18, is unsatisfactory. Indeed, contrary to our specification, WS-DEQUE-STEAL-SPEC-WEAK and WS-DEQUE-POP-SPEC-WEAK say nothing about the observed content of the deque when the operation fails.

In practice, these weaker specifications, especially that of `pop`, are not sufficient to reason about the *termination* of a work-stealing scheduler. In Chapter 10, we show how our strong specifications are lifted all the way up to the scheduler.

Another point we would like to make is that weakening the specification does make the verification simpler, but one may argue that the most subtle and interesting part of it is lost.

9.7.1.3 Implementation

The implementation relies on (1) an infinite array (see Section 6.11), (2) a *monotonic* front index for the thieves, and (3) a back index reserved to the owner of the deque.

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ \iota)$$

$$\textsc{inf-ws-deque-model-exclusive}\quad \frac{\mathsf{model}\ t\ vs_1 \qquad \mathsf{model}\ t\ vs_2}{\mathsf{False}}$$

$$\textsc{inf-ws-deque-owner-exclusive}\quad \frac{\mathsf{owner}\ t\ ws_1 \qquad \mathsf{owner}\ t\ ws_2}{\mathsf{False}}$$

$$\textsc{inf-ws-deque-owner-model}\quad \frac{\mathsf{owner}\ t\ ws \qquad \mathsf{model}\ t\ vs}{\mathsf{suffix}\ vs\ ws}$$

INF-WS-DEQUE-CREATE-SPEC

$$\frac{\dfrac{\mathsf{True}}{\texttt{create ()}}}{t.\ \mathsf{inv}\ t\ \iota * \mathsf{model}\ t\ [] * \mathsf{owner}\ t\ []}$$

INF-WS-DEQUE-SIZE-SPEC

$$\begin{array}{c} \mathsf{inv}\ t\ \iota * \mathsf{owner}\ t\ ws \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \texttt{size}\ t\ ;\ \iota \\ \hline \mathsf{suffix}\ vs\ ws * \mathsf{model}\ t\ vs \\ \hdashline res.\ res = \mathsf{length}\ vs * \mathsf{owner}\ t\ vs \end{array}$$

INF-WS-DEQUE-IS-EMPTY-SPEC

$$\begin{array}{c} \mathsf{inv}\ t\ \iota * \mathsf{owner}\ t\ ws \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \texttt{is\_empty}\ t\ ;\ \iota \\ \hline \mathsf{suffix}\ vs\ ws * \mathsf{model}\ t\ vs \\ \hdashline res.\ res = \mathsf{decide}\ (vs = []) * \mathsf{owner}\ t\ vs \end{array}$$

INF-WS-DEQUE-PUSH-SPEC

$$\begin{array}{c} \mathsf{inv}\ t\ \iota * \mathsf{owner}\ t\ ws \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \texttt{push}\ t\ v\ ;\ \iota \\ \hline \mathsf{suffix}\ vs\ ws * \mathsf{model}\ t\ (vs +\!\!+ [v]) \\ \hdashline ().\ \mathsf{owner}\ t\ (vs +\!\!+ [v]) \end{array}$$

INF-WS-DEQUE-POP-SPEC

$$\begin{array}{c} \mathsf{inv}\ t\ \iota * \mathsf{owner}\ t\ ws \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \texttt{pop}\ t\ ;\ \iota \\ \hline \begin{array}{l} o\ ws'.\ \mathsf{suffix}\ vs\ ws * \\ \quad \textbf{match}\ o\ \textbf{with} \\ \quad \mid \mathsf{None} \Rightarrow \\ \qquad vs = [] * ws' = [] * \\ \qquad \mathsf{model}\ t\ [] \\ \quad \mid \mathsf{Some}\ v \Rightarrow \\ \qquad \exists\ vs'. \\ \qquad vs = vs' +\!\!+ [v] * ws' = vs' * \\ \qquad \mathsf{model}\ t\ vs' \\ \quad \textbf{end} \end{array} \\ \hdashline res.\ res = o * \mathsf{owner}\ t\ ws' \end{array}$$

INF-WS-DEQUE-STEAL-SPEC

$$\begin{array}{c} \mathsf{inv}\ t\ \iota \\ \hdashline vs.\ \mathsf{model}\ t\ vs \\ \hline \texttt{steal}\ t\ ;\ \iota \\ \hline \mathsf{model}\ t\ (\mathsf{tail}\ vs) \\ \hdashline res.\ res = \mathsf{head}\ vs \end{array}$$

Figure 9.17: `Inf_ws_deque`: Specification

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ \iota)$$

WS-DEQUE-CREATE-SPEC-WEAK

$$\frac{\mathsf{True}}{\texttt{create}\ ()}$$
$t.\ \mathsf{inv}\ t\ \iota\ *$
$\quad\mathsf{model}\ t\ []\ *$
$\quad\mathsf{owner}\ t$

WS-DEQUE-PUSH-SPEC-WEAK

$\mathsf{inv}\ t\ \iota\ *$
$\mathsf{owner}\ t$
- - - - - - - - - - - -
$vs.\ \mathsf{model}\ t\ vs$
$$\frac{}{\texttt{push}\ t\ v\ ⨾\ \iota}$$
$\mathsf{model}\ t\ (vs + [v])$
- - - - - - - - - - - -
$().\ \mathsf{owner}\ t$

WS-DEQUE-STEAL-SPEC-WEAK

$\mathsf{inv}\ t\ \iota$
- - - - - - - - - - - -
$vs.\ \mathsf{model}\ t\ vs$
$$\frac{}{\texttt{steal}\ t\ ⨾\ \iota}$$
$o.$ **match** o **with**
$\quad|\ \mathsf{None} \Rightarrow$
$\qquad\mathsf{model}\ t\ vs$
$\quad|\ \mathsf{Some}\ v \Rightarrow$
$\qquad\exists\ vs'.$
$\qquad vs = v :: vs'\ *$
$\qquad\mathsf{model}\ t\ vs'$
end
- - - - - - - - - - - -
$res.\ res = o$

WS-DEQUE-POP-SPEC-WEAK

$\mathsf{inv}\ t\ \iota\ *$
$\mathsf{owner}\ t$
- - - - - - - - - - - -
$vs.\ \mathsf{model}\ t\ vs$
$$\frac{}{\texttt{pop}\ t\ ⨾\ \iota}$$
$o.$ **match** o **with**
$\quad|\ \mathsf{None} \Rightarrow$
$\qquad\mathsf{model}\ t\ vs$
$\quad|\ \mathsf{Some}\ v \Rightarrow$
$\qquad\exists\ vs'.$
$\qquad vs = vs' + [v]\ *$
$\qquad\mathsf{model}\ t\ vs'$
end
- - - - - - - - - - - -
$res.\ res = o\ *$
$\mathsf{owner}\ t$

Figure 9.18: `Ws_deque`: Weak specification

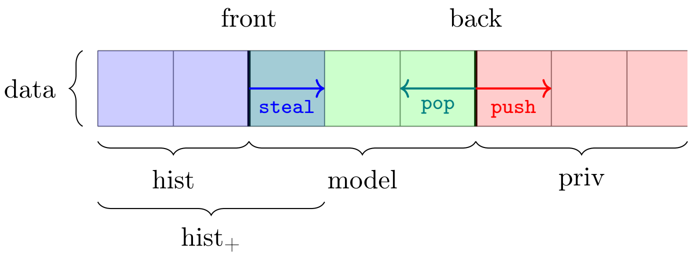


Figure 9.19: `Inf_ws_deque`: Physical state

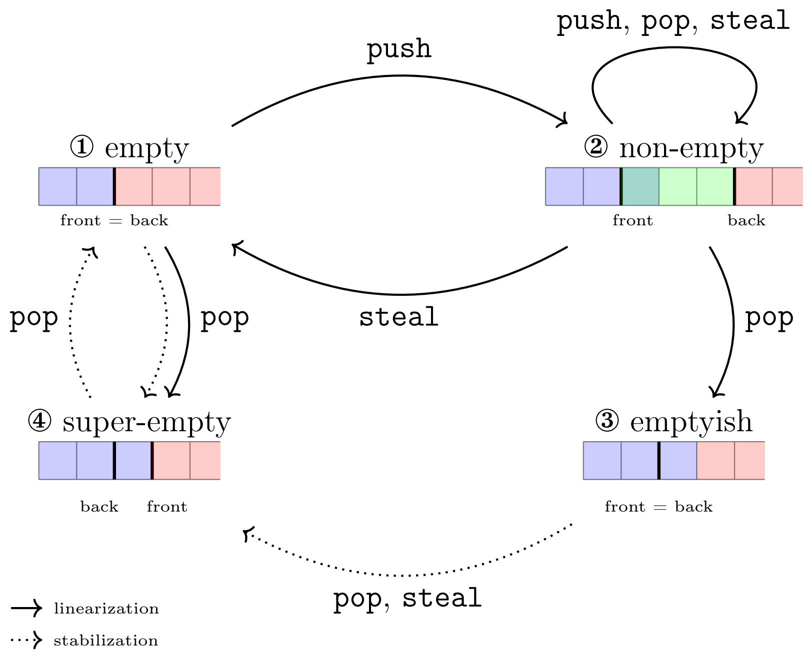


Figure 9.20: `Inf_ws_deque`: Logical state

In general, we can divide the infinite array as in Figure 9.19. The first part, between 0 and the front index, corresponds to the *persistent* history of stolen values. The second part, between the two indices, corresponds to the logical content of the deque, as represented by model. The last part, beyond the back index, corresponds to the private section of the array, reserved to the owner.

Given this representation, the algorithm proceeds as follows. `push` t v writes v into the first private cell and atomically increments the back index, thereby publishing the value. Symmetrically, `pop` t atomically decrements the back index and returns the value of the cell it just privatized. `steal` t is much more careful: (1) it reads the front and the back indices; (2) if the deque looks empty, it fails; (3) otherwise, it attempts to advance the front index; (4) if the update succeeds, the value at the front index is returned; (5) otherwise, it starts over.

The above description overlooked one crucial aspect: what happens at the limit, when `pop` and `steal` compete for the last value in the deque? In that case, the deque must be *stabilized*: `pop` also attempts to advance the front index before incrementing the back index — whether it wins the update or not — thereby equalizing the two indices.

9.7.1.4 Logical states

Figure 9.20 tells the same story as above in terms of four *logical states*: (1) in the stable "empty" state, the deque is indeed empty, as indicated by the two equal indices; (2) in the stable "non-empty" state, the model is non-empty, meaning thieves may compete for the first value; (3) in the unstable "emptyish" state, the thieves and the owner compete for the same value; (4) in the unstable "super-empty" state, some operation won the value and the deque is waiting to be stabilized by the owner.

Let us now focus on the "emptyish" state. In this physical configuration, it makes sense to say that the model of the deque should be empty. In fact, is has to be empty: if a `steal` operation observed this state, it would conclude that the deque is empty — except under a weak specification. But then, if the model should be empty, which operation was linearized during the transition to the "emptyish" state? We have no choice: it should be the winner of the front update, *i.e.* the operation which triggers the transition to the "super-empty" state. In conclusion, we have to predict the winner using a (multiplexed) prophecy variable (see Chapter 5).

9.7.2 Bounded work-stealing deque

In the bounded variant, the infinite array is replaced with a finite circular array. As a consequence, the convenient infinite representation goes away and tedious reasoning about circular array slices is required. However, the logical states and transitions as well as the prophecy mechanism are essentially the same.

It is an open question whether we could factorize part of the verification through a well-chosen abstraction that could be instantiated both with infinite and circular arrays. One certainty is that this is not possible without slightly altering the implementation of the infinite variant: in `steal`, the front cell is read after performing the update in the infinite variant, which would be incorrect in the finite variant since the owner is allowed to overwrite the value.

9.7.3 Dynamic work-stealing deque

In the original algorithm, the owner may dynamically resize the circular array. More precisely, it can change the array at will provided that the public part (between the two indices) is preserved. Thus, while only one array is stored in the deque, there can be many different circular arrays alive at the same time, *i.e.* accessible by thieves.

While the invariant of Choi [2023] requires additional ghost state to keep track of the arrays and maintain their compatibility, the precision of our notion of logical state allows to only maintain compatibility between the current array and the array read by the next winner (if any).

9.8 Future work

Relaxed memory model. As mentioned in Section 4.4, the main shortcoming of ZooLang is its sequentially consistent memory model. This is of particular concern in the algorithms we verified in this chapter, which use shared non-atomic variables for efficiency but should also guarantee synchronization. Thus, it would be interesting to apply the methodology of Mével and Jourdan [2021] to adapt our invariants to relaxed memory.

Other data structures. Two important data structures from `Saturn` remain unverified as of today: a hash table and a skiplist. We already started working on the hash table on paper; although some parts are technical, it should be feasible to finish and mechanize the proof. The skiplist, however, seems more challenging; recent work [Carrott, 2022; Patel et al., 2024; Park et al., 2025] suggests new avenues that are worth exploring.

Chapter 10

Parabs: A library of parallel abstractions

The culminating point of our work is the verified Parabs library, offering parallel abstractions atop a task scheduler. While it was originally based on Domainslib [Multicore OCaml development team, 2025] (see Section 2.4), it evolved as a more ambitious project aimed at unifying various existing paradigms and scheduling strategies. It was designed with a focus on *flexibility*, letting users choose the scheduling strategy and build their own scheduler. One of the motivations of this design is to provide a framework to easily develop and experiment parallel infrastructures in OCaml 5.

10.1 Overview

Figure 10.1 gives an overview of Parabs; solid edges represent module dependencies while dashed edges represent interface implementations. Essentially, the library is made of four abstraction levels built on top of each other: `Ws_deques`, `Ws_hub`, `Pool` and `Vertex`.

The `Pool` module provides a task scheduler; internally, it maintains a pool of domains. Its design is inspired by Domainslib, Taskflow [Huang et al., 2022] and Moonpool [Cruanes, 2025]. As of today, it supports three scheduling strategies: (1) standard randomized work-stealing [Blumofe and Leiserson, 1999] with public deques (as presented in Section 9.7), (2) randomized work-stealing with private deques [Acar et al., 2013], (3) a simple "first-in first-out" strategy with one shared queue. In addition, it should be possible to implement other scheduling strategies (see Section 10.10), *e.g.* work sharing.

On top of `Pool`, the `Vertex` module provides a *task graph* abstraction. More precisely, it is an implementation of *DAG-calculus* [Acar et al., 2016] — we present it in Section 10.6.

Remarkably, the three upper levels implemented on top of `Ws_deques` should be OCaml functors. Unfortunately, ZooLang does not currently support functors; therefore, only one branch of the tree of Figure 10.1 is active at a time.

10.2 Work-stealing deques

At the first level, `Ws_deques` provides a generic interface for a set of work-stealing deques, abstracting over the underlying scheduling strategy. We describe its specification (Section 10.2.1) and two current realizations: `Ws_deques_public` (Section 10.2.2) and `Ws_deques_private` (Section 10.2.3).

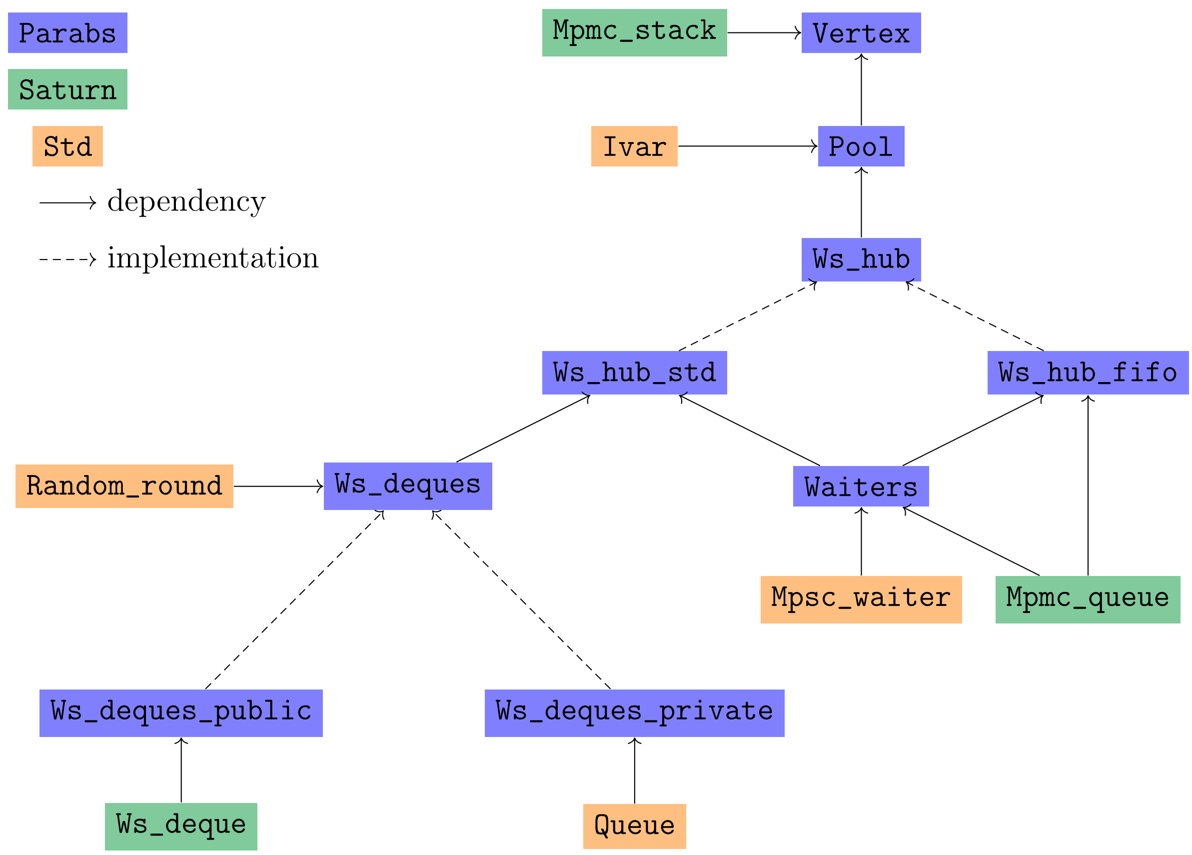


Figure 10.1: Overview of the `Parabs` library

10.2.1 Specification

The specification of `Ws_deques` is given in Figure 10.2. Similarly to Section 9.7, it features three predicates: inv, model and owner.

The persistent assertion inv t γ sz represents the knowledge that t is a set of work-stealing deques; γ is a user-provided invariant name and sz is the number of deques. It is returned by `create` (WS-DEQUES-CREATE-SPEC) and required by all operations.

The assertion model t vss represents the possession of t and the knowledge that it currently contains values vss (list of lists of values, one per deque). It is returned by `create` (WS-DEQUES-CREATE-SPEC) and modified atomically by `push` (WS-DEQUES-PUSH-SPEC), `pop` (WS-DEQUES-POP-SPEC), `steal_to` (WS-DEQUES-STEAL_TO-SPEC) and `steal_as` (WS-DEQUES-STEAL-AS-SPEC).

