Verified Persistent Catenable Deques

Juliette Ponsonnet[1] and François Pottier[2]

[1]ENS Lyon
[2]Inria Paris

The simple persistent catenable deques invented by Kaplan, Okasaki, and Tarjan (2000) support insertion and extraction at either end and concatenation. They have mutable internal state and rely on a restricted form of mutation; yet they are persistent, that is, they appear to be immutable. Using Iris, we verify that they are correct in sequential and concurrent usage scenarios. Using Iris with time credits, we verify that, provided concurrent accesses are forbidden, every operation has amortized time complexity $O(1)$. This requires repairing a certain mysterious condition in Kaplan, Okasaki, and Tarjan's description.

1 Introduction

The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous.
— *John Hughes*

In this paper, we verify the correctness and the worst-case amortized time complexity of the *simple persistent catenable deques* invented by Kaplan, Okasaki, and Tarjan [KOT00]. This data structure stores a sequence of elements. It supports inserting and extracting an element at its left end and at its right end, hence the name *deque*, short for *double-ended queue*. Furthermore, it supports an efficient concatenation operation.

This data structure is *persistent* [DSST89]: that is, apparently immutable. It is not destroyed or invalidated by any of the operations that can be applied to it. Such a data structure is also known as *confluently persistent* or *fully persistent*. We say just *persistent*. This is consistent with the meaning of this word in the program logic Iris [JKJ+18]. There, an assertion P is persistent when $P \vdash \Box P$ holds, that is, when "P implies forever P". Thus, a data structure is persistent exactly if it can be described, in Iris, by a persistent assertion.

This data structure can be considered *purely functional*, provided this terminology is understood in an unusual sense. *Purely functional programming* usually takes one of two forms: *strictly functional programming* forbids the use of mutable state; *lazy functional programming* [Hug89, Oka99] allows the use of *thunks*, a simple mutable data structure that offers a memoization service. In contrast, Kaplan, Okasaki, and Tarjan introduce an unusual third form of purely functional programming. They use references (mutable heap-allocated objects) in a very restricted manner: every time a new value v' is assigned to a reference, overwriting a previous value v, the values v and v' must be equivalent in a certain sense—for example, they must be two concrete representations of the same abstract sequence of elements. This restriction has a rather striking consequence: *whether an assignment is executed or skipped cannot affect the correctness of the code*. That is, if every assignment $x := y$ was replaced with a non-deterministic conditional assignment $\mathsf{if}\ \mathit{flip}()\ \mathsf{then}\ x := y$

then the code would produce the same observable results. Only *the time complexity* of the data structure is influenced by the assignments contained in the code.

We refer to references, used in this restricted way, as *stable references*. A stable reference is equipped with an invariant ϕ, a property of values. When one wishes to write a value v into the reference, one must guarantee that $\phi\ v$ holds. When one reads a value v out of the reference, one can assume that $\phi\ v$ holds, *and nothing more*: one cannot assume that v is the value that one has most recently written. Furthermore, in Separation Logic, $\phi\ v$ does not have to be a *proposition* of type *Prop*, which encodes just knowledge; it can be an *assertion* of type *iProp*, which expresses both knowledge and (exclusive or shared) ownership of certain resources. We exploit this idea.

Kaplan, Okasaki, and Tarjan describe two data structures, namely noncatenable deques and catenable deques. We have implemented both in OCaml (§2) and, using Monolith [Pot21], submitted them to heavy random testing. In this paper, we focus on catenable deques, and refer to them simply as *deques*. Our OCaml code is available online. The verified part represents about 380 lines, excluding blank lines and comments.

Our OCaml code uses references that are intended to be stable. Yet, the OCaml type-checker is not aware of this discipline, and cannot enforce it. Therefore, to verify this code, we turn to Iris. We propose two incomparable Iris APIs for stable references (§3). The first API, which we refer to as the *concurrent stable reference* API, is simpler. It ensures that the invariant property $\phi\ v$, where v is the current content of the reference, holds at all times. Therefore, it allows the reference to be accessed at all times; in particular, it allows concurrent accesses. This API requires ϕ to be persistent. The second API, which we refer to as the *sequential stable reference* API, is slightly more complex. It does not require ϕ to be persistent. It allows the invariant to be broken by a read instruction and later re-established by a write instruction. (Yes, an invariant can be broken by *reading*! cf. §3.2.) While its invariant is broken, a reference must not be read. To enforce this discipline, references are indexed with integer levels, and a ghost token, also indexed with an integer level, serves as an access permission. This has the (necessary) effect of forbidding concurrent accesses. In either API, a stable reference is described by a persistent assertion. As a result, Kaplan, Okasaki, and Tarjan's data structure, which is built out of stable references and immutable memory blocks, is clearly persistent as well. We use Iris, as opposed to a simpler Separation Logic such as CFML [Cha20], because CFML lacks persistent assertions, invariants, ghost state, and support for concurrency.

We manually transcribe our OCaml code into HeapLang, a λ-calculus with mutable state and concurrency, which is part of the implementation of the separation logic Iris [JKJ+18]. Then, for this code, we propose two specifications (§4, §6), and carry out two separate proofs (§5, §7). The first specification and proof are expressed using plain Iris. They guarantee that the data structure is correct under both sequential and concurrent usage scenarios. The proof relies on our concurrent stable reference API. The second specification and proof are expressed using Iris with time credits [MJP19]. They guarantee that, *provided concurrent accesses are forbidden*, every operation has worst-case amortized time complexity $O(1)$. The proof exploits our sequential stable reference API. In the course of this proof, we find that Kaplan, Okasaki, and Tarjan's pseudo-code possibly does *not* have the desired worst-case complexity: at least, there is one case where the proof does not go through. We repair this problem by changing a certain mysterious condition in the code (§2.2, §5.2).

In a concurrent setting, our time complexity analysis breaks, and indeed, in that setting, we believe that one cannot say that the data structure has worst-case amortized complexity $O(1)$. After n operations have taken place, the data structure can be in a high-potential state, where $\Omega(n)$ time credits are virtually stored. In this state, an expensive operation, whose real cost is $\Omega(n)$, may be possible. To achieve apparent cost $O(1)$, this operation needs to spend all of the saved credit. Now, if n threads start performing this expensive operation at the same time then all of them may need to spend the *same* reserve of credits, which is not permitted. Put another way, each thread may find the data structure exactly in the

same state and perform the same expensive work, so the total work may be $\Omega(n^2)$, even though only $2n$ operations have taken place. So, one cannot claim that the worst-case amortized complexity of an operation is $O(1)$. This argument must be taken with some caution, because we have not actually constructed such a scenario, and because there may not be agreement as to the meaning of the concept of "worst-case amortized complexity" in a concurrent setting. What is clear is that if in our formal specification (Figure 10) one erases the tokens $\lightning^\pi$, which forbid concurrent accesses, then we are no longer able to prove that the code satisfies the specification.

Both of our proofs are machine-checked by the Rocq proof assistant. They are available online. They represent about 1000 lines of definitions and statements and 4000 lines of proof scripts, excluding blank lines and comments.

As an auxiliary contribution, we draw attention to a possibly new or little-known way of writing the specification of an iteration function (§5.3). Our style does not involve a loop invariant; instead, it expresses the informal idea that applying *iter* f to a collection c is like applying f to each element of the collection in turn. This specification, as well as several other specifications in this paper, uses nested Separation Logic triples.

2 OCaml Implementation

Like knives, destructive updates can be dangerous when misused,
but tremendously effective when used properly.
— Chris Okasaki

The public API of this data structure appears in Figure 1a. The abstract type α *deque* is parameterized with the type α of the deque's elements. The empty deque, *empty*, is a constant. The functions *push* and *inject* insert an element at the left and right ends of a deque. *pop* and *eject* extract an element out of the left and right ends of a nonempty deque. *concat* concatenates two deques.

Our implementation of deques is a functor: as a parameter, it requires an implementation `B` of *buffers*, which are noncatenable deques of length at most 8. Buffers must provide the operations *empty*, *length*, *push*, *pop*, *inject*, *eject*, *fold_left*, *fold_right*, and a few more, which can be implemented in terms of these. Buffers can be implemented as an algebraic data type or using immutable arrays.

2.1 Type Definitions

The structure of Kaplan, Okasaki, and Tarjan's deques is given by the type definitions in Figure 1b. These definitions can be described as follows. A *deque* is either empty or a nonempty deque. A *nonempty deque* is a reference to a five-tuple. This is a stable reference: whenever it is updated, its old value and its new value represent the same sequence of elements. A *five-tuple* is composed of a prefix buffer, a left child deque, a middle buffer, a right child deque, and a suffix buffer. The child deques do not store ordinary elements; instead, they contain triples, where a *triple* is composed of a buffer, a child deque, and another buffer. This is a typical example of a nested data type [BM98]. In a five-tuple and in a triple, the size of the buffers is subject to constraints that we describe later on (§5).