The exclusive assertion owner t i $status$ ws is the owner token of the i-th deque of t. It is also returned by `create` (WS-DEQUES-CREATE-SPEC). Similarly to Section 9.7, it grants its possessor (the owner of the i-th deque) the right to access the owner's end of the i-th deque, *i.e.* the right to call `push` (WS-DEQUES-PUSH-SPEC) and `pop` (WS-DEQUES-POP-SPEC); more generally, it is used by owner-only operations. The values ws represent an upper bound on the actual values of the deque (WS-DEQUES-MODEL-OWNER); in particular, when ws is empty, the deque must be empty. $status$ is the current status of the deque, *i.e.* either Blocked or Nonblocked. Indeed, blocking a deque through `block` (WS-DEQUES-BLOCK-SPEC) is necessary for its owner to call `steal_to` (WS-DEQUES-STEAL-TO-SPEC) and `steal_as` (WS-DEQUES-STEAL-AS-SPEC); conversely, a deque can be unblocked using `unblock` (WS-DEQUES-UNBLOCK-SPEC).

Most specifications are straightforward; atomic specifications update model in the

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ \iota\ sz)$$

WS-DEQUES-INV-AGREE
$$\frac{\mathsf{inv}\ t\ \iota_1\ sz_1 \qquad \mathsf{inv}\ t\ \iota_2\ sz_2}{sz_1 = sz_2}$$

WS-DEQUES-OWNER-EXCLUSIVE
$$\frac{\mathsf{owner}\ t\ i\ status_1\ ws_1 \qquad \mathsf{owner}\ t\ i\ status_2\ ws_2}{\mathsf{False}}$$

WS-DEQUES-INV-MODEL
$$\frac{\mathsf{inv}\ t\ \iota\ sz \qquad \mathsf{model}\ t\ vss}{\mathsf{length}\ vss = sz}$$

WS-DEQUES-INV-OWNER
$$\frac{\mathsf{inv}\ t\ \iota\ sz \qquad \mathsf{owner}\ t\ i\ status\ ws}{i < sz}$$

WS-DEQUES-MODEL-OWNER
$$\frac{\mathsf{model}\ t\ vss \qquad \mathsf{owner}\ t\ i\ status\ ws}{\exists\ vs.\ vss\,[i] = \mathsf{Some}\ vs * \mathsf{suffix}\ vs\ ws}$$

WS-DEQUES-CREATE-SPEC
$$\begin{array}{c} 0 \leq sz \\ \hline \texttt{create}\ sz \\ \hline t.\ \mathsf{inv}\ t\ \iota\ sz * \mathsf{model}\ t\ (\mathsf{replicate}\ sz\ []) * \mathop{\Huge\ast}_{i \in [\![0;sz)} \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ [] \end{array}$$

WS-DEQUES-SIZE-SPEC
$$\begin{array}{c} \mathsf{inv}\ t\ \iota\ sz \\ \hline \texttt{size}\ t \\ \hline res.\ res = sz \end{array}$$

WS-DEQUES-BLOCK-SPEC
$$\begin{array}{c} \mathsf{inv}\ t\ \iota\ sz * \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ ws \\ \hline \texttt{block}\ t\ i \\ \hline ().\ \mathsf{owner}\ t\ i\ \mathsf{Blocked}\ ws \end{array}$$

WS-DEQUES-UNBLOCK-SPEC
$$\begin{array}{c} \mathsf{inv}\ t\ \iota\ sz * \mathsf{owner}\ t\ i\ \mathsf{Blocked}\ ws \\ \hline \texttt{unblock}\ t\ i \\ \hline ().\ \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ ws \end{array}$$

WS-DEQUES-PUSH-SPEC
$$\begin{array}{c} \mathsf{inv}\ t\ \iota\ sz * \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ ws \\ \hdashline vss.\ \mathsf{model}\ t\ vss \\ \hline \texttt{push}\ t\ i\ v\ ⨾\ \iota \\ \hline vs.\ vss\,[i] = \mathsf{Some}\ vs * \mathsf{suffix}\ vs\ ws * \mathsf{model}\ t\ (vss\,[i \mapsto vs +\!\!+ [v]]) \\ \hdashline ().\ \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ (vs +\!\!+ [v]) \end{array}$$

WS-DEQUES-POP-SPEC
$$\begin{array}{c} \mathsf{inv}\ t\ \iota\ sz * \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ ws \\ \hdashline vss.\ \mathsf{model}\ t\ vss \\ \hline \texttt{pop}\ t\ i\ ⨾\ \iota \\ \hline \begin{array}{l} o\ ws'.\ \textbf{match}\ o\ \textbf{with} \\ \quad \mid \mathsf{None} \Rightarrow \\ \qquad vss\,[i] = \mathsf{Some}\ [] * \\ \qquad ws' = [] * \\ \qquad \mathsf{model}\ t\ vss \\ \quad \mid \mathsf{Some}\ v \Rightarrow \\ \qquad \exists\ vs. \\ \qquad vss\,[i] = \mathsf{Some}\ (vs +\!\!+ [v]) * \\ \qquad \mathsf{suffix}\ (vs +\!\!+ [v])\ ws * \\ \qquad ws' = vs * \\ \qquad \mathsf{model}\ t\ (vss\,[i \mapsto vs]) \\ \quad \textbf{end} \end{array} \\ \hdashline res.\ res = o * \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ ws' \end{array}$$

Figure 10.2: `Ws_deques`: Specification (1/2)

WS-DEQUES-STEAL-TO-SPEC

$$
\begin{array}{l}
0 \leq j < sz \ast \\
\mathsf{inv}\ t\ \iota\ sz \ast \\
\mathsf{owner}\ t\ i\ \mathsf{Blocked}\ ws \\
\hdashline
vss.\ \mathsf{model}\ t\ vss \\
\hline
\texttt{steal\_to}\ t\ i\ j\ \fatsemi\ \iota \\
\hline
o.\ \textbf{match}\ o\ \textbf{with} \\
\quad |\ \mathsf{None} \Rightarrow \\
\quad\quad \mathsf{model}\ t\ vss \\
\quad |\ \mathsf{Some}\ v \Rightarrow \\
\quad\quad \exists\ vs. \\
\quad\quad vss\,[j] = \mathsf{Some}\ v :: vs \ast \\
\quad\quad \mathsf{model}\ t\ (vss\,[j \mapsto vs]) \\
\quad \textbf{end} \\
\hdashline
res.\ res = o \ast \\
\quad\quad \mathsf{owner}\ t\ i\ \mathsf{Blocked}\ ws
\end{array}
$$

WS-DEQUES-STEAL-AS-SPEC

$$
\begin{array}{l}
0 < sz \ast \\
\mathsf{inv}\ t\ \iota\ sz \ast \\
\mathsf{owner}\ t\ i\ \mathsf{Blocked}\ ws \ast \\
\mathsf{random\text{-}round.model}\ round\ (sz - 1)\ (sz - 1) \\
\hdashline
vss.\ \mathsf{model}\ t\ vss \\
\hline
\texttt{steal\_as}\ t\ i\ round\ \fatsemi\ \iota \\
\hline
o.\ \textbf{match}\ o\ \textbf{with} \\
\quad |\ \mathsf{None} \Rightarrow \\
\quad\quad \mathsf{model}\ t\ vss \\
\quad |\ \mathsf{Some}\ v \Rightarrow \\
\quad\quad \exists\ j\ vs. \\
\quad\quad i \neq j \ast \\
\quad\quad vss\,[j] = \mathsf{Some}\ v :: vs \ast \\
\quad\quad \mathsf{model}\ t\ (vss\,[j \mapsto vs]) \\
\quad \textbf{end} \\
\hdashline
res.\ \exists\ n. \\
\quad res = o \ast \\
\quad \mathsf{owner}\ t\ i\ \mathsf{Blocked}\ ws \ast \\
\quad \mathsf{random\text{-}round.model}\ round\ (sz - 1)\ n
\end{array}
$$

Figure 10.2: `Ws_deques`: Specification (2/2)

expected way. Note that there are two stealing operations: `steal_to` and `steal_as`[1]; `steal_to` $t\ i\ j$ attempts to steal from the j-th deque only once while `steal_as` $t\ i\ round$ performs one random round (see Section 6.5) of `steal_to` attempts, *i.e.* tries to steal from all other deques in a random order given by *round*.

Remarkably, the specifications of `steal_to` and `steal_as` are weaker than one might expect after reading Section 9.7, which corresponds to standard work-stealing with public deques. Indeed, when these operations return None, the atomic postcondition is not informative: we learn nothing about the observed model values. This reflects the weak behavior of other work-stealing strategies for which we cannot show that we observed an empty deque. Fortunately, this is not a problem in practice; in particular, this weak specification is sufficient for proving the *termination property* of `Pool` (see Section 10.5).

10.2.2 Public deques

10.2.2.1 Implementation

The first realization, `Ws_deques_public`, implements the standard work-stealing strategy with *public deques*. More precisely, it simply relies on a shared array of Chase-Lev work-stealing deques, as implemented in `Saturn` (see Section 9.7). These deques are public in the sense that both their owner and the thieves can access it directly — which requires synchronization.

[1]In fact, only `steal_to` is really needed by `Ws_deques`; the implementation of `steal_as` depends on `steal_to` but is the same for all realizations. Unfortunately, factorizing `steal_as` would require functors, which are not supported by ZooLang.

10.2.2.2 Ghost state

The definition of the predicates and the proofs are relatively straightforward. No special ghost state is needed.

10.2.3 Private deques

10.2.3.1 Implementation

The second realization, `Ws_deques_private`, implements the *receiver-initiated* work-stealing algorithm proposed by Acar et al. [2013][2]. Their idea is to reduce synchronization costs in the fast path of local (owner-only) operations by essentially introducing an indirection. They show that this work-stealing strategy performs well for *fine-grained* parallel programs, *i.e.* when task sizes are small, especially irregular graph computations.

Instead of stealing directly from public deques, thieves follow a protocol: (1) having selected a victim, a thief attempts to send a request by atomically updating the *request cell* of the victim; (2) if the update fails, the thief starts over with another victim, otherwise it awaits a response by repeatedly checking its *response cell*; (3) if the response is negative, the thief starts over, otherwise it returns the task transferred by the victim.

Symmetrically, busy domains regularly poll their request cell and respond accordingly through response cells. Crucially, tasks are stored in private, non-concurrent deques that are only accessed by their owner. In addition, each domain has a *status cell* indicating whether it is (1) blocked, meaning it has no task to share, or (2) non-blocked, meaning it may have tasks to share; before sending a request, thieves check that their victim is non-blocked.

10.2.3.2 Ghost state

External linearization. As the informal description of the algorithm suggests, thieves rely on their victim for locally updating their tasks, including at the logical level. As a result, the linearization of a successful `steal_to` or `steal_as` is always *external*. In Iris, this is handled in the usual way: when a thief sends a request, it also sends an *atomic update* (see Section 3.8), materializing its linearization point, through inv.

channels theory (Figure 10.3). The most interesting bit of the ghost state is the channels theory, responsible for enforcing the communication protocol. It features three predicates: channels-sender, channels-receiver and channels-waiting.

At the start of the protocol, a thief corresponding to the i-th domain owns both $\mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_1\ \mathsf{None}$ and $\mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi_2\ \mathsf{None}$; the third part, $\mathsf{channels\text{-}waiting}\ \gamma\ i$, is stored in inv and remains there until the thief receives a response. Before making any request, the predicates Ψ_1 and Ψ_2 are updated to Ψ (CHANNELS-PREPARE), the postcondition of the thief's atomic update.

When the thief succeeds in sending a request, it also sends channels-receiver to the corresponding victim through inv. Then, when the victim responds, it updates channels-sender (CHANNELS-SEND), consuming channels-waiting in the process, and sends it back to the thief along with Ψ (obtained by triggering the atomic update).

Crucially, the last part of the protocol is divided into two parts, as dictated by the implementation. When the thief detects the response, it updates channels-receiver using

[2] They also propose a *sender-initiated* algorithm that we have not implemented.

$$\textsc{channels-sender-exclusive}\quad \frac{\mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_1\ \mathit{state}_1 \qquad \mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_2\ \mathit{state}_2}{\mathsf{False}}$$

$$\textsc{channels-waiting-receiver}\quad \frac{\mathsf{channels\text{-}waiting}\ \gamma\ i \qquad \mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi\ (\mathsf{Some}\ o)}{\mathsf{False}}$$

$$\textsc{channels-sender-receiver-agree}\quad \frac{\mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_1\ (\mathsf{Some}\ o_1) \qquad \mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi_2\ (\mathsf{Some}\ o_2)}{\triangleright(\Psi_1\ o_1 \equiv \Psi_2\ o_1) * o_1 = o_2 * \mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_1\ (\mathsf{Some}\ o_1) * \mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi_2\ (\mathsf{Some}\ o_1)}$$

$$\textsc{channels-prepare}\quad \frac{\mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_1\ \mathsf{None} \qquad \mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi_2\ \mathsf{None}}{\dot{\Rrightarrow}\ \mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi\ \mathsf{None} * \mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi\ \mathsf{None}}$$

$$\textsc{channels-send}\quad \frac{\mathsf{channels\text{-}waiting}\ \gamma\ i \qquad \mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi\ \mathsf{None}}{\dot{\Rrightarrow}\ \mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi\ (\mathsf{Some}\ o)}$$

$$\textsc{channels-receive}\quad \frac{\mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_1\ (\mathsf{Some}\ o) \qquad \mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi_2\ \mathsf{None}}{\mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_1\ (\mathsf{Some}\ o) * \mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi_2\ (\mathsf{Some}\ o)}$$

$$\textsc{channels-reset}\quad \frac{\mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_1\ (\mathsf{Some}\ o_1) \qquad \mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi_2\ (\mathsf{Some}\ o_2)}{\dot{\Rrightarrow}\ \mathsf{channels\text{-}waiting}\ \gamma\ i * \mathsf{channels\text{-}sender}\ \gamma\ i\ \Psi_1\ \mathsf{None} * \mathsf{channels\text{-}receiver}\ \gamma\ i\ \Psi_2\ \mathsf{None}}$$

Figure 10.3: `Ws_deques_private`: channels theory

$$\mathsf{persistent}\ (\mathsf{inv}\ t)$$

$$\textsc{waiters-waiter-exclusive}\quad \frac{\mathsf{waiter}\ t_1\ wt \qquad \mathsf{waiter}\ t_2\ wt}{\mathsf{False}}$$

$$\textsc{waiters-create-spec}\quad \frac{\mathsf{True}}{\frac{\texttt{create ()}}{t.\ \mathsf{inv}\ t}}$$

$$\textsc{waiters-prepare-wait-spec}\quad \frac{\mathsf{inv}\ t}{\frac{\texttt{prepare\_wait}\ t}{wt.\ \mathsf{waiter}\ t\ wt}}$$

$$\textsc{waiters-cancel-wait-spec}\quad \frac{\mathsf{inv}\ t\ * \quad \mathsf{waiter}\ t\ wt}{\frac{\texttt{cancel\_wait}\ t\ wt}{().\ \mathsf{True}}}$$

$$\textsc{waiters-commit-wait-spec}\quad \frac{\mathsf{inv}\ t\ * \quad \mathsf{waiter}\ t\ wt}{\frac{\texttt{commit\_wait}\ t\ wt}{().\ \mathsf{True}}}$$

$$\textsc{waiters-notify-spec}\quad \frac{\mathsf{inv}\ t}{\frac{\texttt{notify}\ t}{().\ \mathsf{True}}}$$

$$\textsc{waiters-notify-many-spec}\quad \frac{0 \leq n\ * \quad \mathsf{inv}\ t}{\frac{\texttt{notify\_many}\ t\ n}{().\ \mathsf{True}}}$$

Figure 10.4: `Waiters`: Specification

channels-sender but cannot retrieve it yet because the response cell has not been cleared; from that moment on, however, it knows that channels-receiver awaits in inv (CHANNELS-WAITING-RECEIVER). Finally, when the response cell is atomically cleared, the thief can retrieve Ψ and channels-sender; the latter is combined with channels-receiver (CHANNELS-RESET) to reset the state of the protocol and generate a new channels-waiting to replace channels-sender in inv.

10.3 Waiters

In the realizations of the second level, described in the next section, we use a *sleep-based mechanism* to adapt the number of active thieves. The idea is to put to sleep desperate thieves who do not find work after a number of failed steal attempts. In practice, doing so can improve the overall system performance, especially when tasks are scarce.

To manage sleeping thieves, we use the `Waiters` module. Following the design of `Taskflow` [Huang et al., 2022], it implements a *two-phase commit protocol*[3] — `Domainslib`[4] relies on a similar mechanism, although it is not as clear-cut.

10.3.1 Specification

The specification is given in Figure 10.4. It features two predicates: inv and waiter.

The persistent assertion inv t represents the knowledge that t is a set of waiters. It is returned by `create` (WAITERS-CREATE-SPEC) and require by all operations.

The exclusive assertion waiter t wt represents the ownership of waiter wt attached to t. To go to sleep, a thief first calls `prepare_wait`, returning a new waiter (WAITERS-PREPARE-WAIT-SPEC). Then, it performs a few more steal attempts in case tasks were

[3] `https://www.1024cores.net/home/lock-free-algorithms/eventcounts`

[4] `https://github.com/ocaml-multicore/domainslib/blob/main/lib/multi_channel.ml`

to be inserted. If it finds some, it cancels the wait through `cancel_wait` (WAITERS-CANCEL-WAIT-SPEC); otherwise, it commits through `commit_wait` (WAITERS-COMMIT-WAIT-SPEC), which actually puts it to sleep.

To wake up one or several sleeping thieves, one may respectively call `notify` (WAITERS-NOTIFY-SPEC) and `notify_many` (WAITERS-NOTIFY-MANY-SPEC). In practice, this happens when tasks are inserted or the scheduler is killed.

10.3.2 Implementation

The implementation relies on a concurrent queue of waiters. The queue is taken from `Saturn` (see Section 9.2). As for the waiter data structure, it comes from our standard library (see Chapter 6); it simply consists of a flag coupled with a mutex and a condition variable.

`Taskflow` uses a more efficient implementation that can be found in the `Eigen`[5] and `Folly`[6] libraries. Unfortunately, it requires low-level bit manipulation not currently supported by ZooLang.

10.4 Work-stealing hub

At the second level, `Ws_hub` provides a generic interface for a set of tasks supporting work-stealing operations — a so-called "work-stealing hub". We describe its specification (Section 10.4.1) and two current realizations: `Ws_hub_std` (Section 10.4.2) and `Ws_hub_fifo` (Section 10.4.3).

10.4.1 Specification

The specification is given in Figure 10.5. It is more or less similar to Section 10.2.1; we highlight the differences in the following.

The model predicate now carries a multiset of values, as opposed to per-domain lists of values. In other words, deques are not materialized anymore, which gives the realization more freedom. As a matter of fact, the `Ws_hub_fifo` realization (see Section 10.4.3) uses only one shared queue. More generally, this flexibility is needed to support complex scheduling strategies with various task providers — hence the name "work-stealing hub"; for example, both `Domainslib` and `Taskflow`[7] introduce a foreign queue in addition to standard work-stealing deques for external domains to submit tasks. Note that, although this interface does not enforce work-stealing, it must still support it; consequently, most operations are parameterized with a deque index that may or may not be used by the implementation.

The owner predicate carries a status as before but also an *emptiness hint* indicating whether the deque is probably empty (Empty) or non-empty (Nonempty). Crucially, if all deques think they are empty, the hub must be empty (WS-HUB-MODEL-EMPTY).