Our OCaml implementation of deques follows the English presentation given by Kaplan, Okasaki, and Tarjan. We found this presentation easy to understand: we noticed just one place where the text was unclear. We found it to be functionally correct. However, we believe that we found a flaw or a problem in their complexity analysis: there is one path where too many time credits are spent. Fortunately, we are able to repair this problem by changing a condition, which we refer to as "the mysterious condition", in the auxiliary function *pop_triple* (§2.2). Furthermore, to make the code thread-safe, we represent a nonempty deque as a mutable reference to an immutable five-tuple, whereas Kaplan, Okasaki, and

```
type 'a deque
val empty : 'a deque
val push  : 'a -> 'a deque -> 'a deque
val inject: 'a deque -> 'a -> 'a deque
val pop   : 'a deque -> 'a * 'a deque
val eject : 'a deque -> 'a deque * 'a
val concat: 'a deque -> 'a deque -> 'a deque
```

(a) API

```
type 'a deque =
  'a nonempty_deque option
and 'a nonempty_deque =
  'a five_tuple ref
and 'a five_tuple = {
  prefix : 'a buffer;
  left   : 'a triple deque;
  middle : 'a buffer;
  right  : 'a triple deque;
  suffix : 'a buffer;
}
and 'a triple = {
  first  : 'a buffer;
  child  : 'a triple deque;
  last   : 'a buffer;
}
```

(b) Type Definitions

Figure 1. Deques: OCaml API and Type Definitions

Tarjan use a five-tuple whose five fields are mutable. Our approach ensures that updates are atomic: that is, the five fields of a five-tuple are updated at once.

Our code is not small, but is reasonably concise. *push* and *inject*, together with their auxiliary functions, occupy about 100 lines of code. *concat* and its auxiliary functions take up about 50 lines. *pop*, *eject* and their auxiliary functions represent about 230 lines.

2.2 On Pop and its Auxiliary Functions

Together with *eject*, *pop* is the most complex operation on deques. Its implementation involves several auxiliary functions, some of which are shown in Figure 2. Determining exactly what the specification(s) of each function should be, in our proofs of functional correctness and time complexity, was quite challenging.

The main function in Figure 2 is *pop_nonempty*, which pops an element out of a nonempty deque. (Using *pop_nonempty*, it is easy to implement *pop*.) *pop_nonempty* works as follows. By reading the reference r, it obtains a five-tuple f. Then, it tests whether it is safe to use *naive_pop*, a simple function, which (in constant time) extracts an element out of the prefix or suffix buffer of the five-tuple f. If so, *naive_pop* is used. Otherwise, the five-tuple f is transformed by *prepare_pop* into a new five-tuple (also named f in our code) to which *naive_pop* can safely be applied; the reference r is updated with this new five-tuple and *naive_pop* is invoked.

So far, so good. One might believe, at this point, that the condition that is tested by *naive_pop_safe*—namely, in the five-tuple f, either the middle buffer must be empty or the prefix buffer must have more than 3 elements—should be the precondition of *naive_pop*. Indeed it is the case that if the five-tuple f is *safe* (that is, *naive_pop_safe* f returns *true*) then *naive_pop* f produces a valid deque.

However, there is another place where *naive_pop* is called, inside *pop_triple* (Figure 2). There, *naive_pop* is *not* necessarily applied to a safe five-tuple, and in that case, it can produce an invalid deque, that is, one whose invariant is superficially broken. Such a deque can be repaired, Kaplan, Okasaki, and Tarjan explain, by pushing an element into it. Our specifications of *naive_pop* and *push* must also account for this scenario.

Let us say a few words about *prepare_pop*. This function, whose code is omitted in Figure 2, operates as follows. If both child deques of the five-tuple f are empty, then it redistributes some elements among the prefix, middle and suffix buffers of f. Otherwise, it applies *pop_triple* to the leftmost nonempty child deque of f, say d. This yields a triple t.

```
let naive_pop_safe (type a) (f : a five_tuple) : bool =
  let { prefix; middle; _ } = f in
  B.is_empty middle || B.length prefix > 3

let naive_pop (type a) (f : a five_tuple) : a * a deque =
  (* omitted; just 7 lines *)

let inspect_first (type a) (f : a five_tuple) : a =
  (* omitted; just 6 lines *)

let rec pop_nonempty : type a. a nonempty_deque -> a * a deque = fun r ->
  let f = !r in
  if naive_pop_safe f then
    naive_pop f
  else
    let f = prepare_pop f in
    r := f;
    assert (naive_pop_safe f);
    naive_pop f

and prepare_pop : type a. a five_tuple -> a five_tuple = fun f ->
  (* omitted; about 100 lines; uses [pop_triple] *)

and pop_triple : type a. a triple nonempty_deque -> a triple * a triple deque = fun r ->
  let f = !r in
  let t = inspect_first f in
  let { first; last; _ } = t in
  (* The (repaired) mysterious condition: *)
  if not (B.is_empty last) || B.has_length_3 first then
    (* [naive_pop_safe f] is not necessarily true here! *)
    naive_pop f
  else
    pop_nonempty r
```

Figure 2. Pop: Auxiliary Functions

The first buffer of this triple is used to pad the five-tuple f so as to make it safe. The child deque and last buffer of the triple t are then concatenated and pushed back into d, if they are nonempty. This *push* operation repairs the deque d if it was invalid.

The most surprising aspect of the function *pop_triple* is the manner in which a choice between *naive_pop* and *pop_nonempty* is made. It is worth remarking that if one chose to always use *pop_nonempty* then *pop_triple* would be functionally correct. Therefore, Kaplan, Okasaki, and Tarjan's motivation for sometimes using *naive_pop* instead is that *naive_pop* is cheaper than *pop_nonempty*. Furthermore, the condition that governs this choice seems rather mysterious (Figure 2). This condition bears on the triple t, which is the first triple contained in the five-tuple f. As noted earlier, this condition does *not* guarantee that f is safe. Therefore, *naive_pop* f can produce an invalid deque, and so can *pop_triple*. The mysterious condition must ensure that this invalid deque will be repaired, inside *prepare_pop*, by a *push* operation. It is quite nonobvious that this is the case.

In fact, our version of the mysterious condition differs from Kaplan, Okasaki, and Tarjan's. Their condition, which they give in Case 1 in the description of *pop*, is as follows: "either the first nonempty middle buffer in t contains 3 elements or t contains a nonempty deque". In our code, their condition would be: `not (is_empty t.child) || B.has_length_3 first`. We find that, with this condition, the time complexity analysis does not go through: there is a path where one must pay for a call to *pop_nonempty* (inside *pop_triple*) *and* a call to *push* (inside

prepare_pop), yet not enough time credits are available to pay for both calls. By changing the mysterious condition to `not (B.is_empty last) || B.has_length_3 first`, we find that one must pay for *either* a call to *pop_nonempty or* a call to *push*, never both on the same path. This is a key argument in our proof of time complexity.

3 Stable References

Plus ça change et plus c'est la même chose.
— Alphonse Karr

In this section, we review two Iris APIs for stable references, which we refer to as *concurrent stable references* (§3.1) and *sequential stable references* (§3.2). Although these APIs are fairly simple, we believe that it is useful to identify and name the concept of a stable reference. This helps understand the data structures that rely on this concept and simplify their proofs.

Our notation for specifications is standard in the Iris literature. The specification $\{P\}\ e\ (\exists\vec{x})\ v\ \{Q\}$ is a short-hand for the Separation Logic triple $\{P\}\, e\, \{\lambda y.\exists\vec{x}.\ \ulcorner y = v \urcorner * Q\}$. This specification means that (1) in a state where the assertion P holds, the expression e can be safely executed; and (2) if it terminates then (for some values of the variables $\vec{x}$) it must produce the value v and leave the machine in a state where the assertion Q holds. We write once in front of such a specification to indicate that it can be exploited at most once, that is, the expression e can be executed at most once. This is expressed in Iris by a weakest-precondition modality *wp* that is *not* wrapped in a persistence modality.

3.1 Concurrent Stable References

Our concurrent stable reference API appears in Figure 3. The assertion $\ell \Rrightarrow \phi$ means that there exists a stable reference at address ℓ and that the invariant property $\phi : \mathit{Val} \to \mathit{iProp}$ has been associated with this reference. This notation is intentionally reminiscent of the standard points-to assertion $\ell \mapsto v$. However, in a stable points-to assertion, only an invariant property ϕ is known: the current content of the reference is not known.

The property ϕ is fixed when the reference is allocated and cannot be changed thereafter. The initial content of the reference, a value v, must be chosen so that $\phi\ v$ holds. The rule CSRef-Alloc states that, provided $\phi\ v$ holds, a memory allocation expression *ref* v produces a stable reference ℓ, which is described by the assertion $\ell \Rrightarrow \phi$.

The rule CSRef-Persist states that the assertion $\ell \Rrightarrow \phi$ is persistent. This reflects the fact that a stable reference can be shared. This assertion can be shared across multiple threads, thereby allowing a stable reference to be concurrently accessed, for reading and writing, by several threads. This explains why the invariant property ϕ cannot be changed: it is an agreement between all of the parties who have read and write access to this reference.

The predicate ϕ must be persistent. This requirement has to do with reading, and is made explicit in the rule CSRef-Read. Indeed, a read expression $!\ell$ duplicates the value that is stored at address ℓ. The rule states that if ℓ is a stable reference with invariant property ϕ then reading ℓ yields some value v such that $\phi\ v$ holds.

The rule CSRef-Write states that if ℓ is a stable reference with invariant property ϕ and if $\phi\ v$ holds then writing the value v at address ℓ is permitted.

In short, whereas a normal points-to assertion $\ell \mapsto v$ guarantees that the value v is *currently* stored at address ℓ, and guarantees that reading ℓ yields v, when one works with a stable points-to assertion $\ell \Rrightarrow \phi$, one does not know what is the current content of the reference, and one does not care. All that matters is that *reading the reference produces some value that was written to the reference* in the past.

Each of the rules in Figure 3 is a lemma that we are able to prove, based on a concrete definition of the assertion $\ell \Rrightarrow \phi$. This definition is very simple:

$$\ell \Rrightarrow \phi \triangleq \boxed{\exists v.\ (\ell \mapsto v * \phi\ v)}^{\mathcal{N}}$$

CSRef-Persist
$\mathit{persistent}(\ell \Mapsto \phi)$

CSRef-Alloc
$\{\phi\ v\}\ \mathit{ref}\ v\ (\exists \ell)\ \ell\ \{\ell \Mapsto \phi\}$

CSRef-Read
$\mathit{persistent}(\phi) \twoheadrightarrow$
$\{\ell \Mapsto \phi\}\ !\ell\ (\exists v)\ v\ \{\phi\ v\}$

CSRef-Write
$\{\ell \Mapsto \phi\ *\ \phi\ v\}\ \ell := v\ ()\ \{\}$

Figure 3. Concurrent stable references: reasoning rules

This definition involves an Iris invariant, denoted by a square box. The superscript $\mathcal{N}$, a fixed namespace, can be ignored. Inside the invariant, the assertion $\exists v.\ (\ell \mapsto v * \phi\ v)$ means that, at all times, the reference ℓ contains *some* value v such that $\phi\ v$ holds.

Such a combination of an invariant, an existential quantification, and a points-to assertion is not new. As they demonstrate how to interpret the types of a programming language as Iris predicates, Timany et al. [TKDB24] use this combination in the interpretation of ML's type of shared references, the type `ref`. What is somewhat original here is that the property $\phi\ v$ does not have to be a coarse property, such as "v is an integer value" or "v is a deque". It can be a much more precise statement, such as "v is a deque whose elements form the sequence $[v_0; v_1; v_2]$". This choice of ϕ lets us express Kaplan, Okasaki, and Tarjan's idea: a reference described by this property can be updated with *arbitrary* deques that represent the 3-element sequence $[v_0; v_1; v_2]$, and *only* with such deques.

3.2 Sequential Stable References

The stable reference API that we have presented is simple and in some ways very powerful: in particular, it allows concurrent accesses. However, it is also quite restrictive: in particular, it requires the parameter ϕ to be instantiated with a persistent property. Yet, in our analysis of the time complexity of Kaplan, Okasaki, and Tarjan's data structure, we need to instantiate ϕ with an assertion that is *not* persistent. Indeed, with each reference, we need to associate a number of time credits, which represent the *potential* of this reference. A time credit assertion $\$n$ is an affine assertion: it is not persistent. Thus, our concurrent stable reference API cannot explain why each reference can have a potential.

This leads us to look for an alternate API, where no persistence requirement bears on the parameter ϕ. Of course, removing this requirement creates a difficulty in the reasoning rule that governs reading. Suppose that, at a program point pc_1, a read expression $!\ell$ returns a value v. We want its postcondition to contain the assertion $\phi\ v$, so that the continuation of the code can exploit this assertion. In particular, if $\phi\ v$ contains time credits, we may need to *spend* these time credits so as to cover the cost of the code that follows the read expression. In other words, we need to temporarily *violate* the invariant that is associated with the stable reference at address ℓ. We typically expect to restore the invariant at a later program point, say pc_2, where a new value v' is written to address ℓ. At this point, we intend to prove that $\phi\ v'$ holds. This can be done if the time credits that were already at hand at program point pc_1 plus those contained in $\phi\ v$ are enough to cover the credits consumed by the computation that is performed between the program points pc_1 and pc_2 plus those contained in $\phi\ v'$.

In summary, it appears that we need a reasoning rule that allows the invariant of a stable reference ℓ to be broken at a read instruction and later restored at a write instruction. Between these instructions, we say that ℓ is *invalidated*.

Naturally, such a rule must forbid reading the reference while it is invalidated. Therefore, we must abandon the idea that a stable reference is accessible at all times. We must look for an API where reading a reference requires a ghost access token. We must arrange for this token to disappear when the invariant is broken and to re-appear once it is restored.

SSRef-New-Pool
$\Rrightarrow \exists \pi.\, \lightning^{\pi.0}$

SSRef-Persist
$\mathit{persistent}(\ell \overset{\pi.n}{\Longmapsto} \phi)$

SSRef-Alloc
$\{\phi\ v\}\ \mathit{ref}\ v\ (\exists \ell)\ \ell\ \{\ell \overset{\pi.n}{\Longmapsto} \phi\}$

SSRef-Read
$(\forall v.\, \phi\ v \mathrel{-\!\!*} \phi\ v\ *\ \psi\ v) \mathrel{-\!\!*}$
$\{\ell \overset{\pi.n}{\Longmapsto} \phi\ *\ \lightning^{\pi.n}\}\ !\ell\ (\exists v)\ v\ \{\psi\ v\ *\ \lightning^{\pi.n}\}$

SSRef-Read-Write
$\{\ell \overset{\pi.n}{\Longmapsto} \phi\ *\ \lightning^{\pi.n}\}\ !\ell\ (\exists v)\ v\ \{\phi\ v\ *\ \lightning^{\pi.n+1}\ *\ \forall v'.\, \mathsf{once}\ \{\triangleright \phi\ v'\ *\ \lightning^{\pi.n+1}\}\ \ell := v'\ ()\ \{\lightning^{\pi.n}\}\}$

Figure 4. Sequential stable references: reasoning rules

The API that we have just sketched is sound, but it is not expressive enough for our needs. If a single token governs access to all stable references, then as soon as we invalidate *one* stable reference, we lose access to *all* stable references. Yet, in Kaplan, Okasaki, and Tarjan's data structure, between the point where ℓ is read and the point where ℓ is updated, a computation takes place, which needs to access certain stable references. We must have a way of arguing that ℓ is not among them, so as to retain permission to access these references, even though we have lost permission to access ℓ. Fortunately, a *stratification* argument can be made: the stable references that we need to access while ℓ is invalidated are located *at deeper levels* in the data structure. By indexing references and access tokens with integer levels, we can offer an API where access to references at levels greater than n is permitted while a reference at level n is invalidated.

This new API appears in Figure 4. In this API, the stable reference assertion $\ell \overset{\pi.n}{\Longmapsto} \phi$ and the ghost access token $\lightning^{\pi.n}$ are indexed with a *pool* π and with a natural integer level n. The token $\lightning^{\pi.n}$ represents a permission to access all stable references in the pool π at level n and greater levels. A *pool* can be thought of as a region of memory, or a group of references. Pools are a purely static concept: they do not exist at runtime.

The rule SSRef-New-Pool is a ghost update. It creates a fresh pool π and produces an access token for all references in this pool. Initially, there are none.

The rule SSRef-Persist states that stable references are persistent; they can be shared. In contrast, an access token is not persistent: it is affine (that is, unique).

SSRef-Alloc is analogous to CSRef-Alloc. The newly allocated reference can be placed in an arbitrary pool and at an arbitrary level. No access token is required.