Contrary to Section 10.2.1, all operations except `unblock` require the hub to be non-blocked. Indeed, blocking and unblocking is performed internally by stealing operations (`steal_until`, `pop_steal_until`, `steal`, `pop_steal`) — we refrained from doing the

[5] `https://gitlab.com/libeigen/eigen/-/blob/master/Eigen/src/ThreadPool/EventCount.h`
[6] `https://github.com/facebook/folly/blob/main/folly/synchronization/EventCount.h`
[7] We have not yet implemented this feature yet but do not anticipate any difficulty.

$$\mathsf{persistent}\ (\mathsf{inv}\ t\ \iota\ sz)$$

WS-HUB-INV-AGREE

$$\frac{\mathsf{inv}\ t\ \iota_1\ sz_1 \qquad \mathsf{inv}\ t\ \iota_2\ sz_2}{sz_1 = sz_2}$$

WS-HUB-OWNER-EXCLUSIVE

$$\frac{\mathsf{owner}\ t\ i\ status_1\ empty_1 \qquad \mathsf{owner}\ t\ i\ status_2\ empty_2}{\mathsf{False}}$$

WS-HUB-MODEL-EMPTY

$$\frac{\mathsf{inv}\ t\ \iota\ sz \qquad \mathsf{model}\ t\ vs \qquad \mathop{\Huge\ast}_{i \in [\![0;sz)} \mathsf{owner}\ t\ i\ status\ \mathsf{Empty}}{vs = \emptyset}$$

WS-HUB-CREATE-SPEC

$$\begin{array}{l}
\quad 0 \le sz \\ \hline
\quad \mathtt{create}\ sz \\ \hline
t.\ \mathsf{inv}\ t\ \iota\ sz\ * \\
\quad \mathsf{model}\ t\ \emptyset\ * \\
\quad \mathop{\Huge\ast}_{i \in [\![0;sz)} \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathsf{Empty}
\end{array}$$

WS-HUB-SIZE-SPEC

$$\begin{array}{c}
\mathsf{inv}\ t\ \iota\ sz \\ \hline
\mathtt{size}\ t \\ \hline
res.\ res = sz
\end{array}$$

WS-HUB-BLOCK-SPEC

$$\begin{array}{l}
\mathsf{inv}\ t\ \iota\ sz\ * \\
\mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ empty \\ \hline
\quad \mathtt{block}\ t\ i \\ \hline
().\ \mathsf{owner}\ t\ i\ \mathsf{Blocked}\ empty
\end{array}$$

WS-HUB-UNBLOCK-SPEC

$$\begin{array}{l}
\mathsf{inv}\ t\ \iota\ sz\ * \\
\mathsf{owner}\ t\ i\ \mathsf{Blocked}\ empty \\ \hline
\quad \mathtt{unblock}\ t\ i \\ \hline
().\ \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ empty
\end{array}$$

WS-HUB-KILLED-SPEC

$$\begin{array}{c}
\mathsf{inv}\ t\ \iota\ sz \\ \hline
\mathtt{killed}\ t \\ \hline
b.\ \mathsf{True}
\end{array}$$

WS-HUB-KILL-SPEC

$$\begin{array}{c}
\mathsf{inv}\ t\ \iota\ sz \\ \hline
\mathtt{kill}\ t \\ \hline
().\ \mathsf{True}
\end{array}$$

WS-HUB-PUSH-SPEC

$$\begin{array}{l}
\mathsf{inv}\ t\ \iota\ sz\ * \\
\mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ empty \\ \hdashline
\quad vs.\ \mathsf{model}\ t\ vs \\ \hline
\quad \mathtt{push}\ t\ i\ v\ ⨾\ \iota \\ \hline
\quad \mathsf{model}\ t\ (\{v\} \uplus vs) \\ \hdashline
().\ \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathsf{Nonempty}
\end{array}$$

WS-HUB-POP-SPEC

$$\begin{array}{l}
\quad \mathsf{inv}\ t\ \iota\ sz\ * \\
\quad \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ empty \\ \hdashline
\quad vs.\ \mathsf{model}\ t\ vs \\ \hline
\quad \mathtt{pop}\ t\ i\ ⨾\ \iota \\ \hline
\quad o.\ \mathbf{match}\ o\ \mathbf{with} \\
\quad\quad \mid \mathsf{None} \Rightarrow \\
\quad\quad\quad \mathsf{model}\ t\ vs \\
\quad\quad \mid \mathsf{Some}\ v \Rightarrow \\
\quad\quad\quad \exists\ vs'. \\
\quad\quad\quad vs = \{v\} \uplus vs'\ * \\
\quad\quad\quad \mathsf{model}\ t\ vs' \\
\quad\quad \mathbf{end} \\ \hdashline
res.\ res = o\ * \\
\quad \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ (\mathbf{if}\ o\ \mathbf{then}\ empty\ \mathbf{else}\ \mathsf{Empty})
\end{array}$$

Figure 10.5: `Ws_hub`: Specification (1/3)

WS-HUB-STEAL-UNTIL-SPEC

$0 \leq \mathit{max\text{-}round\text{-}noyield} \ast$
$\mathsf{inv}\ t\ \iota\ \mathit{sz} \ast$
$\mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathit{empty} \ast$
$\{\, \mathsf{True} \,\}\ \mathit{pred}\ ()\ \{\, b.\, \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ \mathsf{True} \,\}$

- -

$\mathit{vs}.\ \mathsf{model}\ t\ \mathit{vs}$

`steal_until` $t\ i\ \mathit{max\text{-}round\text{-}noyield}\ \mathit{pred}\ ⨟\ \iota$

$o.\ \textbf{match}\ o\ \textbf{with}$
$\mid \mathsf{None} \Rightarrow$
$\quad \mathsf{model}\ t\ \mathit{vs}$
$\mid \mathsf{Some}\ v \Rightarrow$
$\quad \exists\ \mathit{vs}'.$
$\quad \mathit{vs} = \{v\} \uplus \mathit{vs}' \ast$
$\quad \mathsf{model}\ t\ \mathit{vs}'$
$\textbf{end}$

- -

$\mathit{res}.\ \mathit{res} = o \ast$
$\mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathit{empty} \ast$
$\textbf{if}\ o\ \textbf{then}\ \mathsf{True}\ \textbf{else}\ P$

WS-HUB-POP-STEAL-UNTIL-SPEC

$0 \leq \mathit{max\text{-}round\text{-}noyield} \ast$
$\mathsf{inv}\ t\ \iota\ \mathit{sz} \ast$
$\mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathit{empty} \ast$
$\{\, \mathsf{True} \,\}\ \mathit{pred}\ ()\ \{\, b.\, \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ \mathsf{True} \,\}$

- -

$\mathit{vs}.\ \mathsf{model}\ t\ \mathit{vs}$

`pop_steal_until` $t\ i\ \mathit{max\text{-}round\text{-}noyield}\ \mathit{pred}\ ⨟\ \iota$

$o.\ \textbf{match}\ o\ \textbf{with}$
$\mid \mathsf{None} \Rightarrow$
$\quad \mathsf{model}\ t\ \mathit{vs}$
$\mid \mathsf{Some}\ v \Rightarrow$
$\quad \exists\ \mathit{vs}'.$
$\quad \mathit{vs} = \{v\} \uplus \mathit{vs}' \ast$
$\quad \mathsf{model}\ t\ \mathit{vs}'$
$\textbf{end}$

- -

$\mathit{res}.\ \exists\ \mathit{empty}.$
$\mathit{res} = o \ast$
$\mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathit{empty} \ast$
$\textbf{if}\ o\ \textbf{then}\ \mathsf{True}\ \textbf{else}\ \mathit{empty} = \mathsf{Empty} \ast P$

Figure 10.5: `Ws_hub`: Specification (2/3)

WS-HUB-STEAL-SPEC

$$\dfrac{\begin{array}{c} 0 \leq \mathit{max\text{-}round\text{-}noyield} \ast \\ 0 \leq \mathit{max\text{-}round\text{-}yield} \ast \\ \mathsf{inv}\ t\ \iota\ \mathit{sz} \ast \\ \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathit{empty} \\ \hdashline \mathit{vs}.\ \mathsf{model}\ t\ \mathit{vs} \end{array}}{\begin{array}{c} \texttt{steal}\ t\ i\ \mathit{max\text{-}round\text{-}noyield}\ \mathit{max\text{-}round\text{-}yield}\ ⨟\ \iota \\ \hline o.\ \textbf{match}\ o\ \textbf{with} \\ \mid \mathsf{None} \Rightarrow \\ \quad \mathsf{model}\ t\ \mathit{vs} \\ \mid \mathsf{Some}\ v \Rightarrow \\ \quad \exists\ \mathit{vs}'. \\ \quad \mathit{vs} = \{v\} \uplus \mathit{vs}' \ast \\ \quad \mathsf{model}\ t\ \mathit{vs}' \\ \textbf{end} \\ \hdashline \mathit{res}.\ \mathit{res} = o \ast \\ \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathit{empty} \end{array}}$$

WS-HUB-POP-STEAL-SPEC

$$\dfrac{\begin{array}{c} 0 \leq \mathit{max\text{-}round\text{-}noyield} \ast \\ 0 \leq \mathit{max\text{-}round\text{-}yield} \ast \\ \mathsf{inv}\ t\ \iota\ \mathit{sz} \ast \\ \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathit{empty} \\ \hdashline \mathit{vs}.\ \mathsf{model}\ t\ \mathit{vs} \end{array}}{\begin{array}{c} \texttt{pop\_steal}\ t\ i\ \mathit{max\text{-}round\text{-}noyield}\ \mathit{max\text{-}round\text{-}yield}\ ⨟\ \iota \\ \hline o.\ \textbf{match}\ o\ \textbf{with} \\ \mid \mathsf{None} \Rightarrow \\ \quad \mathsf{model}\ t\ \mathit{vs} \\ \mid \mathsf{Some}\ v \Rightarrow \\ \quad \exists\ \mathit{vs}'. \\ \quad \mathit{vs} = \{v\} \uplus \mathit{vs}' \ast \\ \quad \mathsf{model}\ t\ \mathit{vs}' \\ \textbf{end} \\ \hdashline \mathit{res}.\ \exists\ \mathit{empty}. \\ \mathit{res} = o \ast \\ \mathsf{owner}\ t\ i\ \mathsf{Nonblocked}\ \mathit{empty} \ast \\ \textbf{if}\ o\ \textbf{then}\ \mathsf{True}\ \textbf{else}\ \mathit{empty} = \mathsf{Empty} \end{array}}$$

Figure 10.5: `Ws_hub`: Specification (3/3)

same in `Ws_deques` for performance reasons. However, we still need to expose the `block` and `unblock` operations, which are used in `Pool` (see Section 10.5).

Speaking of stealing operations[8], they evolved significantly. `steal_until` *t i max-round-noyield pred* (WS-HUB-STEAL-UNTIL-SPEC) repeatedly attempts to steal from other deques until *pred* returns `true`; *max-round-noyield* is an upper bound on the number of attempts that may be performed without yielding, *i.e.* calling `Domain.cpu_relax` (see Section 2.1). `steal` *t i max-round-noyield max-round-noyield* (WS-HUB-STEAL-SPEC) repeatedly attempts to steal from other deques until it succeeds or the hub is killed; *max-round-noyield* and *max-round-yield* are upper bounds on the number of attempts that may be performed respectively without and with yielding before pausing, *e.g.* using `Waiters`. Each of these two operations has a variant, respectively `pop_steal_until` and `pop_steal`, that first calls `pop`.

As mentioned above, a hub can be killed using the `kill` operation (WS-HUB-KILL-SPEC), which is supposed to notify all workers, possibly waking up some in the process.

10.4.2 Work-stealing strategy

10.4.2.1 Implementation

The first realization, `Ws_hub_std`, implements the standard randomized work-stealing strategy. Under the hood, any work-stealing algorithm may be used, provided that it fits into the `Ws_hub` interface; in particular, it can instantiated with both realization of `Ws_deques`.

10.4.2.2 Ghost state

The definition of the predicates and the proofs are relatively straightforward. No special ghost state is needed.

10.4.3 FIFO strategy

10.4.3.1 Implementation

The second realization, `Ws_hub_fifo`, implements a simple "first-in first-out" scheduling strategy. All workers push and pop tasks from a shared concurrent queue taken from `Saturn` (Section 9.2); thieves also attempts to pop from the queue. `Moonpool` adopted a similar strategy[9].

As explained by Cruanes[10], the point of this strategy is to provide better *latency* than work-stealing — as demanded by certain applications like network servers — at the cost of a lower throughput. Indeed, contrary to work-stealing, older tasks have priority over younger tasks.

However, this strategy may also have undesirable consequences. For example, in divide-and-conquer algorithms, this strategy corresponds to *breadth-first* search, whereas work-stealing corresponds to *depth-first* search. On large problems, the former may be unsustainable; on some benchmarks (see Section 10.8), especially for small cutoffs, `Moonpool` saturates the memory.

[8] Similarly to `Ws_deques`, only `steal_until` and `steal` are really needed by `Ws_hub`. Factorizing `pop_steal_until` and `pop_steal` would require functors, which are not supported by ZooLang.

[9] `https://github.com/c-cube/moonpool/blob/main/src/core/fifo_pool.ml`

[10] `https://github.com/c-cube/moonpool/blob/main/src/core/fifo_pool.mli`

10.4.3.2 Ghost state

The definition of the predicates and the proofs are relatively straightforward. Special ghost state is required to enforce the "emptiness consensus" (WS-HUB-MODEL-EMPTY); we refer to the mechanization for details.

10.5 Pool

At the third level, Pool implements a task scheduler on top of a given realization of Ws_hub. It offers essentially the same functionalities as Domainslib with a few notable differences. (1) Exceptions raised by tasks are not caught and therefore not re-raised properly by the scheduler since ZooLang does not currently support them. (2) Since ZooLang does not support algebraic effects [Sivaramakrishnan et al., 2021] neither, the interface is slightly more involved (see *execution contexts* in Section 10.5.1) and the implementation of futures (see Section 10.5.1), especially the `wait` function, is less efficient (see Section 10.5.2).

Moreover, this limitation imposes a *child-stealing* strategy, as opposed to a *continuation-stealing* strategy that would require capturing the continuation of a computation.

Also, this makes it difficult to implement a `yield` operation[11], *i.e.* an operation that yields control to the scheduler, letting it reschedule the current task later.

10.5.1 Specification

The specification is given in Figure 10.6. As it is quite a mouthful, we divide its presentation into two parts: (1) the core interface and (2) the interface for *futures*[12].

Core interface. The persistent assertion inv t vsz represents the knowledge that t is a valid scheduler; sz is the number of worker domains. It is returned by `create` (POOL-CREATE-SPEC) and required only by `size` (POOL-SIZE-SPEC). Its only purpose is to record the immutable characteristics of the scheduler.

The assertion model t represents the ownership of scheduler t. It is returned by `create` (POOL-CREATE-SPEC) and required by external operations (POOL-RUN-SPEC, POOL-KILL-SPEC). For example, `run` t $task$ submits $task$ to scheduler t; it returns both model and the output predicate of $task$.

The assertion context t ctx $scope$ represents the ownership of *execution context* ctx attached to scheduler t; $scope$ is a purely logical parameter connecting input and output context, which is necessary in the proof. Any task execution happens under such a context (POOL-RUN-SPEC, POOL-ASYNC-SILENT-SPEC, POOL-WAIT-UNTIL-SPEC). In particular, all internal operations require and return context. For example, `async_silent` ctx $task$ submits $task$ asynchronously while executing under context ctx; $task$ must be shown to execute safely under any context attached to the same scheduler (POOL-ASYNC-SILENT-SPEC).

The persistent assertion finished t represents the knowledge that scheduler t has finished, meaning all submitted tasks were executed. It can be obtained by calling `kill` (POOL-KILL-SPEC).

[11] `Domainslib` does not currently provide a `yield` operation but it can be easily implemented.

[12] Futures are called *promises* in `Domainslib`. In fact, the two notions are often used in conjunction to represent the two sides of the same object.

$$\text{persistent } (\text{inv } t\ sz) \qquad \text{persistent } (\text{obligation } t\ P) \qquad \text{persistent } (\text{finished } t)$$

$$\frac{\begin{array}{l}\text{inv } t\ sz_1 \\ \text{inv } t\ sz_2\end{array}}{sz_1 = sz_2}\ \textsc{Pool-Inv-Agree} \qquad \frac{\begin{array}{l}\text{obligation } t\ P \\ \text{finished } t\end{array}}{\triangleright \Box P}\ \textsc{Pool-Obligation-Finished}$$

$$\frac{\frac{0 \le sz}{\texttt{create } sz}}{\begin{array}{l} t.\ \text{inv } t\ sz\ * \\ \text{model } t\end{array}}\ \textsc{Pool-Create-Spec}$$

$$\frac{\frac{\begin{array}{l}\text{model } t\ * \\ \quad \forall\ ctx\ scope. \\ \quad \text{context } t\ ctx\ scope \twoheadrightarrow \\ \quad \text{wp } task\ ctx \left\{ \begin{array}{l} v.\ \text{context } t\ ctx\ scope\ * \\ \quad \Psi\ v \end{array} \right\}\end{array}}{\texttt{run } t\ task}}{\begin{array}{l} v.\ \text{model } t\ * \\ \quad \Psi\ v\end{array}}\ \textsc{Pool-Run-Spec}$$

$$\frac{\frac{\text{model } t}{\texttt{kill } t}}{().\ \text{finished } t}\ \textsc{Pool-Kill-Spec}$$

$$\frac{\frac{\begin{array}{l}\text{inv } t\ sz\ * \\ \text{context } t\ ctx\ scope\end{array}}{\texttt{size } ctx}}{\begin{array}{l} res.\ res = sz\ * \\ \quad \text{context } t\ ctx\ scope\end{array}}\ \textsc{Pool-Size-Spec}$$

$$\frac{\frac{\begin{array}{l}\text{context } t\ ctx\ scope\ * \\ \quad \forall\ ctx\ scope. \\ \quad \text{context } t\ ctx\ scope \twoheadrightarrow \\ \quad \text{wp } task\ ctx \left\{ \begin{array}{l} \_.\ \text{context } t\ ctx\ scope\ * \\ \quad \triangleright \Box P \end{array} \right\}\end{array}}{\texttt{async\_silent } ctx\ task}}{\begin{array}{l} ().\ \text{context } t\ ctx\ scope\ * \\ \quad \text{obligation } t\ P\end{array}}\ \textsc{Pool-Async-Silent-Spec}$$

$$\frac{\frac{\begin{array}{l}\text{context } t\ ctx\ scope\ * \\ \{\ \text{True}\ \}\ pred\ ()\ \{\ b.\ \textbf{if } b \textbf{ then } P \textbf{ else } \text{True}\ \}\end{array}}{\texttt{wait\_until } ctx\ pred}}{\begin{array}{l} ().\ \text{context } t\ ctx\ scope\ * \\ \quad P\end{array}}\ \textsc{Pool-Wait-Until-Spec}$$

Figure 10.6: `Pool`: Specification (1/2)

$$\textsf{persistent}\ (\textsf{future-inv}\ \mathit{fut}\ \Psi\ \Xi) \qquad \textsf{persistent}\ (\textsf{future-result}\ \mathit{fut}\ v)$$