SSRef-Read allows reading a stable reference ℓ without invalidating it. This rule differs from CSRef-Read in two ways. First, if the stable reference ℓ inhabits pool π at level n, then the access token $\lightning^{\pi.n}$ is required, and preserved. Second, because ϕ is not assumed to be persistent, and because this rule does not invalidate ℓ, the postcondition cannot contain the assertion $\phi\ v$. Instead, it contains $\psi\ v$, where $\psi\ v$ is a consequence of $\phi\ v$ that can be obtained without consuming $\phi\ v$.

Finally, SSRef-Read-Write invalidates a stable reference between a read $!\ell$ and a write $\ell := v'$. The read expression yields the assertion $\phi\ v$ and transforms the access token $\lightning^{\pi.n}$ into the weaker token $\lightning^{\pi.n+1}$. Bearing in mind that ϕ is not persistent, the fact that $\phi\ v$ is produced implies that this assertion must be stolen from the reference. Therefore, at the program point that follows the read, this reference must be considered invalid. This explains why the access token must be weakened. The weakened token does not allow access to ℓ yet grants access to stable references at deeper levels. The last part of the postcondition of the read expression is the one-shot triple $\forall v'.\, \mathsf{once}\ \{\phi\ v' * \lightning^{\pi.n+1}\}\ \ell := v'\ ()\ \{\lightning^{\pi.n}\}$. This triple can be understood as a permission to update the reference ℓ with an arbitrary new value v' such that $\phi\ v'$ holds. This write consumes the weakened token $\lightning^{\pi.n+1}$ and restores the original token $\lightning^{\pi.n}$. Thus, the program point where this triple is used marks the end of the area where ℓ is invalidated.

$$\mathit{Deque} : \mathit{Val} \to \mathit{list}\ \mathit{Val} \to \mathit{iProp}$$

DEQUE-PERSIST
$$\mathit{persistent}(\mathit{Deque}\ d\ \mathit{xs})$$

DEQUE-EMPTY
$$\mathit{Deque}\ \mathit{empty}\ []$$

DEQUE-PUSH
$$\frac{\{\mathit{Deque}\ d\ \mathit{xs}\}}{(\exists d')\ d'\ \ \{\mathit{Deque}\ d'\ ([x] +\!\!+ \mathit{xs})\}}\ \mathit{push}\ x\ d$$

DEQUE-POP
$$\frac{\{\mathit{Deque}\ d\ ([x] +\!\!+ \mathit{xs})\}}{(\exists d')\ (x, d')\ \ \{\mathit{Deque}\ d'\ \mathit{xs}\}}\ \mathit{pop}\ d$$

DEQUE-CONCAT
$$\frac{\{\mathit{Deque}\ d\ \mathit{xs}\ *\ \mathit{Deque}\ d'\ \mathit{xs}'\}}{(\exists d'')\ d''\ \ \{\mathit{Deque}\ d''\ (\mathit{xs} +\!\!+ \mathit{xs}')\}}\ \mathit{concat}\ d\ d'$$

Figure 5. Specification of Functional Correctness

Each of the rules in Figure 4 is a lemma that we prove, based on concrete definitions of the assertions. We follow the same pattern as earlier (§3.1): the concrete definition of the assertion $\ell \overset{\pi.n}{\Longmapsto} \phi$ involves the assertion $\exists v.\ (\ell \mapsto v * \phi\ v)$, which this time is wrapped in an Iris *non-atomic invariant* [Iri25]. Iris's non-atomic invariant library provides the concept of pool, the access tokens, and the access rules that we need.

4 Functional Correctness: Specification

We now present an Iris specification of deques, which appears in Figure 5. Whereas the OCaml API shown in Figure 1a declares the existence of an abstract *type* of deques, α *deque*, and lists the *types* of the six main operations on deques, this specification declares the existence of an abstract *predicate*, $\mathit{Deque}\ d\ \mathit{xs}$, and provides a precise description of the *behavior* of each operation, in the form of a Separation Logic triple.

The predicate *Deque* has type $\mathit{Val} \to \mathit{list}\ \mathit{Val} \to \mathit{iProp}$, where *Val* is the type of values and *iProp* is the type of Iris assertions. The assertion $\mathit{Deque}\ d\ \mathit{xs}$ means that the value d is a valid deque and that the values stored in this deque form the sequence xs. Thus, this assertion can be understood as a *relation* between the concrete data structure d and the abstract object xs that this data structure is intended to represent.

At the mathematical level, we represent a sequence as a list. We write $[]$ for the empty sequence, $[x]$ for a singleton sequence, and $\mathit{xs} +\!\!+ \mathit{xs}'$ for the concatenation of two sequences. If xss is a list of lists of elements, then we write $\mathit{join}(\mathit{xss})$ for its iterated concatenation, a list of elements.

The rule DEQUE-PERSIST guarantees that $\mathit{Deque}\ d\ \mathit{xs}$ is persistent [JKJ+18, §2.3; §5.3]. Technically, this means that if at a certain point $\mathit{Deque}\ d\ \mathit{xs}$ can be established, then, from this point on, this assertion holds forever. In other words, forever, the deque d will remain valid and will represent the sequence xs. In other words, this rule guarantees that deques are persistent data structures.

DEQUE-EMPTY states that *empty* is a valid deque and represents the empty sequence.

DEQUE-PUSH states that, provided d is valid deque, the function call $\mathit{push}\ x\ d$ is permitted and its result d' is also a valid deque. Furthermore, if the deque d represents the sequence xs, then d' represents the sequence $[x] +\!\!+ \mathit{xs}$. This specification implicitly guarantees that this operation is non-destructive: indeed, since $\mathit{Deque}\ d\ \mathit{xs}$ is a persistent assertion, it still holds after $\mathit{push}\ x\ d$ has returned. It need not be repeated in the postcondition of *push*.

DEQUE-POP is symmetric with DEQUE-PUSH. The function call $\mathit{pop}\ d$ requires the deque d to be nonempty. Naturally, it is possible to test whether a deque is empty and to propose a variant of *pop* that can be applied to a possibly-empty deque.

The specifications of *inject* and *eject* are similar to those of *push* and *pop*. We omit them. *inject* and *eject* are part of the code that we have verified.

DEQUE-CONCAT states that, provided d and d' are valid deques, the function call *concat* d d' is safe, and its result d'' is valid deque. Furthermore, if the deques d and d' represent the sequences xs and xs', then d'' represents the sequence $xs +\!\!+ xs'$. In spite of the separating conjunction that appears in the precondition, the deques d and d' need not be disjoint: they might be the same deque, or they might be two distinct deques that share some subdeques. A conjunction of two *persistent* assertions allows these scenarios.

In summary, this specification is very simple. It is stateless: it involves persistent assertions only. In other words, it is the specification of a purely functional data structure. It does not betray the fact that mutable state is used in the implementation of this data structure.

Because a persistent assertion can be shared between two threads, this specification allows concurrent usage. In other words, it guarantees that this data structure is thread-safe. The intuitive reason why this data structure tolerates concurrent usage, even though it has mutable internal state, is that all references involved are stable: thus, it does not matter in what order two competing operations on a reference take place. In contrast, the time-complexity-aware specification that we present later on (§6) involves affine (non-persistent) access tokens and forbids concurrent accesses.

5 Functional Correctness: Proof

We have proved, using Iris, that our implementation of deques satisfies the specification shown in the previous section (Figure 5). Naturally, this proof cannot be described in detail in this paper. We present our concrete definition of the predicate *Deque* (§5.1), provide formal descriptions of the safe and unsafe usage modes of *naive_pop* (§5.2), and describe an original way of writing specifications for *fold* functions in Separation Logic (§5.3).

5.1 Predicate Definitions

The concrete definition of the assertion *Deque* d xs provides a formal description of deques in Iris. It conveys most of the insights needed to understand this data structure.

Because our code is parameterized over an implementation of buffers, our proof, too, must be parameterized over a predicate *buffer*. We assume that the assertion *buffer* b xs means that b is a buffer whose elements form the sequence xs. We assume that this assertion is persistent: this reflects the idea that buffers are immutable.

The formal definition of the assertion *Deque* d xs appears in Figure 6. The main assertion, *Deque* d xs, is defined in terms of four auxiliary predicates, which describe deques, five-tuples, deques-whose-elements-are-triples, and triples.