POOL-FUTURE-CONSUMER-DIVIDE

$$\frac{\begin{array}{c}\textsf{future-inv}\ \mathit{fut}\ \Psi\ \Xi \\ \textsf{future-consumer}\ \mathit{fut}\ \mathrm{X} \\ \forall\ v.\ \mathrm{X}\ v \twoheadrightarrow \mathop{\bigast}_{\mathrm{X} \in \mathrm{X}s} \mathrm{X}\ v\end{array}}{\Rrightarrow \mathop{\bigast}_{\mathrm{X} \in \mathrm{X}s} \textsf{future-consumer}\ \mathit{fut}\ \mathrm{X}}$$

POOL-FUTURE-RESULT-AGREE

$$\frac{\begin{array}{c}\textsf{future-result}\ \mathit{fut}\ v_1 \\ \textsf{future-result}\ \mathit{fut}\ v_2\end{array}}{v_1 = v_2}$$

POOL-FUTURE-INV-RESULT

$$\frac{\begin{array}{c}\textsf{future-inv}\ \mathit{fut}\ \Psi\ \Xi \\ \textsf{future-result}\ \mathit{fut}\ v\end{array}}{\Rrightarrow \triangleright \Box\, \Xi\ v}$$

POOL-FUTURE-INV-RESULT-CONSUMER

$$\frac{\begin{array}{c}\textsf{future-inv}\ \mathit{fut}\ \Psi\ \Xi \\ \textsf{future-result}\ \mathit{fut}\ v \\ \textsf{future-consumer}\ \mathit{fut}\ \mathrm{X}\end{array}}{\Rrightarrow \triangleright^2\, \mathrm{X}\ v}$$

POOL-FINISHED-FUTURE-RESULT

$$\frac{\begin{array}{c}\textsf{finished}\ t \\ \textsf{future-inv}\ \mathit{fut}\ \Psi\ \Xi\end{array}}{\exists\ v.\ \textsf{future-result}\ \mathit{fut}\ v}$$

POOL-ASYNC-SPEC

$$\frac{\begin{array}{l}\textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \ast \\ \quad \forall\ \mathit{ctx}\ \mathit{scope}. \\ \quad \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \twoheadrightarrow \\ \quad \textsf{wp}\ \mathit{task}\ \mathit{ctx} \left\{\begin{array}{l} v.\ \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \ast \\ \quad \triangleright \Psi\ v \ast \\ \quad \triangleright \Box\, \Xi\ v\end{array}\right\}\end{array}}{\begin{array}{c}\texttt{async}\ \mathit{ctx}\ \mathit{task}\end{array}}$$

$$\begin{array}{l}\mathit{fut}.\ \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \ast \\ \qquad \textsf{future-inv}\ \mathit{fut}\ \Psi\ \Xi \ast \\ \qquad \textsf{future-consumer}\ \mathit{fut}\ \Psi \ast \\ \qquad \textsf{obligation}\ t\ (\exists\ v.\ \textsf{future-result}\ \mathit{fut}\ v)\end{array}$$

POOL-WAIT-SPEC

$$\frac{\begin{array}{l}\textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \ast \\ \textsf{future-inv}\ \mathit{fut}\ \Psi\ \Xi\end{array}}{\texttt{wait}\ \mathit{ctx}\ \mathit{fut}}$$

$$\begin{array}{l} v.\ \pounds\, 2 \ast \\ \quad \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \ast \\ \quad \textsf{future-result}\ \mathit{fut}\ v\end{array}$$

POOL-ITER-SPEC

$$\frac{\begin{array}{l}\textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \ast \\ \textsf{future-inv}\ \mathit{fut}\ \Psi\ \Xi \ast \\ \quad \forall\ \mathit{ctx}\ \mathit{scope}\ v. \\ \quad \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \twoheadrightarrow \\ \quad \textsf{future-result}\ \mathit{fut}\ v \twoheadrightarrow \\ \quad \textsf{wp}\ \mathit{fn}\ \mathit{ctx}\ v\ \{ ().\ \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \}\end{array}}{\texttt{iter}\ \mathit{ctx}\ \mathit{fut}\ \mathit{fn}}$$

$$().\ \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope}$$

POOL-MAP-SPEC

$$\frac{\begin{array}{l}\textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \ast \\ \textsf{future-inv}\ \mathit{fut}_1\ \Psi_1\ \Xi_1 \ast \\ \quad \forall\ \mathit{ctx}\ \mathit{scope}\ v_1. \\ \quad \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \twoheadrightarrow \\ \quad \textsf{future-result}\ \mathit{fut}_1\ v_1 \twoheadrightarrow \\ \quad \textsf{wp}\ \mathit{fn}\ \mathit{ctx}\ v_1 \left\{\begin{array}{l} v_2.\ \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \ast \\ \quad \triangleright \Psi_2\ v_2 \ast \\ \quad \triangleright \Box\, \Xi_2\ v_2\end{array}\right\}\end{array}}{\texttt{map}\ \mathit{ctx}\ \mathit{fut}_1\ \mathit{fn}}$$

$$\begin{array}{l}\mathit{fut}_2.\ \textsf{context}\ t\ \mathit{ctx}\ \mathit{scope} \ast \\ \qquad \textsf{future-inv}\ \mathit{fut}_2\ \Psi_2\ \Xi_2 \ast \\ \qquad \textsf{future-consumer}\ \mathit{fut}_2\ \Psi_2\end{array}$$

Figure 10.6: `Pool`: Specification (2/2)

The persistent assertion obligation t P represents a proof obligation attached to scheduler t. It allows retrieving P once t has finished executing (POOL-OBLIGATION-FINISHED). Obligations are obtained by submitting tasks through `async_silent` (POOL-ASYNC-SILENT-SPEC).

Futures. `async` also allows submitting a task asynchronously while executing under a context (POOL-SYNC-SPEC). Furthermore, it returns a *future* representing the future result of the task. To actually get the result, one must call `wait` (POOL-WAIT-SPEC). `iter` *ctx fut fn* attaches callback *fn* to *fut* (POOL-ITER-SPEC) and `map` *ctx* fut_1 *fn* creates a new future to be resolved after fut_1 (POOL-MAP-SPEC).

Futures are specified using three predicates: future-inv, future-result and future-consumer. Their behavior is exactly the same as in the specification of ivars (see Section 6.10). In particular, future-consumer *fut* X represents the right to consume X once *fut* has been determined (POOL-FUTURE-INV-RESULT-CONSUMER); it can be divided using POOL-FUTURE-CONSUMER-DIVIDE.

10.5.2 Implementation

Worker domains. The implementation relies on a pool of worker domains and a work-stealing hub. Each worker runs the following loop: (1) get a task using `Ws_hub.pop_steal`; (2) if it fails, the scheduler has been killed and so the worker stops, otherwise execute the task in the context of the current worker; (3) start over.

Blocking. Care must be taken to block and unblock work-stealing deques properly. When the scheduler is killed, it is crucial that workers block their deque before stopping; otherwise, the scheduler may never terminate because of a running worker waiting forever for a response from a stopped but unblocked worker. Also, the main domain, from which tasks can be submitted externally through `run`, must unblock when it is executing tasks and block when it is not.

Shutdown. In `Domainslib`, scheduler shutdown consists in submitting special tasks through the main domain; when a worker finds such a task, it quickly stops. However, this simple mechanism has at least two drawbacks: (1) it introduces an indirection for every regular task, which may be expensive; (2) it works well under standard work-stealing but is more difficult to implement under other scheduling strategies, especially work-stealing with private deques (see Section 10.2.3). Consequently, we use an alternative mechanism implemented at the level of `Ws_hub`: a shared flag, regularly checked in `Ws_hub.steal` and `Ws_hub.pop_steal`, is set when the scheduler is killed.

Futures. Futures are implemented as ivars (see Section 6.10). `async` creates an ivar and calls `async_silent` to resolve it asynchronously. `wait` calls `wait_until` to wait *actively* until the ivar is resolved and returns the resulting value. `wait_until` runs a loop similar to that of the worker domains described above; the wait is *active* in the sense that the domain participate in the execution of tasks. Consequently, `wait` and `wait_until` can be nested. This can be a problem in practice because it increases the call stack size in an arbitrary way, potentially causing stack overflow.

Instead, `Domainslib` leverages algebraic effects: awaiting a future captures the continuation and stores it into the future; when the future is resolved, it resubmits all the waiting tasks. This avoids any stack issue and is probably more efficient, since no polling is necessary.

10.5.3 Ghost state

The most interesting part of the ghost state is the handling of proof obligations (obligation), especially the proof of POOL-OBLIGATION-FINISHED. The idea is the following: at any point in time, a submitted task is either (1) finished, (2) in the global work-stealing hub, or (3) in the local task stack of one of the workers. When the scheduler is finished, all the workers are finished; therefore, the task stacks are empty and so is the global hub, thanks to WS-HUB-MODEL-EMPTY; thus, any submitted task must be finished and the corresponding obligation must be fulfilled.

Emitting proof obligations for futures, *e.g.* returning obligation t ($\exists\, v.$ future-result fut v) in ASYNC-SPEC, makes things more complicated. Indeed, in the presence of `iter` and `map`, futures form a *forest*; the previous reasoning only applies to the *root futures*. However, any unfinished future is *reachable* from a root future, *i.e.* there is a path from a root future to the future in the forest. When the scheduler is finished, there are no more roots, hence no future is reachable, hence all futures are finished. We formalized this reasoning on paper, which requires additional ghost state, but have not mechanized it yet.

10.6 Vertex

At the fourth level, `Vertex` implements *DAG-calculus* [Acar et al., 2016], *i.e.* a task graph abstraction. `Taskflow` offers similar, although much more developed, abstractions. The longer term goal is to support the more practical `Taskflow` interface, including static, dynamic, module and condition tasks.

The raison d'être of these works is to represent more interesting dependency relations than is possible using standard parallel primitives (`fork`/`join`, futures, *etc.*) in order to express irregular parallel computations, *e.g.* those for graph problems.

Concretely, this takes the form of a simple and elegant programming model: a parallel computation is seen as a graph where vertices represent basic sequential computations and edges represent dependencies between vertices. A vertex can be executed only when its predecessors, *i.e.* dependencies, are finished. Crucially, the structure of the graph is not static: while executing, a vertex may create new vertices and edges. Naturally, with great expressivity comes great responsibility: care must be taken not to introduce cycles in the graph, although the model does allow looping on a vertex.

10.6.1 Specification

The specification is given in Figure 10.7. It features no less than six predicates: inv, model, running, output, finished and predecessor.

The persistent assertion inv t P R represents the knowledge that t is a valid vertex; P is the *non-persistent* output while R is the *persistent* output. It is returned by `create` (VERTEX-CREATE-SPEC) and required by most operations.

The exclusive assertion model t *task* *iter* represents the ownership of vertex t. It is returned by `create` (VERTEX-CREATE-SPEC). *task* is the current computation attached to t; it can accessed using `task` (VERTEX-TASK-SPEC) and `set_task` (VERTEX-SET-TASK-SPEC). *iter* is the current *logical iteration* of t. Indeed, a vertex may be executed several

$$\textsf{persistent}\ (\textsf{inv}\ t\ P\ R) \qquad \textsf{persistent}\ (\textsf{running}\ \mathit{iter}) \qquad \textsf{persistent}\ (\textsf{finished}\ t)$$

$$\textsf{persistent}\ (\textsf{predecessor}\ t\ \mathit{iter})$$

$$\frac{\begin{array}{c}\textsf{model}\ t\ \mathit{task}_1\ \mathit{iter}_1 \\ \textsf{model}\ t\ \mathit{task}_2\ \mathit{iter}_2\end{array}}{\textsf{False}}\ \textsc{Vertex-model-exclusive} \qquad \frac{\begin{array}{c}\textsf{model}\ t\ \mathit{task}\ \mathit{iter} \\ \textsf{finished}\ t\end{array}}{\textsf{False}}\ \textsc{Vertex-model-finished}$$

$$\frac{\begin{array}{c}\textsf{inv}\ t\ P\ R \\ \textsf{output}\ t\ Q \\ Q \twoheadrightarrow \mathop{\Huge *}_{Q \in Qs} Q\end{array}}{\Rrightarrow \mathop{\Huge *}_{Q \in Qs} \textsf{output}\ t\ Q}\ \textsc{Vertex-output-divide}$$

$$\frac{\begin{array}{c}\textsf{predecessor}\ t\ \mathit{iter} \\ \textsf{running}\ \mathit{iter}\end{array}}{\textsf{finished}\ t}\ \textsc{Vertex-predecessor-finished} \qquad \frac{\begin{array}{c}\textsf{inv}\ t\ P\ R \\ \textsf{finished}\ t\end{array}}{\Rrightarrow \triangleright \Box R}\ \textsc{Vertex-inv-finished}$$

$$\frac{\begin{array}{c}\textsf{inv}\ t\ P\ R \\ \textsf{finished}\ t \\ \textsf{output}\ t\ Q\end{array}}{\Rrightarrow \triangleright^2 Q}\ \textsc{Vertex-inv-finished-output}$$

$$\textsc{Vertex-create-spec}\quad \begin{array}{l}\hline \textsf{True} \\ \hline \texttt{create}\ \mathit{task} \\ \hline t.\ \exists\ \mathit{iter}. \\ \quad \textsf{inv}\ t\ P\ R\ * \\ \quad \textsf{model}\ t\ (\textsf{option.get}\ (\texttt{fun: <> => false})\ \mathit{task})\ \mathit{iter}\ * \\ \quad \textsf{output}\ t\ P\end{array}$$

$$\textsc{Vertex-task-spec}\quad \begin{array}{c}\textsf{model}\ t\ \mathit{task}\ \mathit{iter} \\ \hline \texttt{task}\ t \\ \hline \mathit{res}.\ \mathit{res} = \mathit{task}\ * \\ \quad \textsf{model}\ t\ \mathit{task}\ \mathit{iter}\end{array} \qquad \textsc{Vertex-set-task-spec}\quad \begin{array}{c}\textsf{model}\ t\ \mathit{task}_1\ \mathit{iter} \\ \hline \texttt{set\_task}\ t\ \mathit{task}_2 \\ \hline ().\ \textsf{model}\ t\ \mathit{task}_2\ \mathit{iter}\end{array}$$

$$\textsc{Vertex-precede-spec}\quad \begin{array}{c}\textsf{inv}\ t_1\ P_1\ R_1\ * \\ \textsf{inv}\ t_2\ P_2\ R_2\ * \\ \textsf{model}\ t_2\ \mathit{task}\ \mathit{iter} \\ \hline \texttt{precede}\ t_1\ t_2 \\ \hline ().\ \textsf{model}\ t_2\ \mathit{task}\ \mathit{iter}\ * \\ \quad \textsf{predecessor}\ t_1\ \mathit{iter}\end{array} \qquad \textsc{Vertex-release-spec}\quad \begin{array}{l}\textsf{pool.context}\ \mathit{pool}\ \mathit{ctx}\ \mathit{scope}\ * \\ \textsf{inv}\ t\ P\ R\ * \\ \textsf{model}\ t\ \mathit{task}\ \mathit{iter}\ * \\ \textsf{wp}\ t\ P\ R\ \mathit{task}\ \mathit{iter} \\ \hline \quad \texttt{release}\ \mathit{ctx}\ t \\ \hline ().\ \textsf{pool.context}\ \mathit{pool}\ \mathit{ctx}\ \mathit{scope}\end{array}$$

Figure 10.7: `Vertex`: Specification

$$
\begin{array}{rcl}
\textsf{wp}\ t\ P\ R\ \mathit{task}\ \mathit{iter} & \triangleq & \forall\ \mathit{pool}\ \mathit{ctx}\ \mathit{scope}\ \mathit{iter}'.\\
& & \textsf{pool.context}\ \mathit{pool}\ \mathit{ctx}\ \mathit{scope} -\!\!* \\
& & \textsf{running}\ \mathit{iter} -\!\!* \\
& & \textsf{model}\ t\ \mathit{task}\ \mathit{iter}' -\!\!* \\
& & \textsf{wp}\ \mathit{task}\ \mathit{ctx} \left\{ \begin{array}{l}
b.\ \exists\ \mathit{task}.\\
\quad \triangleright\, \textsf{pool.context}\ \mathit{pool}\ \mathit{ctx}\ \mathit{scope} * \\
\quad \triangleright\, \textsf{model}\ t\ \mathit{task}\ \mathit{iter}' * \\
\quad \textbf{if}\ b\ \textbf{then} \\
\qquad \triangleright\, \textsf{wp}\ t\ P\ R\ \mathit{task}\ \mathit{iter}' \\
\quad \textbf{else} \\
\qquad \triangleright\, P * \\
\qquad \triangleright\, \square\, R
\end{array} \right\}
\end{array}
$$

Figure 10.8: `Vertex`: Weakest precondition

times; more precisely, a vertex task returns a boolean indicating whether the vertex should be re-executed.

The persistent assertion running *iter* represents the knowledge that the iteration identified by *iter* has started — it may be finished and obsoleted by subsequent iterations.

The assertion output t Q represents the right to consume Q from the non-persistent output of t once the latter has finished executing. It is returned by `create` (VERTEX-CREATE-SPEC) with the full non-persistent output and can then be divided using VERTEX-OUTPUT-DIVIDE.

The persistent assertion finished t represents the knowledge that vertex t has finished executing. It allows retrieving both the persistent (VERTEX-INV-FINISHED) and non-persistent (VERTEX-INV-FINISHED-OUTPUT) output of t.

The persistent assertion predecessor t *iter* represents the knowledge that iteration *iter* has predecessor t, *i.e.* *iter* can only run once vertex t has finished (VERTEX-PREDECESSOR-FINISHED). It can be obtained through `precede` (VERTEX-PRECEDE-SPEC), including while the target vertex is executing; in other words, a vertex may add dependencies to itself so that its next iteration only starts when the new dependencies have finished.

The most important operation is `release` (VERTEX-RELEASE-SPEC), which declares a vertex ready for execution, provided that its dependencies (more precisely, those of the corresponding iteration) have finished. It requires a *vertex weakest precondition* wp t P R *task iter*, whose definition is given in Figure 10.8. Intuitively, wp is the counterpart of the standard weakest precondition for vertices: in any execution context, given the possession of vertex t, *task* executes safely; if it returns false, it yields the two outputs; otherwise, we must show wp again for the new task at the next iteration. Crucially, for the execution of *task*, we learn that iteration *iter* has indeed started and therefore its predecessors have finished thanks to VERTEX-PREDECESSOR-FINISHED.

This interface is slightly different from that of Acar et al. [2016]. Indeed, in the original DAG-calculus, vertex re-execution is performed using the `yield` operation, which behaves the same as returning `true` in our interface except the computation is resumed at the same point. This is more convenient than returning a boolean and possibly going through mutable state to resume the computation properly. Unfortunately again, this mechanism requires capturing the continuation of the computation using algebraic effects, which are not supported by ZooLang.

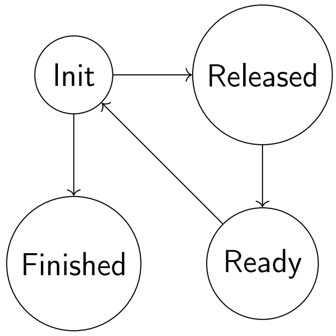


Figure 10.9: `Vertex`: Logical state

10.6.2 Implementation

Our implementation is very close to that of Acar et al. [2016]. The representation of a vertex consists of: (1) the current task, (2) an atomic counter corresponding to the number of unfinished predecessors, (3) a closable concurrent stack from `Saturn` (see Section 9.1) corresponding to the successors. When creating a new edge through `precede`, the target is added to the successors of the source and the counter of the target is incremented. After executing, a vertex atomically closes its successors and decrements their counter, releasing those with zero remaining predecessors.