Each of these auxiliary predicates is parameterized with an integer depth n. The outermost level of the data structure corresponds to depth 0: therefore, *Deque* d xs is defined as *deque* 0 d xs. Then, as one goes down into the data structure, the depth increases. For example, a five-tuple at depth n contains (pointers to) two deques-of-triples at depth $n + 1$. At this stage, this depth parameter is in fact entirely unnecessary: its presence is not needed for our definitions to be well-formed, and it is not exploited in our proof. We introduce it in anticipation of the next stage, namely, our proof of functional correctness and time complexity of the data structure (§6, §7). There, as noted earlier (§3.2), a stratification of stable references using integer depths is needed.

The definitions of the four auxiliary predicates in Figure 6 are mutually recursive. This is not structural recursion over the parameter n: indeed, this parameter does not decrease along every edge. Instead, this is guarded recursion in the sense of Iris: every cycle must go through a "contractive" connective. This is indeed the case here. One cycle, from *dequeOfTriples* to *triple* and back, goes through a "later" modality, written $\triangleright$. This modality is contractive. A second cycle, from *deque* through *fiveTuple* through *dequeOfTriples* back

$$
\begin{array}{rcl}
\mathit{Deque}\ d\ \mathit{xs} & \triangleq & \mathit{deque}\ 0\ d\ \mathit{xs} \\
\\
\mathit{deque}\ n\ d\ \mathit{xs} & \triangleq & \ulcorner d = \mathit{None}\ \wedge\ \mathit{xs} = []\urcorner\ \vee \\
& & \exists\ \ell.\ \ulcorner d = \mathit{Some}(\ell)\urcorner\ *\ \ell \Mapsto (\lambda \mathit{ft}.\ \mathit{fiveTuple}\ n\ \mathit{ft}\ \mathit{xs}) \\
\\
\mathit{fiveTuple}\ n\ \mathit{ft}\ \mathit{xs} & \triangleq & \exists\ p, l, m, r, s, \mathit{xs}_p, \mathit{xss}_l, \mathit{xs}_m, \mathit{xss}_r, \mathit{xs}_s. \\
& & \ulcorner \mathit{ft} = (p, l, m, r, s)\urcorner\ * \\
& & \ulcorner \mathit{config}_5(|\mathit{xs}_p|, |\mathit{xss}_l|, |\mathit{xs}_m|, |\mathit{xss}_r|, |\mathit{xs}_s|)\urcorner\ * \\
& & \ulcorner \mathit{xs} = \mathit{xs}_p +\!\!+ \mathit{join}(\mathit{xss}_l) +\!\!+ \mathit{xs}_m +\!\!+ \mathit{join}(\mathit{xss}_r) +\!\!+ \mathit{xs}_s\urcorner\ * \\
& & \mathit{buffer}\ p\ \mathit{xs}_p\ * \\
& & \mathit{dequeOfTriples}\ (n+1)\ l\ \mathit{xss}_l\ * \\
& & \mathit{buffer}\ m\ \mathit{xs}_m\ * \\
& & \mathit{dequeOfTriples}\ (n+1)\ r\ \mathit{xss}_r\ * \\
& & \mathit{buffer}\ s\ \mathit{xs}_s \\
\\
\mathit{dequeOfTriples}\ n\ d\ \mathit{xss} & \triangleq & \exists \mathit{ts}.\ \mathit{deque}\ n\ d\ \mathit{ts}\ *\ \mathop{\Large *}_{t,\, \mathit{xs} \in \mathit{ts},\, \mathit{xss}}\ \triangleright\, \mathit{triple}\ n\ t\ \mathit{xs} \\
\\
\mathit{triple}\ n\ t\ \mathit{xs} & \triangleq & \exists\ f, c, l, \mathit{xs}_f, \mathit{xss}_c, \mathit{xs}_l. \\
& & \ulcorner t = (f, c, l)\urcorner * \\
& & \ulcorner \mathit{config}_3(|\mathit{xs}_f|, |\mathit{xss}_c|, |\mathit{xs}_l|)\urcorner\ * \\
& & \ulcorner \mathit{xs} = \mathit{xs}_f +\!\!+ \mathit{join}(\mathit{xss}_c) +\!\!+ \mathit{xs}_l\urcorner\ * \\
& & \mathit{buffer}\ f\ \mathit{xs}_f\ * \\
& & \mathit{dequeOfTriples}\ (n+1)\ c\ \mathit{xss}_c\ * \\
& & \mathit{buffer}\ l\ \mathit{xs}_l
\end{array}
$$

Figure 6. Iris Assertions that Describe a Well-Formed Deque

to *deque*, goes through the "stable reference" connective, which is also contractive. Therefore, this definition is accepted by Iris.

Let us now briefly describe the meaning of the auxiliary assertions in Figure 6.

- *deque* n d *xs* means that d is a deque at depth n that represents the sequence *xs*.
- *fiveTuple* n *ft* *xs* means that *ft* is a five-tuple at depth n that represents *xs*.
- *dequeOfTriples* n d *xss* means that d is a deque of triples at depth n, which represents the sequence of sequences *xss*.[1]
- *triple* n t *xs* means that t is a triple at depth n that represents the sequence *xs*.

In the definition of *deque* n d *xs*, a concurrent stable reference is used: the assertion $\ell \Mapsto (\lambda \mathit{ft}.\, \mathit{fiveTuple}\ n\ \mathit{ft}\ \mathit{xs})$ means that, forever, the mutable reference ℓ must contain a five-tuple that has depth n and that represents the sequence *xs*. It is permitted to write a new five-tuple into this reference, provided it represents the same sequence, namely *xs*.

In the definition of *fiveTuple* n *ft* *xs*, the letters p, l, m, r, s are named after the fields `prefix`, `left`, `middle`, `right`, `suffix` in the code (Figure 1b). Similarly, in the definition of *triple* n t *xs*, the letters f, c, l are named after the fields `first`, `child`, `last`.

In the definition of *dequeOfTriples*, *ts* is a list of triples, and *xss* is the list of the sequences that these triples represent. The iterated separating conjunction requires the lists *ts* and *xss* to have the same length and iterates synchronously over them.

All of the assertions defined in Figure 6 are persistent. This is intuitively obvious: indeed, the definition involves no *non*-persistent assertion, of which a typical example would be

[1] Because one triple represents a sequence of elements, a deque of triples represents a sequence of sequences of elements. One might consider parameterizing *dequeOfTriples* with a sequence *xs*: then its definition would begin with $\exists \mathit{xss}.\ \ulcorner \mathit{xs} = \mathit{join}(\mathit{xss})\urcorner * \ldots$ However, we find it convenient to expose *xss* because this lets us pass its length as an argument to config_3 and config_5.

$$\textsc{Triple-Left-Leaning}\ \frac{c = 0 \qquad f \in \{2,3\} \qquad l \in \{0,2,3\}}{\mathit{config}_3(f,c,l)} \qquad \textsc{Triple-Has-Child}\ \frac{c \neq 0 \qquad f \in \{2,3\} \qquad l \in \{2,3\}}{\mathit{config}_3(f,c,l)}$$

$$\textsc{5Tuple-Suffix-Only}\ \frac{s \in \{1 \ldots 8\}}{\mathit{config}_5(0,0,0,0,s)} \qquad \textsc{5Tuple-Has-Middle}\ \frac{p \in \{3 \ldots 6\} \qquad s \in \{3 \ldots 6\}}{\mathit{config}_5(p,l,2,r,s)}$$

Figure 7. Size Constraints on Buffers in a Well-Formed Deque

a points-to assertion. Instead, the definition of *deque* involves a stable points-to assertion, which is persistent (§3).

The last aspect that remains to be mentioned is the use of the predicates config_3 and config_5. These predicates are defined in Figure 7 following Kaplan et al.'s indications [KOT00]. They impose size constraints on buffers and child deques, as follows:

- In a triple, if the child deque is nonempty, then the two buffers must have size 2 or 3; if the child deque is empty then the last buffer can have size 0.
- In a five-tuple, the middle buffer must have size 0 or 2. If it has size 0 then the first four components of the five-tuple must be empty and the suffix buffer must be nonempty. If it has size 2 then the prefix and suffix buffers must have size between 3 and 6.

5.2 On Naive-Pop

Our earlier informal discussion of *naive_pop* can now be made more precise. We establish two distinct specifications of this function, which appear in Figure 8. They describe the behavior of *naive_pop* when applied to a *safe* five-tuple (§2.2) and to an arbitrary five-tuple, respectively.

The specification NaivePop-Safe states that when it is applied to a *safe* five-tuple that represents the sequence $[x] ++ xs$, *naive_pop* returns a pair (x, d) where d is a *valid* deque that represents the sequence xs. We omit the definition of *safeFiveTuple*.