Actually, a vertex counter does not exactly correspond to the number of predecessors. Before the vertex is released for the first time and during its execution, there is one phantom predecessor preventing premature release; it is removed by `release`.

10.6.3 Ghost state

The implementation is fairly short but subtle, and so is the ghost state. We discuss two interesting aspects of the latter.

Recursive invariant. Since a vertex stores its successors, which are themselves vertices, the inv predicate is *recursively* defined. This involves an Iris fixpoint. The same thing can be said and the same mechanism is used for wp.

Logical state. As often when studying non-trivial fine-grained concurrent data structures, the physical state of a vertex does not determine its logical state. The set of logical states and the transitions between them are displayed in Figure 10.9. Before a vertex is released for the first time and during its execution, it is in the Init state; at this point, the phantom predecessor is active. When it finishes executing, it transitions to the Finished state. When it is released or re-executed, it transitions to the Released state; at this point, it is not ready to be executed. When its counter reaches zero, meaning it has no more predecessors, it transitions to the Ready state; at this point, it is submitted to the scheduler. When it is scheduled and starts executing, it transitions back to the Init state.

10.7 Unverified algorithms

On top of `Pool`, we also implemented standard parallel iterators 🐫 that are particularly useful for benchmarks (see Section 10.8): `for_`, `divide`, `fold` and `find`. This part has not yet been verified but we do not anticipate any difficulty.

10.8 Benchmarks

In this section, we propose preliminary benchmarks to assess the performance of `Pool` relatively to `Domainslib` and `Moonpool` on simple workloads. We only looked at the standard work-stealing strategy; we leave the extension of the benchmarks to the two other strategies for future work.

Remark: we developed the benchmarks with Gabriel Scherer, who wrote most of this section.

10.8.1 Setting

The benchmarks were produced on a 12-core AMD Ryzen 5 7640U machine, set at a fixed frequency of 2GHz. They were *not* run in an isolated environment, so at least one core was busy with other programs. It is fair to assume that workloads running with more than 11 domains suffered from CPU contention.

Note that the OCaml 5 runtime is known to behave badly under CPU contention (when there are more domains running than CPU cores available to run them) due to the stop-the-world design being sensitive to OS-imposed pauses, so we expect performance to plummet for benchmark runs using 11 or 12 domains. This sensitivity to CPU contention explains why OCaml end-users are strongly encouraged to use a scheduler to spawn many concurrent tasks on top of a fixed number of domains.

10.8.2 Parameters

For each benchmark, we pick an input parameter that gives long-enough computation times on our test machine, typically between 200ms and 2s. We use the `hyperfine` tool and run each benchmark ten time. All benchmark were run with two parameters varying:

- `DOMAINS`, the number of domains used for computation;
- `CUTOFF`, representing an input size or chunk size below which a sequential baseline is used.

For each benchmark, we show:

- per-cutoff results with a fixed value `DOMAINS` = 6, which should be enough to experience scaling issues while not suffering from CPU contention;
- per-domain results with a `CUTOFF` value that is chosen to work well for all implementations for this benchmark.

10.8.3 Scheduler implementations

Each benchmark is written on top of a simple scheduler interface, for which the following implementations are provided:

- `domainslib` uses the `Domainslib` library;

- `parabs` uses the `Parabs` library, including the unverified `for_` routine of Section 10.7;
- `moonpool-fifo` uses the `Moonpool` scheduler in its FIFO configuration;
- `moonpool-ws` uses the `Moonpool` scheduler in its work-stealing configuration;
- `sequential` runs all tasks sequentially on one domain.

10.8.4 Benchmarks

`fibonacci`. The concurrent implementation of Fibonacci of Figure 2.5, extended with a sequential cutoff: below the cutoff value, a sequential implementation is used.

`iota`. This benchmark uses a parallel-for to write a default value in each cell of an array. We expect significant variations due to the `CUTOFF` parameter.

`for_irregular`. This benchmark uses a parallel-for loop with irregular per-element workload: as a first approximation, the i-th iteration computes fibonacci i; this cost grows exponentially in i, so the majority of computation work is concentrated on the largest loop indices. In particular, this parallel-for loop should exhibit poor parallelization on larger `CUTOFF` values, so we expect that `Moonpool` will be at a disadvantage.

`lu`. This benchmark performs the LU factorization of a random matrix of floating-point values. It consists in $O(N)$ repetitions of a parallel-for loop of $O(N)$ iterations, where each iteration performs $O(N)$ sequential work.

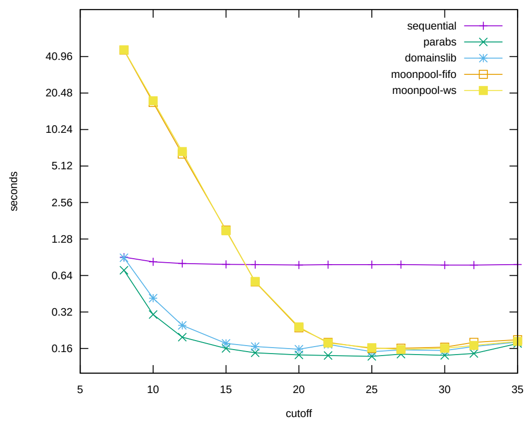
`matmul`. This benchmark computes matrix multiplication with a very simple parallelization strategy — only the outer loop is parallelized. In other word, there is a parallel-for loop with $O(N)$ iterations, where each iteration performs $O(N^2)$ sequential work work.

10.8.5 Overall results

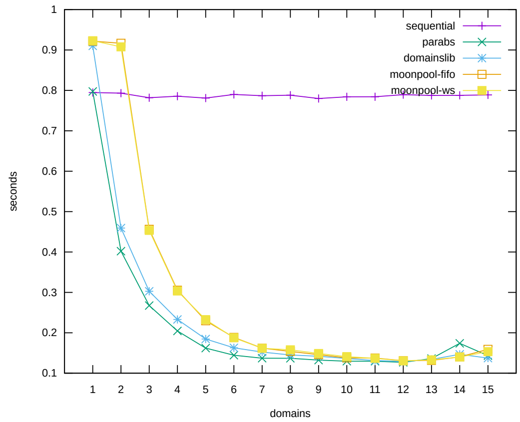
Overall, `Parabs` has the same qualitative performance as `Domainslib`. In fact it performs slightly better (sometimes 10%-20% better) on the benchmarks `fibonacci` and `for_irregular`, which have irregular tasks.

The parallel-for implementation of `Moonpool` does not use the main domain, which has to wait until all iterations have been computed by the worker domains. In contrast `Parabs` and `Domainslib` have the main domain participate by performing tasks until all iterations are complete. This difference in design results in a "one-domain shift" for `Moonpool`, where for example the performance of `Parabs` and `Domainslib` with `DOMAINS` $= 4$ will be comparable to the performance of `Moonpool` with `DOMAINS` $= 5$.

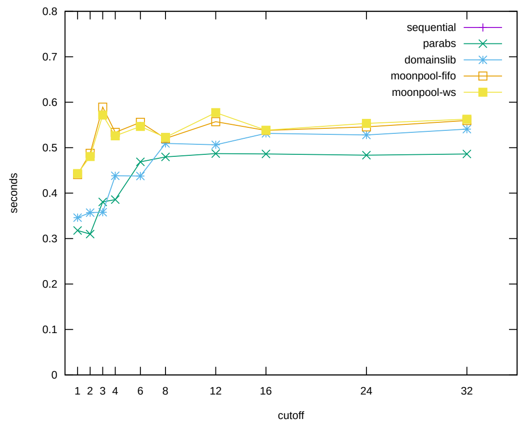
Finally, we observe that the performance of `Parabs` and `Domainslib` suffer noticeably under CPU contention (when `DOMAINS` ≥ 12), whereas the performance of `Moonpool` is surprisingly stable for larger amount of domains. The cause for this difference is not yet understood; one plausible explanation is that the less-efficient, centralized-queue design of `Moonpool` makes domains wait more, and that having several domains waiting on a lock actually helps in CPU-contended situations.



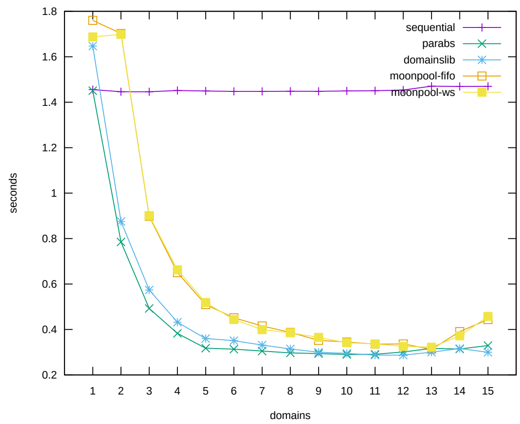
(a) `fibonacci`: varying `CUTOFF`



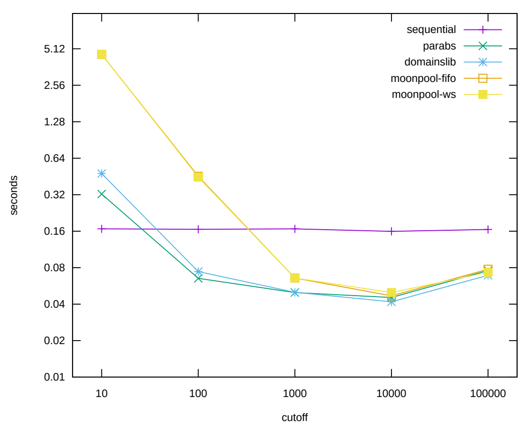
(b) `fibonacci`: varying `DOMAINS`



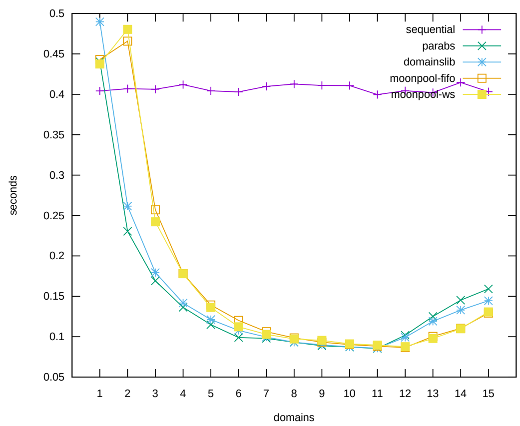
(c) `for_irregular`: varying `CUTOFF`



(d) `for_irregular`: varying `DOMAINS`

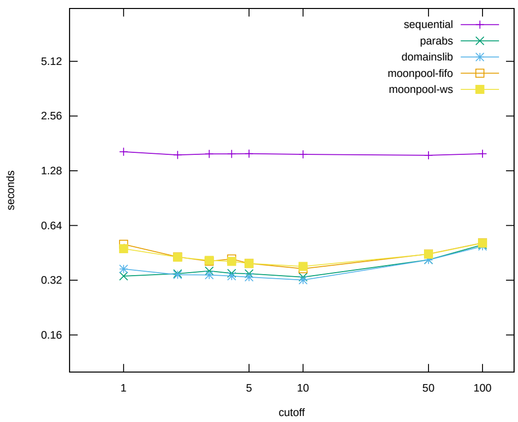


(e) `iota`: varying `CUTOFF`

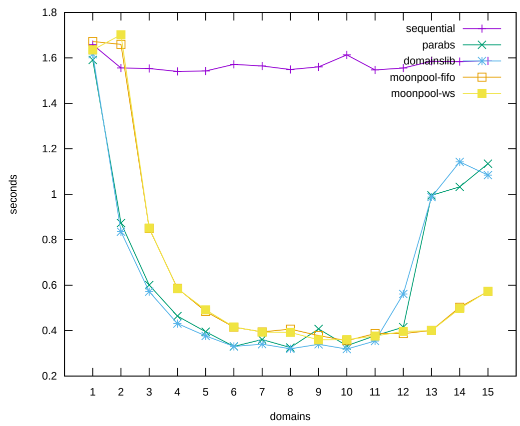


(f) `iota`: varying `DOMAINS`

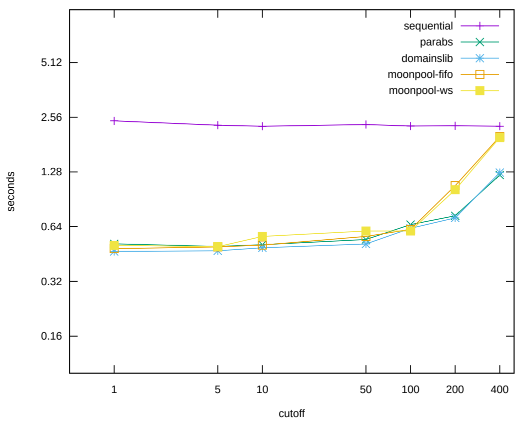
Figure 10.10: Benchmarks (1/2)



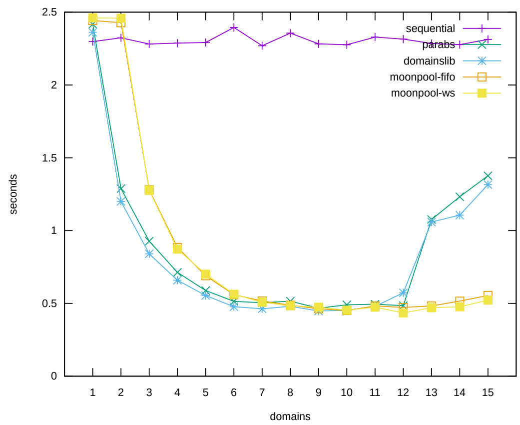
(g) `lu`: varying `CUTOFF`



(h) `lu`: varying `DOMAINS`



(i) `matmul`: varying `CUTOFF`



(j) `matmul`: varying `DOMAINS`

Figure 10.10: Benchmarks (2/2)

10.8.6 Per-benchmark results

The results of the benchmarks are displayed in Figure 10.10.

`fibonacci`. In the per-cutoff results, we see that all schedulers start to behave badly when the `CUTOFF` becomes small enough, with exponentially-decreasing performance after a certain drop point. For `Moonpool`, performance drops around `CUTOFF` = 22. For `Parabs` and `Domainslib`, performance drops around `CUTOFF` = 15. In fact, even for the sequential scheduler we observe a small performance drop: the task-using version creates closures and performs indirect calls, so it is noticeably slower (by a constant factor) than the version used below the sequential cutoff.

Note: we observe very large memory usage with `Moonpool` at smaller cutoff values — when computing fibonacci 40, attempting to run the benchmark with `CUTOFF` = 5 fails with out-of-memory errors on a machine with 32Gio of RAM. This seems to come from the FIFO architecture which runs the oldest and thus biggest task first, and thus stores an exponential number of smaller tasks in the queue.

Per-domain results: `parabs` behaves slightly better than `domainslib` and they both behave better than `moonpool` due to the one-domain shift. Performance becomes very close for larger number of domains (`DOMAINS` ≥ 7).

`for_irregular`. This benchmark is designed to behave poorly with large `CUTOFF` values. We indeed observe better noticeably performance with `CUTOFF` = 1 than with larger values, across all schedulers — for example `domainslib` is 50% slower with `CUTOFF` = 8. The performance benefits of `CUTOFF` = 1 is smaller for `moonpool` due to its higher scheduling overhead: it is only 18% slower with `CUTOFF` = 8.

We observe that `parabs` and `domainslib` perform similarly, and better than `moonpool`.

The per-domain results show that `parabs` and `domainslib` are similar and behave better than `moonpool`, especially (but not only) due to the one-domain shift. For example `domainslib` is only 13% slower than `parabs` with `DOMAINS` = 4, whereas the `DOMAINS` = 5 result for `moonpool` (controlling for the one-domain shift) is 32% slower.

`iota`. Each iteration of parallel-for in `iota` is immediate, so as expected we observe a large sensitivity to the choice of `CUTOFF`, with `parabs` and `domainslib` performing much better than `moonpool` on smaller `CUTOFF` values.

When `DOMAINS` ≤ 8, `parabs` and `domainslib` have similar performance but are faster than `moonpool`; but in fact they behave similarly if we control for the one-domain shift of `moonpool`. When `DOMAINS` ≤ 12, we observe that `moonpool` behaves better under CPU contention.

`lu`. The performance is stable over most choices of `CUTOFF`. The per-domain results are similar across all benchmarks after controlling for the one-domain shift of `Moonpool`. After `DOMAINS` ≥ 12, we see that `Moonpool` behaves much better under CPU contention.

`matmul`. The performance is stable across a wide range of `CUTOFF` values. The parallel-loop performs 500 iterations, so `CUTOFF` values closer to 500 prevent parallelization and bring performance closer to the sequential scheduler.

The per-domain analysis is similar to `lu`: similar on all schedulers (after controlling for the one-domain shift of `Moonpool`), except under CPU contention.

10.9 Related work

To our knowledge, `Parabs` is the first realistic scheduler to be verified in Iris. Previous works cover toy implementations, not suitable for real-world usage; in contrast, our implementation is close to state-of-the-art schedulers and offers comparable performance according to our preliminary experiments.

De Vilhena and Pottier [2021] verify a simple cooperative scheduler based on algebraic effects, which serves as a case study for their Iris-based program logic. This scheduler does not support parallelism; it runs fibers inside a single domain. Their notion of future/promise is rudimentary; it only supports persistent output predicates. However, their work, especially the way they formalize the scheduler's effects, will be of particular interest when introducing algebraic effects into ZooLang and `Parabs`.

Ebner et al. [2025] verify a parallel scheduler with the same interface as `Domainslib`, which also serves as a case-study for their program logic. However, their implementation is extremely simplified: a task list protected by a mutex. Their notion of future/joinable is also somewhat rudimentary.

10.10 Future work

As already mentioned throughout this chapter, there are many avenues for future work.

Language features. `Parabs` suffers from the lack of a number of language features unsupported by ZooLang. If functors were supported, we could make the `Parabs` library completely modular. If exceptions were supported, we could catch and re-raise exceptions in `Pool` and `Vertex`. If algebraic effects were supported, we could get rid of evaluation contexts in `Pool`, implement continuation-stealing, `Pool.yield` and `Vertex.yield` and improve `Pool.wait`.

Extensions. In the future, we would like to extend the library in several directions: (1) develop the interface of futures, similarly to `Moonpool`[13]; (2) support the different task types of `Taskflow`, aiming at a more practical `Vertex` interface.

Other designs. We could experiment other designs. For instance, one of the two designs of `Moonpool` relies on a bounded work-stealing deque combined with a master queue. In the literature, many other scheduling strategies were proposed: continuation-stealing [Schmaus et al., 2021; Williams and Elliott, 2025], steal-half work-stealing [Hendler and Shavit, 2002], split work-stealing [Dinan et al., 2009; Rito and Paulino, 2022; van Dijk and van de Pol, 2014; Custódio et al., 2023; Cartier et al., 2021], idempotent work-stealing [Michael et al., 2009].

[13]https://github.com/c-cube/moonpool/blob/main/src/core/fut.mli

Chapter 11

`Kcas`: Lock-free multi-word compare-and-set

Traditional synchronization mechanisms like mutexes and concurrent queues do not compose and can be challenging to use. *Transactional memory* [Shavit and Touitou, 1995] is an abstraction that offers both a relatively familiar programming model and composability.

The `Kcas` [Karvonen, 2025a] library provides a *software transactional memory* (STM) implementation for OCaml. Thanks to its convenient direct style interface, writing a concurrent transaction is a simple as in Figure 11.1. Essentially, a transaction is a specification for generating a list of compare-and-set (CAS) operations to be committed together atomically.

Under the hood, `Kcas` relies on a state-of-the-art *multi-word compare-and-set* (MCAS) algorithm [Guerraoui et al., 2020], a generalization of CAS: given a set of distinct memory locations and corresponding expected and desired values, MCAS atomically either (1) updates all locations from expected values to desired values and succeeds or (2) observes an unexpected value at some location and fails.

The actual implementation of `Kcas` significantly improves and extends this algorithm. In this chapter, we present a verified implementation of its core; the verification of the full interface is left for future work (see Section 11.5).

11.1 Specification

The specification of MCAS is given in Figure 11.2. The persistent assertion $\mathsf{loc\text{-}inv}\ \ell\ \iota$ represents the knowledge that ℓ is a valid MCAS location. The exclusive assertion $\ell \rightarrowtail v$ represents the ownership of location ℓ and the knowledge that it contains value v. `make` v creates a new location initialized with v. `get` ℓ atomically reads the content of ℓ. `cas` ℓs *befores* *afters* performs an MCAS operation on locations ℓs with expected values *befores* and desired values *afters*; on success, it atomically updates ℓs from *befores* to *afters*; on failure, it must have observed a location whose value was different than expected.

```
let a = Loc.make a in
let b = Loc.make b in
let x = Loc.make x in

let tx ~xt =
  let a = Xt.get ~xt a in       CAS (a, a, a)
  let b = Xt.get ~xt b in       CAS (b, b, b)
  Xt.set ~xt x (b - a)          CAS (x, x, b − a)
in

Xt.commit { tx }
```

Figure 11.1: Transaction example and the corresponding list of CASes

$$\mathsf{persistent}\ (\textsf{loc-inv}\ \ell\ \iota)$$

KCAS-POINTSTO-EXCLUSIVE

$$\frac{\ell \rightarrowtail v_1 \qquad \ell \rightarrowtail v_2}{\mathsf{False}}$$

KCAS-MAKE-SPEC

$$\frac{\dfrac{\mathsf{True}}{\texttt{make}\ v}}{\ell.\ \textsf{loc-inv}\ \ell\ \iota * \ell \rightarrowtail v}$$

KCAS-GET-SPEC

$$\textsf{loc-inv}\ \ell\ \iota$$
$$\frac{\dfrac{v.\ \ell \rightarrowtail v}{\texttt{get}\ \ell\ ⨾\ \iota}}{\ell \rightarrowtail v}$$
$$w.\ v \approx w$$

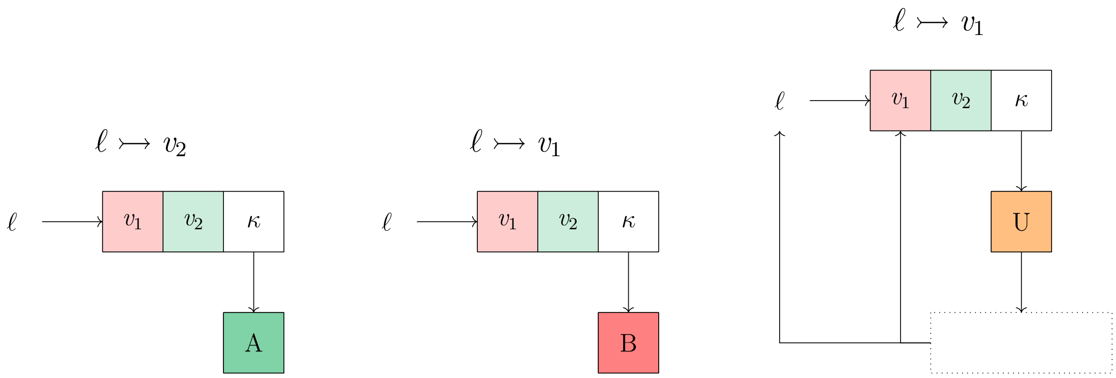
KCAS-CAS-SPEC

$$\mathop{\LARGE\ast}_{\ell \in \ell s} \textsf{loc-inv}\ \ell\ \iota$$

$$\frac{\dfrac{vs.\ \mathop{\LARGE\ast}_{\ell, v \in \ell s, vs} \ell \rightarrowtail v}{\texttt{cas}\ \ell s\ \mathit{befores}\ \mathit{afters}\ ⨾\ \iota}}{\begin{array}{l} b.\ \textbf{if}\ b\ \textbf{then} \\ \quad vs \approx \mathit{befores} * \\ \quad \mathop{\LARGE\ast}_{\ell, v \in \ell s, \mathit{afters}} \ell \rightarrowtail v \\ \textbf{else} \\ \quad \exists\ i\ v\ \mathit{before}. \\ \quad vs\,[i] = \mathsf{Some}\ v * \\ \quad \mathit{befores}\,[i] = \mathsf{Some}\ \mathit{before} * \\ \quad v \not\approx \mathit{before} * \\ \quad \mathop{\LARGE\ast}_{\ell, v \in \ell s, vs} \ell \rightarrowtail v \end{array}}$$

$$\mathit{res}.\ \mathit{res} = b$$

Figure 11.2: Specification



(a) Successful MCAS (b) Failed MCAS (c) Undetermined MCAS

Figure 11.3: Representation of locations

11.2 Implementation

We describe the implementation of the MCAS algorithm of Guerraoui et al. [2020] in OCaml, as proposed[1] by Vesa Karvonen.

Representation of locations. An MCAS location is represented as an atomic reference to a (mutable) *descriptor*, as shown in Figure 11.3. A descriptor contains two values, one of which corresponds to the logical content of the location, and the MCAS operation it belongs to; this MCAS operation is the last to have *locked* the location by writing the descriptor in question. Crucially, each descriptor is unique and belongs to only one MCAS.

An MCAS operation is represented as an atomic reference to a *status*. When the MCAS has finished, its status is either A (after), meaning it succeeded, or B (before), meaning it failed. While the MCAS is still ongoing, its status is U (undetermined). To make the algorithm lock-free, other operations have to be able to help the MCAS finish; therefore, a U status links back to the target locations and the corresponding descriptors, resulting in a cyclic structure. In the algorithm we verified, this structure is a list; in the actual `Kcas` implementation, it is a splay tree.

Locking. Comparatively to previous implementations like that of Harris et al. [2002], this MCAS algorithm is simpler: it includes a locking phase but no unlocking phase — on the other hand, there is an extra indirection for reads.

Figure 11.4 shows a successful execution of an MCAS operation. Initially, the status is U. For each target location, the operation first checks that the current value is as expected and then attempts to atomically install its descriptor (Figures 11.4b and 11.4c), thereby "locking" the location. If the locking phase succeeds, the operation then atomically updates its status to A (Figure 11.4d), which corresponds to the point when locations are logically modified. Additionally, contrary to the original implementation, cleaning up is required to allow stale values to be garbage-collected (Figures 11.4e and 11.4f).

Figure 11.5 shows a failing execution of an MCAS operation. If the operation finds

[1] `https://github.com/ocaml-multicore/kcas/blob/main/doc/gkmz-with-read-only-cmp-ops.md`

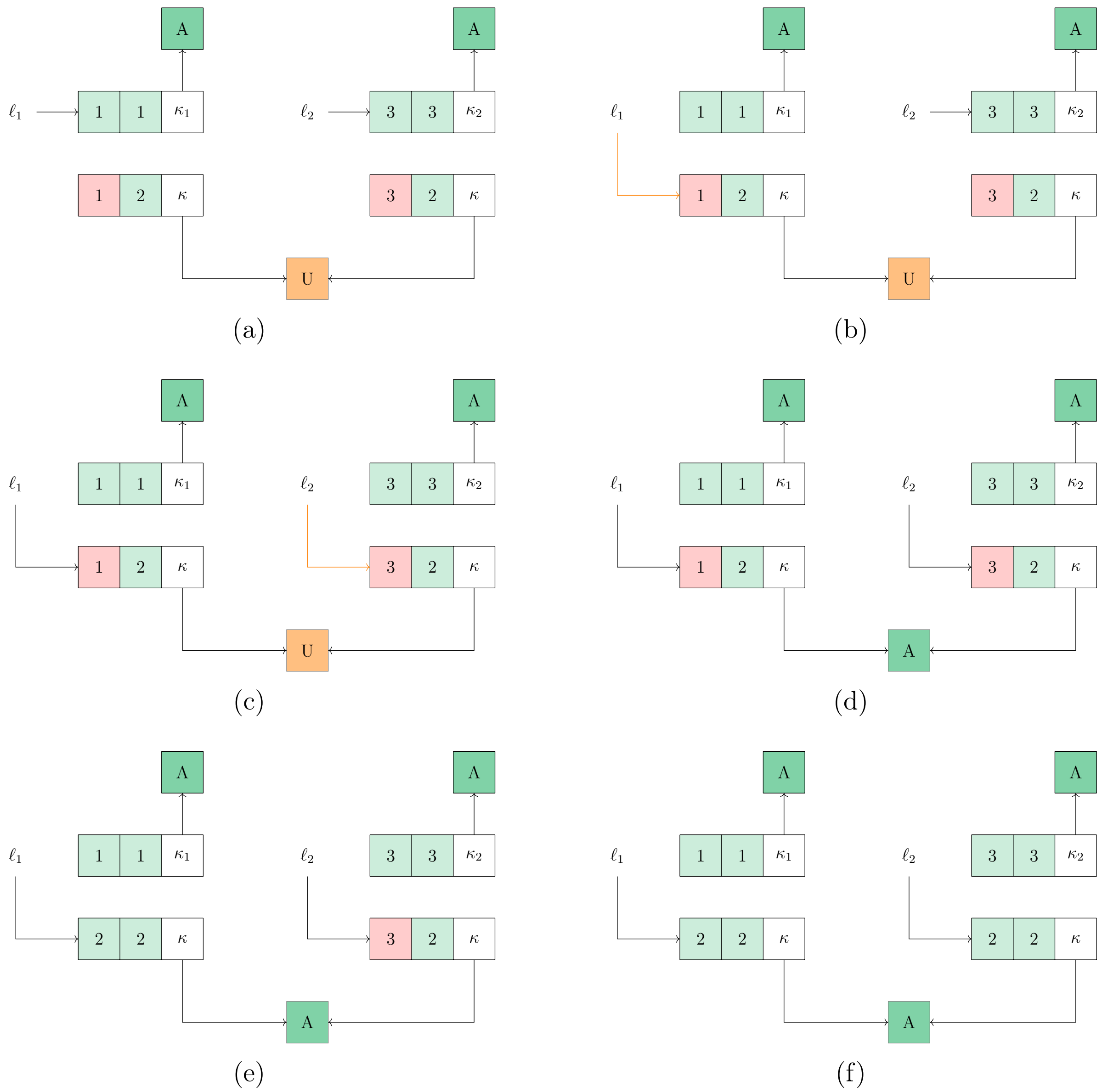


Figure 11.4: Successful MCAS execution

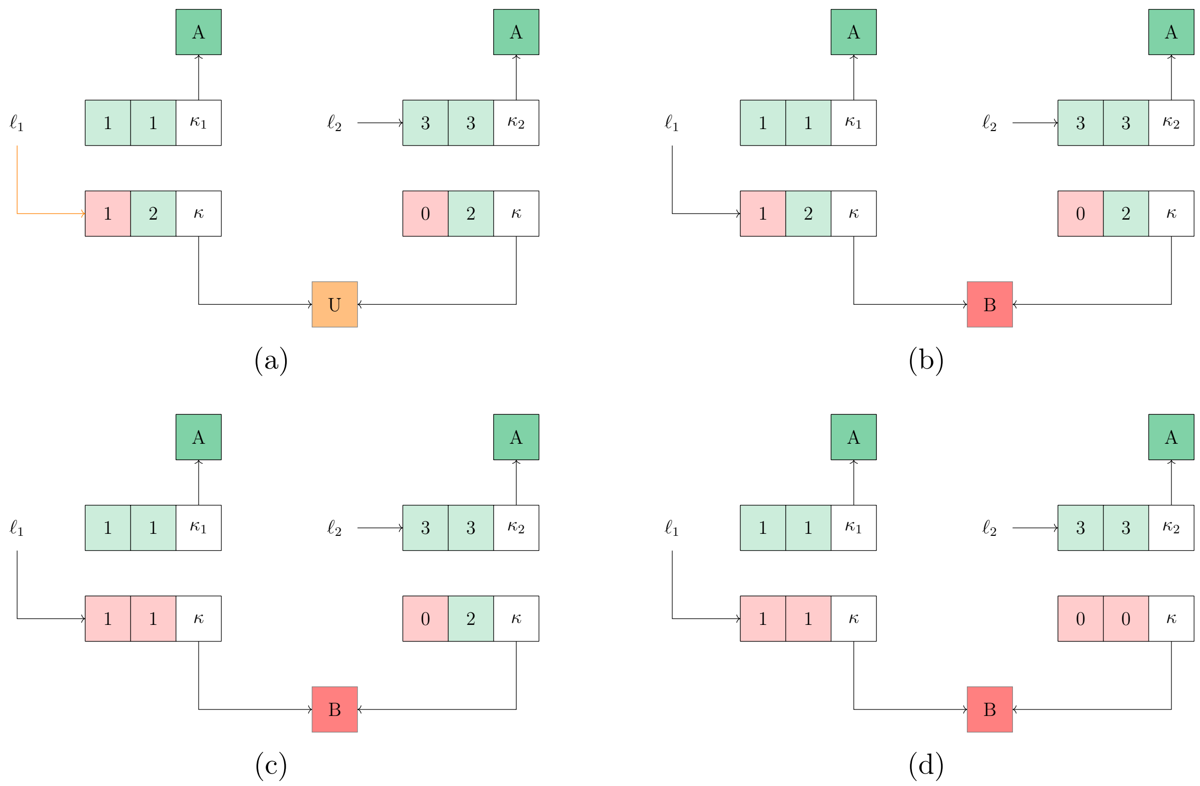


Figure 11.5: Failing MCAS execution

an unexpected value during the locking phase, it atomically updates its status to B (Figure 11.5b) and cleans up stale values (Figures 11.5c and 11.5d).

Helping. So far, we only considered uncontended executions; in reality, one location may be targeted by concurrent `get` and `cas` operations. Basically, everything happens just as before except operations help one another: if an operation encounters a descriptor whose MCAS operation is still in progress, it helps that MCAS operation to finish first.

External linearization. Consequently, the linearization point of both `get` and `cas` may be external. In fact, a helper may linearize many operations at once, including the original MCAS and other helpers. Also, a helper may linearize a helped MCAS while helping another MCAS.

Future-dependent linearization. When two operations helping an MCAS detect an unexpected value, they both attempt to atomically update its status to B. The question is: which of them linearizes the MCAS operation (and other helpers) at the point when it observes an unexpected value? The answer is: the one which "wins" the update. Consequently, the linearization point of `cas` may also be future-dependent.

Liveness. As it is, the algorithm allows MCAS operations to recursively help one another. Obviously, this is problematic in practice. A simple solution consists in sorting the locations beforehand, thereby preventing cycles.

11.3 Proof insights

The implementation is fairly short (one hundred lines of code) but extremely subtle. We outline the main challenges.

Mutually recursive invariants. Due to the cyclic structure of ongoing MCASes (see Figure 11.3c), the definitions of the location invariant (loc-inv) and the MCAS invariant are mutually recursive. Thankfully, Iris provides a way to define such fixpoints.

Another difficulty comes from the fact that the creation of a location involves a dummy MCAS operation whose invariant has to be initialized at the same time as the location's. We had to introduce a way to break the cycle.

Logical state. The verification involves a monotonic logical status that is not determined by the physical status: the physical U status is divided into indexed logical U status indicating the progress made by the MCAS operation.

External linearization. To handle external linearization of an MCAS operation and its helpers by a helper, their atomic updates (see Section 3.8) are stored into the MCAS invariant. At the linearization point, the winning helper triggers all the atomic updates and puts them back into the invariant to be retrieved later by their respective owner.

Global prophecy variable. To handle future-dependent linearization, we predict both the winner and the outcome of an MCAS operation through a shared prophecy variable (see Chapter 5). To distinguish the winner, we use the same trick as Jung et al. [2020] in their proof of RCDSS: each MCAS participant is assigned a unique physical identifier (implemented using a prophecy variable) that is part of the prediction.

Local prophecy variable. For subtle reasons related to the semantics of physical equality (see Section 4.2.3), we also need to introduce a local prophecy variable in `cas`. Essentially, the problem is that we cannot predict the outcome of physical equality in advance just by looking at the values; in other words, equality in Rocq does not imply physical equality in OCaml.

MCAS history. Another subtle point in the algorithm is the fact that a location cannot be locked more than once by a single MCAS operation. While this is obvious when reading the code, it is not in the proof. To enforce it logically, each location maintains a ghost history of distinct MCASes; all MCAS in this history are finished except the more recent one.

11.4 Related work

We believe our work is the first verification of the MCAS algorithm of Guerraoui et al. [2020] and more generally the first foundational verification of an MCAS algorithm.

We are aware of one closely related line of work. Vafeiadis [2008] sketches a proof of the RDCSS and MCAS algorithms of Harris et al. [2002]. However, Jung et al. [2020] show that his proof of RDCSS is flawed and verify it in Iris, using prophecy variables; they also verify both RDCSS and MCAS using VeriFast [Jacobs et al., 2011].

$$
\begin{array}{l}
\textsc{kcas-cmp-cas-spec} \\
\quad \mathop{\Huge\ast}_{k \in ks} \textsf{loc-inv}\ k\ \iota\ * \\
\quad \mathop{\Huge\ast}_{\ell \in \ell s} \textsf{loc-inv}\ \ell\ \iota \\
\hdashline
\langle\, ws\ vs.\ \mathop{\Huge\ast}_{k,v \in ks, ws} k \rightarrowtail v\ * \\
\qquad \mathop{\Huge\ast}_{\ell,v \in \ell s, vs} \ell \rightarrowtail v \,\rangle \\
\hline
\texttt{cas}\ ks\ \ell s\ befores\ afters\ ⨟\ \iota \\
\hline
\langle\, b.\ \mathop{\Huge\ast}_{k,v \in ks, ws} k \rightarrowtail v\ * \\
\quad \textsf{if}\ b\ \textsf{then} \\
\qquad vs \approx befores\ * \\
\qquad \mathop{\Huge\ast}_{\ell,v \in \ell s, afters} \ell \rightarrowtail v \\
\quad \textsf{else} \\
\qquad \mathop{\Huge\ast}_{\ell,v \in \ell s, vs} \ell \rightarrowtail v \,\rangle \\
\hdashline
\quad res.\ res = b
\end{array}
$$

Figure 11.6: Specification with read-only locations

11.5 Future work

Read-only locations. When a location is read during a transaction, the generated MCAS operation (see Figure 11.1) includes a read-only CAS whose expected and desired value are the same; the intended behavior of this CAS is to only assert that the location has not changed after the read. As we have seen in the implementation, however, every location has to be locked for the MCAS to succeed. As a consequence, CAS operations targeting the same location can only execute sequentially, even though they do not change the logical content of the location. This makes read-only CASes inefficient and unscalable.