The specification NaivePop-Unsafe is more unusual and interesting. It states that when *naive_pop* is applied to an *arbitrary* five-tuple that represents the sequence $[x] ++ xs$, it returns a pair (x, d), where d is *not* necessarily a valid deque, but can be turned into one by a *push* operation. Indeed, the postcondition of *naive_pop* in this case is a one-shot triple: $\forall y$, once $\{\}$ *push* y d $(\exists d')$ d' $\{\mathit{deque}\ n\ d'\ ([y] ++ xs)\}$. In other words, about the value d that is returned by *naive_pop*, we know nothing, except that pushing an arbitrary element y into it is safe and will produce a valid deque d', which represents the sequence $[y] ++ xs$. In other words, d is broken in a way that *push* can repair.

In short, the explanation of this little miracle is that the broken deque d returned by *naive_pop* has a specific shape that hits a base case in the function *push*. In this specific case,

$$\textsc{NaivePop-Safe}\ \{\mathit{safeFiveTuple}\ n\ ft\ ([x] ++ xs)\}\ \mathit{naive\_pop}\ ft\ (\exists d)\ (x, d)\ \{\mathit{deque}\ n\ d\ xs\}$$

$$\textsc{NaivePop-Unsafe}\ \{\mathit{fiveTuple}\ n\ ft\ ([x] ++ xs)\}\ \mathit{naive\_pop}\ ft\ (\exists d)\ (x, d)\ \left\{\forall y,\ \begin{array}{c}\text{once } \{\}\\ \mathit{push}\ y\ d\\ (\exists d')\ d'\ \{\mathit{deque}\ n\ d'\ ([y] ++ xs)\}\end{array}\right\}$$

Figure 8. Two Distinct Specifications of *naive_pop*

push pushes an element into the same buffer out of which *naive_pop* just took an element. Therefore this buffer, and the entire deque, are repaired.

We find this use of nested Separation Logic triples a somewhat unusual but particularly pleasant and readable way of conveying this idea. Proving the lemma NaivePop-Unsafe is also particularly pleasant: at the point where *naive_pop* returns the broken deque d, one jumps straight to the proof that *push* will repair this deque. Even though the time when *naive_pop* returns and the time when *push* is called are somewhat far apart, Separation Logic lets us reason as if these two events took place in immediate succession.

This understanding of *naive_pop* suffices to prove that the code is functionally correct. The proof requires checking that if NaivePop-Unsafe is used (that is, if the mysterious condition in *pop_triple* is true) then the broken deque returned by *pop_triple* to *prepare_pop* will be repaired inside *prepare_pop* by a *push* operation.

5.3 A New Specification Style for Fold Functions

In the concatenation of two deques, one must sometimes iterate on a buffer. Two symmetric situations arise: either one iterates from left to right over the buffer, injecting its elements one by one into a deque; or one iterates from right to left over the buffer, pushing its elements one by one into a deque. For this purpose, we have assumed that buffers are equipped with two functions *fold_left* and *fold_right*. We must express the specifications of these functions in Separation Logic. How should this be done?

Because a *fold* function is essentially a loop, it may seem natural to take inspiration from Hoare's reasoning rule for loops. This leads to a specification that begins with a universal quantification over a loop invariant. This loop invariant, a Separation Logic assertion, is typically parameterized with the sequence of the elements produced so far and with the current state. An example appears in a paper by the second author [Pot17, Figure 10].

Yet, the informal specification of the function *fold* in OCaml's `Set` library is written in a different style, which arguably is more accessible to non-experts in program verification. It does not mention a loop invariant. Instead, it states that the function call *fold* f c s (where c is a collection, in this case a set, and s is an initial state) behaves like the sequence of function calls

$$\begin{array}{l} \mathit{let}\ s = f\ x_1\ s\ \mathit{in} \\ \mathit{let}\ s = f\ x_2\ s\ \mathit{in} \\ \ldots \\ \mathit{let}\ s = f\ x_n\ s\ \mathit{in} \\ s \end{array}$$

where $x_1, x_2, \ldots, x_n$ is the sequence of the elements of the set c, listed in increasing order. We remark that this style can be emulated in Separation Logic. The rough idea is as follows: the specification of *fold* should take the form: "if a sequence of calls to f is safe, then a call to *fold* is safe".

Our first step is to define an assertion *safe-fold call* f s xs ϕ which means that a sequence of calls to f is safe. More precisely, this assertion means that, starting with the initial state s, it is safe to apply the function f in succession to each element of the sequence xs, and the final state s' will satisfy $\phi\ s'$. (For now, ignore the parameter *call*.) In other words, this is a generalized weakest-precondition (*wp*) assertion, which concerns a sequence of function calls. The predicate *safe-fold* is inductively defined by the rules Fold-Safe-Nil and Fold-Safe-Cons in Figure 9. In Fold-Safe-Nil, the sequence of elements is empty: the final state is the initial state s, so the postcondition ϕ must be true of s. In Fold-Safe-Cons, the sequence of elements is $x :: xs$, so the function call $f\ s\ x$ must be permitted, and thereafter, it must be safe to fold f on the sequence xs. This is expressed by a nested triple: the assertion *safe-fold call* f s' xs ϕ appears in the postcondition of the triple that allows applying f to s and x.

$$\textsc{Fold-Safe-Nil}\ \frac{\phi\ s}{\mathit{safe\text{-}fold}\ \mathit{call}\ f\ s\ []\ \phi} \qquad \textsc{Fold-Safe-Cons}\ \frac{\mathsf{once}\ \{\}\ \mathit{call}\ f\ s\ x\ (\exists s')\ s'\ \{\mathit{safe\text{-}fold}\ \mathit{call}\ f\ s'\ \mathit{xs}\ \phi\}}{\mathit{safe\text{-}fold}\ \mathit{call}\ f\ s\ (x :: \mathit{xs})\ \phi}$$

$$\textsc{Fold-Left}\ \{\mathit{safe\text{-}fold}\ \mathit{straight}\ f\ s\ \mathit{xs}\ \phi\ *\ \mathit{coll}\ c\ \mathit{xs}\}\ \mathit{fold\_left}\ f\ s\ c\ (\exists s')\ s'\ \{\phi\ s'\ *\ \mathit{coll}\ c\ \mathit{xs}\}$$

$$\textsc{Fold-Right}\ \{\mathit{safe\text{-}fold}\ \mathit{flipped}\ f\ s\ (\mathit{rev}\ \mathit{xs})\ \phi\ *\ \mathit{coll}\ c\ \mathit{xs}\}\ \mathit{fold\_right}\ f\ c\ s\ (\exists s')\ s'\ \{\phi\ s'\ *\ \mathit{coll}\ c\ \mathit{xs}\}$$

Figure 9. Inductive Definition of *safe-fold* and Two Specifications of Fold Functions

Then, the predicate *safe-fold* can be exploited to write down simple specifications for *fold* functions. Two typical specifications appear in the lower half of Figure 9. We assume that the assertion *coll xs c* means that the collection c is valid and represents the sequence of elements xs. Both of these specifications are of the form announced above: "if a sequence of calls to f is safe, then calling *fold* is safe". We find these specifications easy to use on the provider side and on the client side. On the provider side, when verifying a specific *fold* function, one receives a permission to perform a succession of calls to f, and the proof amounts to checking that *fold* performs precisely this sequence of calls, in the correct order and until the end. On the client side, when reasoning about a call to *fold*, one must prove that a sequence of calls to f is permitted, just as if this invocation of *fold* had been magically expanded away and replaced with a sequence of calls to f. This can require inventing a loop invariant and setting up a proof by induction. So, the concept of a client-side loop invariant has not disappeared, but is not visible in our specification of *fold*.

The specifications Fold-Left and Fold-Right in Figure 9 differ in two ways. First, whereas *fold_left* iterates from left to right over the collection, *fold_right* iterates from right to left. This is expressed by passing *xs* versus *rev xs* as an argument to *safe-fold*. Second, by convention in the OCaml world, in *fold_left*, the state s is the first parameter of f, whereas in *fold_right* it is the second parameter. We use the parameter *call*, which can be instantiated with either *straight* or *flipped*, to abstract this away. In short, the meta-level function application *straight f s x* expands to the object-level function application $f\ s\ x$, whereas *flipped f s x* expands to $f\ x\ s$.