To address this issue, Vesa Karvonen extended upon the original algorithm [Guerraoui et al., 2020] to allow read-only operations to be expressed directly and not write into memory; the result is a k-CAS-n-CMP algorithm. The idea is the following: at the start, we read the states of read-only locations; at the end, after all other locations have been locked, we check that read-only locations have not changed before finishing the MCAS operation. Crucially, read-only locations are not modified, allowing CMP operations to run in parallel.

There is one drawback, however. This new algorithm is *obstruction-free* but not *lock-free* like the original one. In particular, two MCASes may cancel each other indefinitely. To get the best of both worlds, `Kcas` first attempts the MCAS operations in obstruction-free mode (CAS and CMP) and switches to lock-free mode (CAS only) after a number of failed attempts; the resulting algorithm is lock-free.

From the verification perspective, the new algorithm behaves slightly differently. Indeed, it does not satisfy the former specification (see Figure 11.2): it may happen that `cas` fails although none of the locations was observed to have a different value than expected.

This is due to the fact that the state of a read-only location might have changed without its logical content changing. In practice, such spurious failures are rare and simply cause the MCAS operation to be retried. The new, weaker specification is given in Figure 11.6. We sketched the new invariant and proof on paper, leaving the mechanization for future work.

Relaxed memory model. It would be interesting to carry out the verification in the relaxed memory model. We expect to be able to prove the specification of Figure 11.7. Similarly to primitive atomic locations [Mével et al., 2020], MCAS locations carry a logical memory view. `get` ℓ acquires the view stored in ℓ. On success, `cas` ℓs *befores afters* releases the view of the caller through locations ℓs and acquires their own views.

Transactional interface. In the future, we would also like to extend the verification to the full `Kcas` interface, including transactions. This is challenging for two reasons. (1) The real implementation is even more complex: internally, CAS operations are stored in a splay tree, used as a transaction log, rather than a list; the MCAS algorithm traverses the tree in a depth-first manner. (2) Specifying transactions in a composable way appears to be non-trivial. That begin said, the implementation is relatively short (approximatively one thousand lines of code), so it should be feasible in practice.

$$\mathsf{persistent}\ (\mathsf{loc\text{-}inv}\ \ell\ \iota)$$

KCAS-POINTSTO-EXCLUSIVE

$$\frac{\ell \rightarrowtail (v_1, \mathcal{V}_1) \qquad \ell \rightarrowtail (v_2, \mathcal{V}_2)}{\mathsf{False}}$$

KCAS-MAKE-SPEC-RELAXED

$$\frac{\sqsupseteq \mathcal{V}}{\mathtt{make}\ v} \qquad \ell.\ \mathsf{loc\text{-}inv}\ \ell\ \iota \ast \ell \rightarrowtail (v, \mathcal{V})$$

KCAS-GET-SPEC-RELAXED

$$\mathsf{loc\text{-}inv}\ \ell\ \iota$$

$$v.\ \ell \rightarrowtail (v, \mathcal{V})$$

$$\mathtt{get}\ \ell\ ⨟\ \iota$$

$$\ell \rightarrowtail (v, \mathcal{V})$$

$$w.\ v \approx w \ast \sqsupseteq \mathcal{V}$$

KCAS-CAS-SPEC-RELAXED

$$\sqsupseteq \mathcal{W} \ast \mathop{\Huge\ast}_{\ell \in \ell s} \mathsf{loc\text{-}inv}\ \ell\ \iota$$

$$vs\ \mathcal{V}s.\ \mathop{\Huge\ast}_{\ell, v, \mathcal{V} \in \ell s, vs, \mathcal{V}s} \ell \rightarrowtail (v, \mathcal{V})$$

$$\mathtt{cas}\ \ell s\ \mathit{befores}\ \mathit{afters}\ ⨟\ \iota$$

b. **if** b **then**

$$vs \approx \mathit{befores} \ast \mathop{\Huge\ast}_{\ell, v, \mathcal{V} \in \ell s, \mathit{afters}, \mathcal{V}s} \ell \rightarrowtail (v, \mathcal{V} \sqcup \mathcal{W})$$

else

$$\exists\ i\ v\ \mathit{before}.\ vs\,[i] = \mathsf{Some}\ v \ast \mathit{befores}\,[i] = \mathsf{Some}\ \mathit{before} \ast v \not\approx \mathit{before} \ast \mathop{\Huge\ast}_{\ell, v, \mathcal{V} \in \ell s, vs, \mathcal{V}s} \ell \rightarrowtail (v, \mathcal{V})$$

$$\mathit{res}.\ \mathit{res} = b \ast \textbf{if}\ b\ \textbf{then}\ \mathop{\Huge\ast}_{\mathcal{V} \in \mathcal{V}s} \sqsupseteq \mathcal{V}\ \textbf{else}\ \mathsf{True}$$

Figure 11.7: Specification in relaxed memory

Chapter 12

Memory safety

OCaml is a *memory-safe* language: well typed program cannot go wrong (segfault at runtime). Actually, similarly to Rust, this strong property only holds for programs that do not make use of *unsafe features* of the language, including the `Obj`[1] module and unchecked array accesses (`Array.unsafe_get`, `Array.unsafe_set`).

12.1 Unsafe features

Unsafe features are reserved for expert programmers who want either (1) more expressive types — for instance, the Rocq extraction mechanism —, (2) take advantage of the low-level value representation — for instance, the trick of Section 2.3.2.2 —, or (3) better performance — for instance, performing unsafe array accesses when bound checks are redundant. They should be used with great care, as they may not only break memory safety but also interact in a complex way — this is somewhat of a dark corner of the language — with compiler optimizations, especially Flambda[2], possibly across module boundaries.

Usually, unsafe features are *encapsulated* inside a function or a module: although the implementation relies on unsafe features, the API is safe. Informally, the programmer has to ensure that each unsafe operation is used in a context where it is safe do so; for example, unsafe array accesses require somehow checking bounds. In the case of a module, he can reason on *internal invariants* attached to exposed types and maintained by exposed functions; the abstraction barrier prevents users from breaking these invariants.

12.2 Semantic typing

To formally reconcile memory safety and unsafe features, RustBelt [Jung et al., 2018a] popularized *semantic typing* [Timany et al., 2024]. Concretely, this approach consists in interpreting types as *persistent* Iris predicates. For example, the `bool` and `int` types are interpreted as:

$$[\![\texttt{bool}]\!] \triangleq \lambda\, v.\, \exists\, b \in \mathbb{B}.\, v = b$$

$$[\![\texttt{int}]\!] \triangleq \lambda\, v.\, \exists\, n \in \mathbb{Z}.\, v = n$$

[1] `https://ocaml.org/manual/5.3/api/Obj.html`
[2] `https://ocaml.org/manual/5.3/flambda.html`

More interestingly, product and function types are interpreted as:

$$\llbracket \tau_1 \times \tau_2 \rrbracket \triangleq \lambda\, v.\, \exists\, v_1\; v_2.\, v = (v_1\,, v_2) * \llbracket \tau_1 \rrbracket\; v_1 * \llbracket \tau_2 \rrbracket\; v_2$$
$$\llbracket \tau_1 \to \tau_2 \rrbracket \triangleq \lambda\, v.\, \Box \left(\forall\; w.\, \llbracket \tau_1 \rrbracket\; w \mathrel{-\!\!*} \mathsf{wp}\; v\; w\; \left\{ \llbracket \tau_2 \rrbracket \right\}\right)$$

In words: $\tau_1 \times \tau_2$ represents the set of pairs $(v_1\,, v_2)$ such that v_1 is in τ_1 and v_2 is in τ_2; $\tau_1 \to \tau_2$ represents the set of functions that are safe to call given an argument in τ_1 and whose return value, if any, is in τ_2.

Given these basic semantic types, one may easily show that the (ZooLang translation of the) following function, which reimplements `snd` using the unsafe `Obj` module, is in $\tau_1 \times \tau_2 \to \tau_2$ for any τ_1 and τ_2:

```
let snd (p : 'a * 'b) : 'b =
  Obj.(obj (field (repr p) 1))
```

After type erasure, both `Obj.obj` and `Obj.repr` become the identity. `Obj.field` v i reads the i-th field of the block v; if v is not a block or i is greater than the size of the block, the program crashes. It is safe here because we know from $\tau_1 \times \tau_2$ that `p` is a block with exactly two fields.

Semantic typing allows to give a meaning to the unsafe parts of a program. Crucially, these parts can be linked with the rest, that is the syntactically well-typed parts, thanks to the *fundamental theorem*[3] [Timany et al., 2024]: syntactic typing implies semantic typing. In other words, to prove that a program is memory-safe, it suffices to prove that its unsafe parts are.

Note that the RustBelt semantic types are derived from Rust types, where mutating operations get a mutable borrow and thus exclusive ownership (for a time). In contrast, OCaml semantic types carry no ownership of the values; consequently, they must be robust against concurrent interferences. In particular, mutable types are represented as invariants and all modifications must preserve these invariants atomically. For example, the semantic type corresponding to non-atomic references (`'a ref`) is:

$$\llbracket \texttt{ref}\ \tau \rrbracket \triangleq \lambda\, v.\, \exists\, \ell.\, v = \ell * \boxed{\exists\; w.\, \ell \mapsto w * \llbracket \tau \rrbracket\; w}$$

12.3 Dynarray

Until version 5.2, unlike other languages like C++[4], OCaml did not provide a standard implementation of dynamic arrays. The reason is that, although this data structure is very common, it is actually quite difficult to reach a consensus on the implementation.

To explain why, let us first recall how dynamic arrays are implemented. A dynamic array is represented as a record with two mutable fields: (1) a `size` field storing the length of the dynamic array as perceived by the user and (2) a `data` field storing the *backing array*, a finite array of length at least `size`.

When the user inserts an element to the end of a dynamic array, it typically suffices to (1) write the element at index `size` in the backing array and (2) increment `size`. However, when `size` reaches the actual length of the backing array, called the *capacity*,

[3] We have not formalized this theorem, as it would require formalizing the OCaml type system — a bold enterprise, to say the least.

[4] https://en.cppreference.com/w/cpp/container/vector.html

we need to (1) allocate a new, larger backing array, (2) copy the elements from the old to the new backing array and (3) overwrite `data` with the new backing array.

The backing array may also be shrunk, either by the implementation to reduce memory usage or when the user explicitly asks for it through `fit_capacity`.

One thing is left unsaid in this description: as the backing array is larger than necessary to amortize the resizing, we need to put *some values* in the invalid slots at the end of the backing array. Importantly, these values stay alive and therefore cannot be garbage-collected until they are overwritten through insertions; naturally, this aspect is irrelevant in manually managed languages. They are many ways to handle this, leading to different implementations; we present four of them.

12.3.1 First implementation

A first solution is to ask the user to provide a default value and use this value to fill the invalid part of the backing array. However, this is problematic because (1) the user has to come up with such a value in the first place and (2), as we said, the default value cannot be garbage-collected. For simple types, including immediate types like `int`, this approach is fine. In general, it is not ideal, if not unacceptable.

12.3.2 Second implementation

A second solution is to use something like `Obj.magic ()` as a default value. We verified the functional correctness of this implementation as part of Zoo's standard library. However, while it should be possible to make this implementation memory-safe in OCaml 4 by exploiting the fact that thread switching happens only at certain points, it is *not* memory-safe in OCaml 5. Indeed, incorrect parallel use may lead to the unsafe default value leaking outside the module. In other words, *the introduction of parallelism in OCaml 5 adds more interleavings and therefore breaks the memory safety of some existing code.*

At this point, some OCaml programmers may argue that it is the user responsibility to use this *sequential* data structure correctly — possibly by using a lock to prevent data races — and that careless users do not deserve memory safety. However, the OCaml maintainers consider that *memory safety should be preserved even under incorrect use.* The only exception is when an operation is *explicitly marked as unsafe*, in which case the preconditions should be documented.

12.3.3 Third implementation

A third solution is to introduce an indirection: instead of storing the elements directly in the backing array, we use something like `'a option`, where `'a` is the type of the elements. `Some` is used for the valid slots while `None` is used for the invalid slots.

In 2023, Gabriel Scherer proposed such an implementation[5], based on the following representation:

```
type 'a slot =
  | Empty
  | Element of { mutable value: 'a }
```

[5] https://github.com/ocaml/ocaml/pull/11882

$$
\begin{aligned}
&\llbracket \texttt{element}\ \tau \rrbracket \triangleq \lambda\, elem.\\
&\quad elem \mapsto_{\mathsf{h}} \{\, \mathsf{size}\text{: } 1;\ \mathsf{tag}\text{: } \texttt{Element} \,\} *\\
&\quad \boxed{\exists\, v.\ elem.\mathsf{value} \mapsto v * \llbracket \tau \rrbracket\ v}
\end{aligned}
\qquad
\begin{aligned}
&\llbracket \texttt{t}\ \tau \rrbracket \triangleq \lambda\, t.\\
&\quad \exists\, \ell.\\
&\quad t = \ell *\\
&\quad \boxed{\begin{aligned}
&\exists\, sz\ cap\ data.\\
&0 \leq sz *\\
&\ell.\mathsf{size} \mapsto sz *\\
&\ell.\mathsf{data} \mapsto data *\\
&\llbracket \texttt{array (element}\ \tau\texttt{)}\ cap \rrbracket\ data
\end{aligned}}
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \texttt{slot}\ \tau \rrbracket \triangleq \lambda\, slot.\\
&\bigvee \left[\begin{aligned}
&slot = \S\texttt{Empty}\\
\\
&\exists\, elem.\\
&slot = elem *\\
&\llbracket \texttt{element}\ \tau \rrbracket\ elem
\end{aligned}\right]
\end{aligned}
$$

Figure 12.1: Dynarray: Semantic type

```
type 'a t =
  { mutable size: int;
    mutable data: 'a slot array;
  }
```

Note that, contrary to Some, the Element constructor is mutable, which allows modifying a slot in-place instead of reallocating it.

We verified the functional correctness and memory safety of (a subset of) this implementation as part of Zoo's standard library. Internally, it relies on unsafe array accesses to avoid redundant bound checks — this can significantly improve performance. Consequently, to ensure memory safety even under incorrect parallel use, the implementation adopts a defensive programming style. Consider, for example, the `push` function that inserts an element at the end of a dynamic array. One may naively implement it as follows:

```
let push t slot =
  let sz = t.size in
  if Array.length t.data <= sz then
    reserve t (sz + 1) ;
  t.size <- sz + 1 ;
  Array.unsafe_set t.data sz slot
```

Functional correctness stems from the fact that, after the potential resizing, we know the backing array has enough space so we can write the slot using `Array.unsafe_set`. However, similarly to Section 12.3.2, this implementation is memory-safe in OCaml 4 but is *not* in OCaml 5. Indeed, another domain could mutate the backing array after the size check and before the unsafe write, *e.g.* the `reset` operation which empties the backing array. By contrast, the implementation proposed by Gabriel Scherer is memory safe:

```
let rec push t slot =
  let sz = t.size in
  let data = t.data in
  if Array.length data <= sz then (
    reserve t (sz + 1) ;
    push t slot
```

```
  ) else (
    t.size <- sz + 1 ;
    Array.unsafe_set data sz slot
  )
```

The difference is that, in the infrequent case when resizing is necessary, we restart the operation instead of performing an unsafe write. This is akin to a retry loop in concurrent programming (see Section 2.3.2.1).

To verify memory safety, we use the semantics types of Figure 12.1; for example, `push` is in $[\![$`t` $\tau \rightarrow \tau \rightarrow$ `unit`$]\!]$. Essentially, `t` τ requires (1) the `size` field to be positive and (2) the backing array to contain elements of type `slot` τ. The `array` type carries the size — here, the capacity of the backing array —; this is needed to type unsafe accesses, *e.g.* the semantic type of `Array.unsafe_get` is `array` τ $sz \rightarrow [\![0; sz) \rightarrow$ `unit`. In general, semantic types are richer than syntactic types, they are refinement types [Jhala and Vazou, 2021].

12.3.4 Fourth implementation

In 2024, Gabriel Scherer proposed a new implementation[6]. Compared to the previous one, it involves no indirection. Instead, each dynamic array creates and stores a *unique dummy value* — essentially an untyped memory block — and uses it as a default value. This is similar to the second solution of Section 12.3.2 except the dummy *is guaranteed to be distinct from any user value*. Thanks to this property, the implementation can be made memory-safe by systematically filtering returned elements: checking that they are distinct from the dummy and raising an exception otherwise.

Then, it is crucial for functional correctness that the property be preserved under *correct* use. Indeed, if the user could come up with and insert an element physically equal to the dummy, this element would be filtered by the safety check, thereby raising an unexpected exception.

Informally, the property holds because the dummy *is never leaked outside the module*. A bit more formally, once the dummy is created, *it remains private to the module*, *i.e.* it does not leave the space composed of the dynamic array itself and the operations — either public or private. In fact, one may see the dynamic array as the *only keeper* of the dummy; the operations locally open the dynamic array to reveal the dummy and close it on return.

Even more formally, the idea is that the dynamic array controls the *reachability* of the dummy, as studied by Moine et al. [2023]. The representation predicate of a dynamic array holds an exclusive *pointed-by* assertion attesting that the dummy is currently unreachable directly from the program stack. To access the dummy, the operations pull the pointed-by out of the representation predicate they are given as precondition and use it to locally and temporarily extend the reachability of the dummy. On return, they reduce the reachability — which is possible only if the dummy has not leaked — and put the pointed-by back in the representation predicate. Interestingly, what looked like a global property can be stated as a local property.

Unfortunately, we currently cannot formalize this reasoning in the ZooLang program logic as it does not support the pointed-by assertion. However, we plan to support it in the future. One difficulty is that we cannot just reproduce the formalization of Moine et al.; this would pollute the entire logic. Instead, we need to distinguish two modes: (1)

[6] `https://github.com/ocaml/ocaml/pull/12885`

in *non-tracking* mode, everything works as usual but we cannot use pointed-by assertions; (2) in *tracking* mode, we can use pointed-by assertions to reason about reachability, at the cost of increased proof burden. Finally, we need a way to switch between modes.

12.4 `Saturn`

The question of memory safety also arises in the concurrent data structures of the `Saturn` library (see Chapter 9).

For example, we explained in Section 9.2 that the `Saturn` implementation of the Michael-Scott MPMC queue is careful to erase values stored in the queue to avoid memory leaks. This erasure is performed by writing the unsafe dummy value `Obj.magic ()`. Despite this, the data structure is memory-safe; we proved it using a semantic type which is essentially the concurrent invariant of the queue.

By contrast, more restrictive data structures like the MPSC and SPSC variants of the Michael-Scott queue are *not* memory-safe and should be used cautiously. If the user does not respect the corresponding discipline, *i.e.* only one producer and/or only one consumer, he loses memory safety — he may observe `Obj.magic ()`. From the Iris perspective, the problem is that the token representing the unique producer/consumer is not persistent and therefore cannot be shared except through an invariant, but then it can be accessed only atomically — which is not sufficient.

The `Saturn` authors also provide safe variants of the unsafe data structures. These variants are slower but memory-safe even under unintended concurrent use. Typically, this involves indirections such as using the type `'a option` instead of `'a`, which provides a type-safe `None` value for dummies.

12.5 Future work

The proofs of memory safety that we presented in this chapter were done *manually* in Iris. This approach is very tedious, all the more so as most of the reasoning is fairly simple, mainly involving integer arithmetic — especially in the `Dynarray` case study.