6 Time Complexity: Specification

We now show a second Iris specification of deques, which appears in Figure 10. (To save space, the specifications of *inject* and *eject* are omitted.) Whereas the specification of Figure 5 is concerned just with functional correctness, and is expressed and established using plain Iris, this specification is concerned with functional correctness and time complexity, and is expressed and established using Iris with time credits [MJP19]. This variant of Iris is extended with a new assertion, $\$n$, which represents n time credits; furthermore, every function call consumes one time credit. Time credits cannot be duplicated or forged. Thus, the credits that a program fragment needs must be passed to it as part of its precondition. If some credits are in excess, then they can be returned as part of the postcondition. It should be noted that, in a precondition or postcondition, not all credits are plainly visible: some time credits can be hidden inside the definition of an abstract assertion, such as $\mathit{Deque}_{\$}\ \pi\ d\ \mathit{xs}$ in Figure 10. As a result, a program fragment sometimes spends fewer or more credits than its precondition visibly requires. In summary, a Separation Logic triple can be read as a statement of *worst-case amortized time complexity* about a program fragment.

The two specifications that we present have incomparable expressive power. One might think that the earlier one (Figure 5) can be obtained from the new one (Figure 10) by erasing

DEQUE$-PERSIST
$$persistent(Deque_\$\ \pi\ d\ xs)$$

DEQUE$-EMPTY
$$Deque_\$\ \pi\ empty\ []$$

DEQUE$-PUSH
$$\left\{ Deque_\$\ \pi\ d\ xs\ *\ \lightning^{\pi}\ *\ \$7 \right\}$$
$$push\ x\ d$$
$$(\exists d')\ d'\ \left\{ Deque_\$\ \pi\ d'\ ([x]\ +\!\!+\ xs)\ *\ \lightning^{\pi} \right\}$$

DEQUE$-POP
$$\left\{ Deque_\$\ \pi\ d\ ([x]\ +\!\!+\ xs)\ *\ \lightning^{\pi}\ *\ \$171 \right\}$$
$$pop\ d$$
$$(\exists d')\ (x, d')\ \left\{ Deque_\$\ \pi\ d'\ xs\ *\ \lightning^{\pi} \right\}$$

DEQUE$-CONCAT
$$\left\{ Deque_\$\ \pi\ d\ xs\ *\ Deque_\$\ \pi\ d'\ xs'\ *\ \lightning^{\pi}\ *\ \$57 \right\}$$
$$concat\ d\ d'$$
$$(\exists d'')\ d''\ \left\{ Deque_\$\ \pi\ d''\ (xs\ +\!\!+\ xs')\ *\ \lightning^{\pi} \right\}$$

Figure 10. Specification of Functional Correctness and Time Complexity

all time credits. This is not the case: whereas our earlier specification allows concurrent use of deques, our new specification forbids it.

In this new specification, a deque is described by the abstract assertion $Deque_\$\ \pi\ d\ xs$. The parameter π is a pool. It can be thought of as the name of a family of deques. Pools offer a form of alias analysis: two deques that inhabit a common pool may share part of their internal representation, whereas two deques that inhabit distinct pools definitely share nothing. For this reason, pools also serve as the basis of our static concurrency control discipline: while a deque in pool π is being accessed, concurrent accesses to all deques in pool π are forbidden, but concurrent accesses to deques in other pools remain permitted. In Iris, this is expressed by creating a unique access permission, or "token", for each pool π. We re-use the pools and tokens of our sequential stable reference API (§3.2). The rule SSREF-NEW-POOL, a ghost update, creates a new pool and an access token $\lightning^{\pi}$ for this pool. As explained in §3.2, tokens are indexed with integer levels, and $\lightning^{\pi}$ is sugar for $\lightning^{\pi.0}$. We set things up so that the token $\lightning^{\pi.n}$ allows access to all deques at depth n or greater in pool π.

The specifications in Figure 10 are very similar to those in Figure 5. There are two main differences, which are highlighted in color. First, an operation that affects a deque (or several deques) in pool π requires and returns the token $\lightning^{\pi}$. This prevents concurrent accesses to this pool. Second, each operation requires a constant number of time credits: for example, a *push* operation requires 7 time credits; a concatenation operation requires 57 time credits. Naturally, because time credits count function calls, these numbers are rather arbitrary: they reflect the internal organization of our code. Nevertheless, up to a constant factor, they reflect its time complexity. Here, following Kaplan, Okasaki, and Tarjan, we claim that every operation has worst-case amortized time complexity $O(1)$.

7 Time Complexity: Proof

We have proved, using Iris with time credits, that our implementation of deques satisfies the specification shown in the previous section (Figure 10). We discuss the concrete definition of the predicate $Deque_\$$ (§7.1) and the specification of the auxiliary function *pop_triple* (§7.2).

7.1 Predicate Definitions

An excerpt of the definition of $Deque_\$\ \pi\ d\ xs$ appears in Figure 11. It is a simple variant of the definitions shown earlier (§5.1 and Figure 6). The reason why it is well-formed is the same as earlier, namely, contractiveness. The main differences are as follows:

$$
\begin{array}{rcl}
\mathit{Deque}_{\$}\ \pi\ d\ xs & \triangleq & \mathit{deque}_{\$}\ \pi\ 0\ d\ xs \\
\mathit{deque}_{\$}\ \pi\ n\ d\ xs & \triangleq & \ulcorner d = \mathit{None}\ \wedge\ xs = []\urcorner\ \vee \\
 & & \exists\, \ell.\, \ulcorner d = \mathit{Some}(\ell)\urcorner\ *\ \ell \overset{\pi.n}{\Longmapsto} (\lambda ft.\, \mathit{fiveTuple}_{\$}\ \pi\ n\ ft\ xs) \\
\mathit{fiveTuple}_{\$}\ \pi\ n\ ft\ xs & \triangleq & \exists\, p, l, m, r, s, xs_p, xss_l, xs_m, xss_r, xs_s. \\
 & & \$\mathit{potential}(|xs_p|, |xs_s|)\ * \\
 & & \ldots
\end{array}
$$

Figure 11. Iris Assertions that Describe a Well-Formed Deque and its Potential

$$
\begin{array}{lll}
\mathit{potential}(\ \_\ ,\ 8\ ) = 3 & & \text{— } \mathit{red/red} \\
\mathit{potential}(\ 0\ ,\ \_\ ) = 0 & & \text{— } \mathit{green/green} \\
\mathit{potential}((3 \mid 6), (3 \mid 6)) = 3 & & \text{— } \mathit{red/red} \\
\mathit{potential}((3 \mid 6),\ \_\ ) = 1 & & \text{— } \mathit{red/green} \\
\mathit{potential}(\ \_\ , (3 \mid 6)) = 1 & & \text{— } \mathit{green/red} \\
\mathit{potential}(\ \_\ ,\ \_\ ) = 0 & & \text{— } \mathit{green/green}
\end{array}
$$

Figure 12. Potential of a Five-Tuple, based on the Sizes of its Prefix and Suffix Buffers

- Every predicate is parameterized with a pool π.
- In the definition of $\mathit{deque}_{\$}\ \pi\ n\ d\ xs$, instead of a concurrent stable reference (§3.1), a sequential stable reference (§3.2) is used. We place this stable reference in pool π and at level n, where n is the current depth. Thus, the index n, which in Figure 6 was useless, is now exploited.
- In the definition of $\mathit{fiveTuple}_{\$}\ \pi\ n\ ft\ xs$, we add the assertion $\$\mathit{potential}(|xs_p|, |xs_s|)$. This assertion represents a number of time credits, which is computed based on the sizes $|xs_p|$ and $|xs_s|$ of the prefix and suffix buffers of the five-tuple ft. Due to the presence of these time credits, the assertion $\mathit{fiveTuple}_{\$}\ \pi\ n\ ft\ xs$ is *not* persistent. Fortunately, this is acceptable: the reasoning rules for sequential stable references (Figure 4) do not require the property ϕ to be persistent.

Whereas Kaplan, Okasaki, and Tarjan define "the potential of a collection of deques" as the sum of the potentials of the five-tuples that appear in this collection [KOT00, §4.3], in our proof, this step is not needed. We define just the potential of a five-tuple and reason locally about one five-tuple at a time.

Our definition of the potential of a five-tuple follows Kaplan, Okasaki, and Tarjan. Whereas they use a concept of "color", we prefer to give a direct definition based on the sizes p and s of the prefix and suffix buffers. (Let us recall that these sizes are subject to the constraints shown in the lower half of Figure 7.) Our definition of $\mathit{potential}(p, s)$ appears in Figure 12. It is a definition by cases, which must be read in order. The comments on the right-hand side are intended to help see the correspondence with Kaplan *et al.*'s notion of color.