In the future, we would like to automate the process in part: we imagine a tool that would take user-annotated OCaml code as input and automatically check memory safety. In general, we need annotations since type invariants are more precise than syntactic types; for example, the verification of `Dynarray` relies on the invariant that the `size` field is positive.

Chapter 13

Conclusion

Initially, the ambition of this thesis was to build a *verified parallel infrastructure* for OCaml 5. This goal has not been fully attained: much remains to be done to get a practical and full-fledged infrastructure. However, the more modest underlying experiment has been overall successful: developing realistic parallel abstractions backed by verified formal specifications.

During this experiment, we developed the Zoo framework (Chapter 4) whose practicality is demonstrated by various case studies. From the research perspective, Zoo corroborates the idea that Iris-based verification frameworks can scale to real-life programming languages and large pieces of software. Yet again, however, much remains to be done:

Relaxed memory model. As we pointed out in Section 4.4, the main limitation of ZooLang is currently its sequentially consistent memory model, as opposed to the relaxed memory model [Dolan et al., 2018] of OCaml 5. This simplification endangers the soundness of our specifications. Hopefully, transitioning Zoo to relaxed memory should not be very difficult — conceptually, at least — thanks to the work of Mével et al. [2020].

Language subset. ZooLang has been designed from the start for pragmatic verification of advanced concurrent data structures; this informed the choice of feature coverage and the semantics design. To extend `Parabs` (Chapter 10), and more generally to accommodate other uses, more features are needed and therefore should be supported: exceptions, algebraic effects, modules, functors.

Iris proof mode. During the mechanization of our work in the Rocq proof assistant using the Iris proof mode [Krebbers et al., 2018], we faced major bottlenecks, as Park et al. [2025] also recently reported: (1) the overwhelming proof burden, including more or less trivial Iris goals, which can be reduced thanks to Diaframe [Mulder et al., 2022; Mulder and Krebbers, 2023]; (2) the poor performance of Iris interactive proof checking (large proof scripts require minutes to be processed), which is currently unavoidable.

Another minor bottleneck was Iris context management, which becomes fairly overwhelming when repeatedly accessing large invariants. As a quality-of-life improvement, we introduced so-called *custom introduction patterns* in the Iris proof mode (still experimental at the time of writing), that allow introducing (naming and normalizing) hypotheses in a systematic way.

Specification language. In the framework that we presented, Iris specifications live entirely in Rocq. In the future, it would be interesting to provide a *specification language* for OCaml programmers to write formal specifications directly in the source code; these specifications would be also translated by `ocaml2zoo`. A natural first candidate is the Gospel [Charguéraud et al., 2019] specification language. However, our first attempts suggest that this is not the best way to go: Gospel proposes concise specifications in simple cases, but falls short rapidly when it comes to higher-order functions, multiple representation predicates and atomic specifications. Another, more promising way is to start from and adapt the VeriFast [Jacobs et al., 2011] specification language; we are currently experimenting this approach.

Automation. In the future, we also would like to develop automation in two directions. (1) Improve Iris proof automation, mainly by customizing Diaframe. (2) We envision a larger framework coupling *foundational* verification in Rocq (current approach) with *semi-automated* verification similarly to Why3 [Filliâtre and Paskevich, 2013] — which requires a specification language in the first place.

Bibliography

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 219–228. doi:10.1145/2442516.2442538

Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. 2016. Dag-calculus: a calculus for parallel computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 18–32. doi:10.1145/2951913.2951946

Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. 2024. Snapshottable Stores. *Proc. ACM Program. Lang.* 8, ICFP (2024), 338–369. doi:10.1145/3674637

Vytautas Astrauskas, Aurel Bilý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13260)*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer, 88–108. doi:10.1007/978-3-031-06773-0_5

Henry G. Baker. 1978. Shallow Binding in LISP 1.5. *Commun. ACM* 21, 7 (1978), 565–569. doi:10.1145/359545.359566

Henry G. Baker. 1991. Shallow binding makes functional arrays fast. *ACM SIGPLAN Notices* 26, 8 (1991), 145–147. doi:10.1145/122598.122614

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. doi:10.1145/3473586

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel Distributed Comput.* 37, 1 (1996), 55–69. doi:10.1006/JPDC.1996.0107

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999), 720–748. doi:10.1145/324133.324234

Sylvain Boulmé. 2021. *Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)*. Accreditation to supervise research. Université Grenoble-Alpes. https://hal.science/tel-03356701 See also http://www-verimag.imag.fr/ boulme/hdr.html.

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 55–72. doi:10.1007/3-540-44898-5_4

Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. 2014. Implementing and Reasoning About Hash-consed Data Structures in Coq. *J. Autom. Reason.* 53, 3 (2014), 271–304. doi:10.1007/S10817-014-9306-0

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *ACM SIGLOG News* 3, 3 (2016), 47–65. doi:10.1145/2984450.2984457

Romain Calascibetta. 2025. Miou. https://github.com/robur-coop/miou

Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. 2022. Applying formal verification to microkernel IPC at meta. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 116–129. doi:10.1145/3497775.3503681

Pedro Luıs Ribeiro Carrott. 2022. *Formal Specification and Verification of the Lazy JellyFish Skip List*. Master's thesis.

Hannah Cartier, James Dinan, and D. Brian Larkins. 2021. Optimizing Work Stealing Communication with Structured Atomic Operations. In *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, Xian-He Sun, Sameer Shende, Laxmikant V. Kalé, and Yong Chen (Eds.). ACM, 36:1–36:10. doi:10.1145/3472456.3472522

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 243–258. doi:10.1145/3341301.3359632

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 447–463. https://www.usenix.org/conference/osdi22/presentation/chajed

Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 871–886. `https://www.usenix.org/conference/osdi23/presentation/chang`

Arthur Charguéraud. 2010. Program verification through characteristic formulae. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 321–332. doi:`10.1145/1863543.1863590`

Arthur Charguéraud. 2023. *Habilitation thesis: A Modern Eye on Separation Logic for Sequential Programs. (Un nouveau regard sur la Logique de Séparation pour les programmes séquentiels)*. Université de Strasbourg. `https://tel.archives-ouvertes.fr/tel-04076725`

Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenco, and Mário Pereira. 2019. GOSPEL - Providing OCaml with a Formal Specification Language. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11800)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer, 484–501. doi:`10.1007/978-3-030-30942-8_29`

David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, Phillip B. Gibbons and Paul G. Spirakis (Eds.). ACM, 21–28. doi:`10.1145/1073970.1073974`

Jaemin Choi. 2023. Formal Verification of Chase-Lev Deque in Concurrent Separation Logic. *CoRR* abs/2309.03642 (2023). doi:`10.48550/ARXIV.2309.03642` arXiv:2309.03642

Guillaume Claret. 2024. coq-of-ocaml. `https://github.com/formal-land/coq-of-ocaml`

Basile Clément, Camille Noûs, and Gabriel Scherer. 2025. Storable types: free, absorbing, custom. In *36es Journées Francophones des Langages Applicatifs (JFLA 2025)*. Roiffé, France. `https://inria.hal.science/hal-04859464`

Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A persistent union-find data structure. In *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*, Claudio V. Russo and Derek Dreyer (Eds.). ACM, 37–46. doi:`10.1145/1292535.1292541`

Sylvain Conchon and Jean-Christophe Filliâtre. 2008. Semi-persistent Data Structures. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4960)*, Sophia Drossopoulou (Ed.). Springer, 322–336. doi:`10.1007/978-3-540-78739-6_25`

Simon Cruanes. 2025. Moonpool. `https://github.com/c-cube/moonpool`

Rafael Custódio, Hervé Paulino, and Guilherme Rito. 2023. Efficient Synchronization-Light Work Stealing. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2023, Orlando, FL, USA, June 17-19, 2023*, Kunal Agrawal and Julian Shun (Eds.). ACM, 39–49. doi:`10.1145/3558481.3591099`

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 207–231. doi:`10.1007/978-3-662-44202-9_9`

Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 792–808. doi:`10.1145/3519939.3523451`

Paulo Emílio de Vilhena and Francois Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. doi:`10.1145/3434314`

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13478)*, Adrián Riesco and Min Zhang (Eds.). Springer, 90–105. doi:`10.1007/978-3-031-17244-1_6`

Cameron Desrochers. 2025. ConcurrentQueue. `https://github.com/cameron314/concurrentqueue`

James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable work stealing. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. ACM. doi:`10.1145/1654059.1654113`

Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 242–255. doi:`10.1145/3192366.3192421`

Brijesh Dongol and John Derrick. 2014. Verifying linearizability: A comparative survey. *CoRR* abs/1410.6268 (2014). arXiv:1410.6268 `http://arxiv.org/abs/1410.6268`

Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1516–1539. doi:`10.1145/3729311`

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128. doi:10.1007/978-3-642-37036-6_8

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 212–223. doi:10.1145/277650.277725

Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1115–1139. doi:10.1145/3656422

Aïna Linn Georges, Benjamin Peters, Laila Elbeheiry, Leo White, Stephen Dolan, Richard A. Eisenberg, Chris Casinghino, Francois Pottier, and Derek Dreyer. 2025. Data Race Freedom à la Mode. *Proc. ACM Program. Lang.* 9, POPL (2025), 656–686. doi:10.1145/3704859

God. 2022. Personal Communication.

Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. *Proc. ACM Program. Lang.* 7, ICFP (2023), 847–877. doi:10.1145/3607859

Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. 2020. Efficient Multi-Word Compare and Swap. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference (LIPIcs, Vol. 179)*, Hagit Attiya (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:19. doi:10.4230/LIPICS.DISC.2020.4

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings (Lecture Notes in Computer Science, Vol. 2508)*, Dahlia Malkhi (Ed.). Springer, 265–279. doi:10.1007/3-540-36108-1_18

Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Catalin Hritcu, and Bas Spitters. 2024. The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 30–44. doi:10.1145/3636501.3636961

Danny Hendler and Nir Shavit. 2002. Non-blocking steal-half work queues. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, Aleta Ricciardi (Ed.). ACM, 280–289. doi:10.1145/571825.571876

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. doi:10.1145/78969.78972

Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, Nils J. Nilsson (Ed.). William Kaufmann, 235–245. http://ijcai.org/Proceedings/73/Papers/027B.pdf

Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Trans. Parallel Distributed Syst.* 33, 6 (2022), 1303–1320. doi:10.1109/TPDS.2021.3104255

Iris development team. 2025a. The Coq mechanization of Iris. https://gitlab.mpi-sws.org/iris/iris/

Iris development team. 2025b. Iris examples. https://gitlab.mpi-sws.org/iris/examples/

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. doi:10.1007/978-3-642-20398-5_4

Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* 6, 3-4 (2021), 159–317. doi:10.1561/2500000032

Jacques-Henri Jourdan. 2016. *Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié)*. Ph. D. Dissertation. Paris Diderot University, France. https://tel.archives-ouvertes.fr/tel-01327023

Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 828–856. doi:10.1145/3622827

Jaehwang Jung, Sunho Park, Janggun Lee, Jeho Yeon, and Jeehoon Kang. 2025. Verifying General-Purpose RCU for Reclamation in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1–25. doi:10.1145/3729246

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. doi:10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

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 (2020), 45:1–45:32. doi:10.1145/3371113

Vesa Karvonen. 2025a. Kcas. https://github.com/ocaml-multicore/kcas

Vesa Karvonen. 2025b. Multicore-magic. https://github.com/ocaml-multicore/multicore-magic

Vesa Karvonen. 2025c. Picos. https://github.com/ocaml-multicore/picos

Vesa Karvonen and Carine Morel. 2025a. Backoff. https://github.com/ocaml-multicore/backoff

Vesa Karvonen and Carine Morel. 2025b. Saturn. https://github.com/ocaml-multicore/saturn

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. doi:10.1145/3236772

Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. 2020. Verifying concurrent search structure templates. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 181–196. doi:10.1145/3385412.3386029

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. doi:10.1109/TC.1979.1675439

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 286–315. doi:10.1145/3586037

Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 69–80. doi:10.1145/2442516.2442524

Janggun Lee, Jeonghyeon Kim, and Jeehoon Kang. 2025. Leveraging Immutability to Validate Hazard Pointers for Optimistic Traversals. *Proc. ACM Program. Lang.* 9, PLDI (2025), 26–47. doi:10.1145/3729247

Anil Madhavapeddy and Thomas Leonard. 2025. Eio. https://github.com/ocaml-multicore/eio

Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. doi:10.1145/3473571

Glen Mével, Jacques-Henri Jourdan, and Francois Pottier. 2020. Cosmo: a concurrent separation logic for multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 96:1–96:29. doi:10.1145/3408978

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, James E. Burns and Yoram Moses (Eds.). ACM, 267–275. doi:10.1145/248052.248106

Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. 2009. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, Daniel A. Reed and Vivek Sarkar (Eds.). ACM, 45–54. doi:10.1145/1504176.1504186

Alexandre Moine, Arthur Charguéraud, and Francois Pottier. 2022. Specification and verification of a transient stack. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 82–99. doi:10.1145/3497775.3503677

Alexandre Moine, Arthur Charguéraud, and Francois Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 7, POPL (2023), 718–747. doi:10.1145/3571218

Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 103–112. doi:10.1145/2442516.2442527

Ike Mulder and Robbert Krebbers. 2023. Proof Automation for Linearizability in Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 462–491. doi:10.1145/3586043

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 809–824. doi:10.1145/3519939.3523432

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering*, Alexander Pretschner, Doron Peled, and Thomas Hutzelmann (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 50. IOS Press, 104–125. doi:10.3233/978-1-61499-810-5-104

Multicore OCaml development team. 2025. Domainslib. `https://github.com/ocaml-multicore/domainslib`

Duc-Than Nguyen, Lennart Beringer, William Mansky, and Shengyi Wang. 2024. Compositional Verification of Concurrent C Programs with Search Structure Templates. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 60–74. doi:`10.1145/3636501.3636940`

Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary (LIPIcs, Vol. 146)*, Jukka Suomela (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:16. doi:`10.4230/LIPICS.DISC.2019.28`

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. doi:`10.1016/J.TCS.2006.12.035`

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2142)*, Laurent Fribourg (Ed.). Springer, 1–19. doi:`10.1007/3-540-44802-0_1`

Leandro Ostera. 2025. Riot. `https://github.com/riot-ml/riot`

Sunho Park, Jaehwang Jung, Janggun Lee, and Jeehoon Kang. 2025. Verifying Lock-Free Traversals in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 9, PLDI (2025), 48–72. doi:`10.1145/3729248`

Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. 2024. A Proof Recipe for Linearizability in Relaxed Memory Separation Logic. *Proc. ACM Program. Lang.* 8, PLDI (2024), 175–198. doi:`10.1145/3656384`

Nisarg Patel, Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2021. Verifying concurrent multicopy search structures. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32. doi:`10.1145/3485490`

Nisarg Patel, Dennis Shasha, and Thomas Wies. 2024. Verifying Lock-Free Search Structure Templates. In *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria (LIPIcs, Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:28. doi:`10.4230/LIPICS.ECOOP.2024.30`

Mário Pereira and António Ravara. 2021. Cameleer: A Deductive Verification Tool for OCaml. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 677–689. doi:`10.1007/978-3-030-81688-9_31`

Francois Pottier. 2017. Verifying a hash table and its iterators in higher-order separation logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 3–16. doi:10.1145/3018610.3018624

Francois Pottier. 2021. Strong Automated Testing of OCaml Libraries. In *JFLA 2021 - 32es Journées Francophones des Langages Applicatifs*. Saint Médard d'Excideuil, France. https://inria.hal.science/hal-03049511

Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL (2023), 1–32. doi:10.1145/3571194

Pedro Ramalhete. 2016. FAAArrayQueue. https://github.com/pramalhe/ConcurrencyFreaks/blob/5b3b9fcd232ccb5417724fa154e948d0f26b6442/CPP/queues/array/FAAArrayQueue.hpp

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. doi:10.1109/LICS.2002.1029817

Erik Rigtorp. 2025a. MPMCQueue. https://github.com/rigtorp/MPMCQueue

Erik Rigtorp. 2025b. SPSCQueue. https://github.com/rigtorp/SPSCQueue

Guilherme Rito and Hervé Paulino. 2022. Scheduling computations with provably low synchronization overheads. *J. Sched.* 25, 1 (2022), 107–124. doi:10.1007/S10951-021-00706-6

Raed Romanov and Nikita Koval. 2023. The State-of-the-Art LCRQ Concurrent Queue Algorithm Does NOT Require CAS2. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). ACM, 14–26. doi:10.1145/3572848.3577485

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. doi:10.1145/3453483.3454036

Florian Schmaus, Nicolas Pfeiffer, Wolfgang Schröder-Preikschat, Timo Hönig, and Jörg Nolte. 2021. Nowa: A Wait-Free Continuation-Stealing Concurrency Platform. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*. IEEE, 360–371. doi:10.1109/IPDPS49936.2021.00044

Remy Seassau, Irene Yoon, Jean-Marie Madiot, and Francois Pottier. 2025. Formal Semantics and Program Logics for a Fragment of OCaml. *Proc. ACM Program. Lang.* 9, ICFP, Article 240 (Aug. 2025), 32 pages. doi:10.1145/3747509

Daniel Selsam, Simon Hudon, and Leonardo de Moura. 2020. Sealing pointer-based optimizations behind pure functions. *Proc. ACM Program. Lang.* 4, ICFP (2020), 115:1–115:20. doi:10.1145/3408997

Roshan Sharma. 2021. Formal Verification of Concurrent Binary Search Tree. (4 2021). doi:10.25417/uic.15261939.v1

Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, James H. Anderson (Ed.). ACM, 204–213. doi:10.1145/224964.224987

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30. doi:10.1145/3408995

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221. doi:10.1145/3453483.3454039

Thomas Somers and Robbert Krebbers. 2024. Verified Lock-Free Session Channels with Linking. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 588–617. doi:10.1145/3689732

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 14–27. doi:10.1145/3167092

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. doi:10.1145/3547631

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2013. Secure distributed programming with value-dependent types. *J. Funct. Program.* 23, 4 (2013), 402–451. doi:10.1017/S0956796813000142

Amin Timany and Lars Birkedal. 2021. Reasoning about monotonicity in separation logic. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 91–104. doi:10.1145/3437992.3439931

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. doi:10.1145/3676954

R. K. Treiber. 1986. *Systems Programming: Coping with Parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center. `https://books.google.fr/books?id=YQg3HAAACAAJ`

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification.* Ph. D. Dissertation. University of Cambridge, UK. `https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221`

Tom van Dijk and Jaco C. van de Pol. 2014. Lace: Non-blocking Split Deque for Work-Stealing. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II (Lecture Notes in Computer Science, Vol. 8806)*, Luís M. B. Lopes, Julius Zilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero, Jens Breitbart, and Michael Alexander (Eds.). Springer, 206–217. doi:`10.1007/978-3-319-14313-2_18`

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue (proof pearl). In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 76–90. doi:`10.1145/3437992.3439930`

Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 100–115. doi:`10.1145/3497775.3503689`

Conor John Williams and James Elliott. 2025. Libfork: Portable Continuation-Stealing With Stackless Coroutines. *IEEE Trans. Parallel Distributed Syst.* 36, 5 (2025), 877–888. doi:`10.1109/TPDS.2025.3543442`

Chaoran Yang and John M. Mellor-Crummey. 2016. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, 16:1–16:13. doi:`10.1145/2851141.2851168`