7.2 On Pop-Triple and the Mysterious Condition

While the correctness of Kaplan, Okasaki, and Tarjan's data structure is relatively easy to prove, its time complexity analysis requires more care. This difficulty stems from the mysterious condition in *pop_triple* (Figure 2) and its impact on the verification of *prepare_pop*. Once *pop_triple* has returned a triple t and a possibly invalid deque d', *prepare_pop* extracts some elements out of t and inserts the remaining elements into d' by using *push* and possibly *concat*. Sometimes there are no remaining elements, so no elements are pushed into d'.

PopTriple

$$\{\mathit{dequeOfTriples}_{\$}\ \pi\ n\ d\ ([t] +\!\!+ ts)\ *\ \lightning^{\pi.n}\ *\ \$171\}$$

$$\mathit{pop\_triple}\ d$$

$$(\exists d')\ (t,\ d')\ \left\{ \lightning^{\pi.n}\ *\ \left(\begin{array}{c} \mathit{nonSpecialTriple}\ \pi\ n\ t\ xs\ *\ \mathit{deque}_{\$}\ \pi\ n\ d'\ ts \\ \bigvee \\ \mathit{specialTriple}\ \pi\ n\ t\ xs\ *\ \$(171-4)\ * \\ \forall t',\ \mathsf{once}\ \{\}\ \mathit{push}\ t'\ d'\ (\exists d'')\ d''\ \{\mathit{deque}_{\$}\ \pi\ n\ d''\ ([t'] +\!\!+ ts)\} \end{array} \right) \right\}$$

Figure 13. Specification of *pop_triple*

Let us say that a triple is *special* if it satisfies the mysterious condition, that is, its last buffer is nonempty or its first buffer has size 3. We have explained and proved (§5) that if the triple t is special then *prepare_pop* must push at least one element into d', thereby producing a valid deque, even though d' in this case is not necessarily a valid deque.

Both of our proofs require analyzing *prepare_pop* twice: once in the case where t is special, once in the case where it is not. To this end, we place a disjunction in the postcondition of *pop_triple*, whose specification appears in Figure 13. The precondition requires the argument to be a deque whose first element is the triple t that we wish to extract. It also requires an access token and enough time credits to perform a *pop* operation. In all cases, *pop_triple* returns a pair of the triple t and a value d'. The ghost access token is also always returned. At this point in the postcondition, the disjunction appears.

In the case where t is not special, *pop_triple* has used *pop_nonempty*, which has consumed all of the time credits. In this case, d' is a valid deque. No credits remain, but, fortunately, in this case, *prepare_pop* does not need to call *push* or *concat*.

In the case where t is special, *pop_triple* has used *naive_pop*, so d' is not necessarily valid, but can be repaired by *push*. Furthermore, only 4 credits have been consumed, so many remain, which can be used by *prepare_pop* to pay for its calls to *push* and perhaps *concat*.

In summary, the proof goes through because there is no scenario where both *pop_nonempty* and *push* are called. This is essential, as there is not enough credit to cover both calls. To successfully complete this proof, though, we had to change the mysterious condition. If one uses Kaplan, Okasaki, and Tarjan's condition then there is a scenario where both calls take place. We have confirmed via testing that this scenario is feasible. Our repaired version does not have this problem. We believe that this is what was originally intended.

8 Conclusion

We have presented two machine-checked proofs of Kaplan, Okasaki, and Tarjan's simple catenable deques [KOT00]. One proof establishes functional correctness in sequential and concurrent usage scenarios. The other proof establishes functional correctness and a constant time worst-case amortized time complexity bound in sequential scenarios. The two results are incomparable: one is not a consequence of the other. Nevertheless, the two proofs have a common structure. At present, we have duplicated this common structure; it might be possible to share it, but is unclear whether such an effort would be worthwhile.

Kaplan, Okasaki, and Tarjan offer only a brief sketch of a complexity analysis (Section 4.3). We find this sketch to be correct *provided* the mysterious condition in the function *pop_triple* is repaired (§2.2, §5.2, §7.2).

Kaplan et al. do not remark that their time complexity analysis requires proving the absence of a certain kind of circularity: while the invariant of a reference at depth n is broken, the code still accesses references at depths greater than n, and the proof depends on the fact that the invariants of these deeper references hold. This is made explicit in our

proof by assigning an integer depth to each reference and by using an access permission $\lightning^{\pi.n}$ that grants access to all references at depth n and beyond.

Kaplan et al. do not remark that the data structure can safely be used in a concurrent setting. We prove that it can, provided that one uses a single reference to an immutable 5-tuple, as opposed to a tuple whose five fields are mutable. This is rather remarkable, as the code does not contain any synchronization instructions. However, we note that this invalidates its complexity analysis. In practice, assuming that its performance in sequential code is satisfactory, it might be an interesting candidate for use in possibly-concurrent code, as there is no extra cost to be paid just to ensure safety under concurrent use.

Our proof of safety and functional correctness is carried out under the assumption of a sequentially consistent memory (SC). Yet, because it relies solely on stable references, this data structure should be safe and correct also under much more relaxed memory models. Indeed, all of the values that are ever written to a stable reference satisfy the same property ϕ. So, when the reference is read, it does not matter which (past, present, or future) value is read; all values are safe to use. To port our proof to a variant of Iris that supports a relaxed memory model, it should suffice to prove that our concurrent stable reference API is still valid with respect to this model. The rest of our proof should be unaffected.

Although Separation Logic is renowned for its ability to reason about unique ownership and disjointness of references [ORY01, Rey02], we make rather little use of this ability. Indeed, a stable reference is always shared, never uniquely owned. In our concurrent stable reference API, two references are never known to be disjoint. In our sequential stable reference API, two references are known to be disjoint only if they are indexed with distinct integer levels. In spite of these remarks, we view Iris as a highly suitable tool in this program verification effort. Our proof does involve a uniquely-owned reference in the situation where *naive_pop* produces an invalid deque, which is later repaired by *push* (§2.2). This reference does *not* satisfy the data structure's invariant, so it is important to check that it is used locally and never allowed (by mistake) to masquerade as a stable reference and participate in the data structure. Furthermore, our proof involves several kinds of uniquely-owned ghost permissions. These include time credits, which are permissions to spend one unit of time, and the ghost tokens that serve as access permissions in our sequential stable reference API.

Our paper illustrates several independent uses of nested Separation Logic triples, an idiom which we believe deserves to be better known. In our sequential stable reference API (SSRef-Read-Write), a nested triple expresses the idea that a reference is invalidated by a read and repaired by a later write. In our specification of *naive_pop* (NaivePop-Unsafe), a nested triple expresses the idea that *naive_pop* returns an invalid deque, which nevertheless can be passed as an argument to *push*. In our new specification style for *fold* functions, nested triples are used to define *safe-fold*, which represents a permission to perform a sequence of calls to a function f with a specific sequence of arguments.

At present, our proofs rely on a manual transcription of our OCaml code into HeapLang. In future work, it would be desirable to use an automated translation, such as those offered by Zoo [All26] or Osiris [SYMP25].

We have encountered serious performance problems with the current implementation of Iris on top of Rocq. Iris's tactics can be very slow and can fail to terminate for unknown reasons; sometimes a change causes divergence in a seemingly unrelated part of the proof. Although we have eventually worked around or tolerated these problems, they have made our progress much slower and more painful than expected. We plan to report these problems to the implementors of Iris, in the hope of informing future implementation efforts.

References

[All26] Clément Allain. Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic. *Proceedings of the ACM on Programming Languages*, 10(POPL), January 2026.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In *Mathematics of Program Construction (MPC)*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

[Cha20] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proceedings of the ACM on Programming Languages*, 4(ICFP):116:1–116:34, 2020.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[Hug89] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[Iri25] The Iris 4.4 reference, February 2025. `https://plv.mpi-sws.org/iris/appendix-4.4.pdf`.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[KOT00] Haim Kaplan, Chris Okasaki, and Robert E. Tarjan. Simple confluently persistent catenable lists. *SIAM Journal on Computing*, 30(3):965–977, 2000.

[MJP19] Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in Iris. In *European Symposium on Programming (ESOP)*, volume 11423 of *Lecture Notes in Computer Science*, pages 1–27. Springer, April 2019.

[Oka99] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, September 2001.

[Pot17] François Pottier. Verifying a hash table and its iterators in higher-order separation logic. In *Certified Programs and Proofs (CPP)*, pages 3–16, January 2017.

[Pot21] François Pottier. Strong automated testing of OCaml libraries. In *Journées Françaises des Langages Applicatifs (JFLA)*, February 2021.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

[SYMP25] Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. Formal semantics and program logics for a fragment of OCaml. *Proceedings of the ACM on Programming Languages*, 9(ICFP), August 2025.

[TKDB24] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. A logical approach to type soundness. *Journal of the ACM*, 71(6):40:1–40:75, 2024.