

# Specification and Verification of a Transient Stack

Alexandre Moine  
Inria, Paris  
alexandre.moine@inria.fr

Arthur Charguéraud  
Inria & Université de Strasbourg,  
CNRS, ICube, France  
arthur.chargueraud@inria.fr

François Pottier  
Inria, Paris  
francois.pottier@inria.fr

## Abstract

A *transient data structure* is a package of an ephemeral data structure, a persistent data structure, and fast conversions between them. In this paper, we describe the specification and proof of a transient *stack* and its iterators. This data structure is a scaled-down version of the general-purpose transient *sequence* data structure implemented in the OCaml library Sek. Internally, it relies on fixed-capacity arrays, or *chunks*, which can be shared between several ephemeral and persistent stacks. Dynamic tests are used to determine whether a chunk can be updated in place or must be copied: a chunk can be updated if it is uniquely owned or if the update is monotonic. Using CFML, which implements Separation Logic with Time Credits inside Coq, we verify the functional correctness and the amortized time complexity of this data structure. Our verification effort covers iterators, which involve direct pointers to internal chunks. The specification of iterators describes what the operations on iterators do, how much they cost, and under what circumstances an iterator is invalidated.

## 1 Introduction

The algorithms literature establishes a distinction between ephemeral and persistent data structures [?]. An *ephemeral* data structure allows only destructive updates. This implies that, at a given point in time, only the most recent version of the data structure is available for use; all prior versions are lost. In contrast, a *persistent* data structure supports non-destructive updates, which produce a new version of the data structure, without destroying the previous version. This implies that every version of the data structure remains valid forever and that multiple versions can be used independently of one another.

There are tradeoffs between these two flavors. On the one hand, an operation on an ephemeral data structure is usually faster than its counterpart on a persistent data structure. On the other hand, compared with ephemeral data structures, persistent data structures are generally easier to use and to reason about. They remove the need to worry about sharing,

the danger of unintended updates, and the need for undo operations in backtracking algorithms.

A *transient data structure* aims to combine the benefits of ephemeral and persistent data structures. Transient data structures are popular in the programming language Clojure [2021]. In Clojure, while every data structure is persistent by default, it offers the ability of creating a “transient”, that is, an ephemeral copy of the data structure. The “transient” supports read-only operations, destructive updates, and can be frozen, that is, converted back to a persistent data structure. This is worthwhile because creating, updating, and freezing a “transient” is cheap.

In our terminology, a transient data structure is simply a package of an ephemeral data structure, a persistent data structure, and fast conversions between them.

An example of a useful transient data structure would be a transient *string* data structure. Indeed, both ephemeral and persistent strings are commonly needed. The lack of one of these variants, or the lack of efficient conversions between them, can lead to less elegant and/or less efficient code. A more general example, implemented in OCaml by Charguéraud and Pottier [2021], is Sek. This library offers a transient *sequence* data structure, that is, an abstract data type of mutable sequences, an abstract data type of immutable sequences, and efficient conversions between them. It provides double-ended sequences: elements can be inserted or extracted at either end. It also provides logarithmic-time random access, splitting, and concatenation operations.

Sek’s ephemeral sequences are adapted from the *chunked sequences* data structure [Acar et al. 2014]. At a high-level, sequences are represented as a balanced tree whose leaves consists of *chunks*. Each chunk stores a group of consecutive elements in a fixed-capacity array. The concept of a chunk is almost as old as programming itself: chunks typically appear in the implementation of device drivers; they also appear in the double-ended queues of the C++ STL.

Sek extends Acar et al. [2014]’s work by adding persistent sequences, by introducing sharing between chunks to allow efficient conversions, and by introducing dynamic ownership tests and sharing of *prefixes* of chunks in order to minimize the number of copy-on-write operations. The subtle interaction of these features, as well as the nontrivial amortization arguments required in the complexity analysis of this data structure, provide a strong motivation for formal verification.

In this work, however, we do not verify Sek itself, because that would represent a major undertaking: the library is

---

Authors’ addresses: Alexandre Moine, Inria, Paris, alexandre.moine@inria.fr; Arthur Charguéraud, Inria & Université de Strasbourg, CNRS, ICube, France, arthur.chargueraud@inria.fr; François Pottier, Inria, Paris, francois.pottier@inria.fr.

about 6K lines of OCaml code, not counting blank lines and comments. Instead, we verify a scaled-down variant of Sek. We restrict our attention to *stacks*, that is, single-ended sequences. Insertion and extraction are allowed only at one end; random access, splitting, and concatenation operations are not supported. This allows us to adopt a simple internal representation of stacks as *lists of chunks*, whereas Sek uses bootstrapped trees of chunks [Acar et al. 2014].

This simplification does not remove the challenges of reasoning about shared mutable data, dynamic ownership tests, and dynamic choices between in-place updates and copies. They do remove the difficulty of reasoning about bootstrapped trees; however, this aspect has been addressed already by Charguéraud [2016, §7.5]. A crucial aspect of Sek that is preserved in our scaled-down variant is its support for *iterators* over both ephemeral and persistent data structures.

In summary, in this paper, we formally specify and verify a transient stack data structure, a scaled-down version of Sek. We verify both the functional correctness and the amortized time complexity of this data structure. In this endeavor, we encounter the following challenges:

- We wish to write specifications that describe both the functional behavior and the amortized time complexity of each operation.
- Insofar as possible, we wish to hide the presence of mutable internal state from users of the persistent API.
- The correctness argument requires proving that our runtime ownership tests are correct and that our in-place updates are safe, that is, invisible to users of the persistent API.
- The complexity argument is tied with sharing and ownership considerations. In fact, the complexity of some operations depends on the presence or absence of internal sharing; in such cases, we wish to provide two specifications.
- An iterator maintains direct pointers to certain chunks. We must justify when and why it is safe for the iterator to access these chunks. The specification must indicate that an iterator on a persistent data structure remains valid forever, while an iterator on an ephemeral data structure is invalidated if the data structure is modified, except if it is modified by this iterator's set operation.

We address these challenges in the setting of Separation Logic with Time Credits [Charguéraud and Pottier 2019]. To carry out the proof, we use CFML [Charguéraud 2011], an implementation of this logic inside Coq. Our contributions include:

- A specification, in Separation Logic with Time Credits, of the correctness and amortized time complexity of a transient data structure (a stack) and of its iterators.
- A proof that our implementation of the transient stack satisfies this specification.

- The first verification, to the best of our knowledge, of code involving runtime ownership tests based on unique identifiers.
- A fine-grained treatment of *monotonic state*, allowing us to distinguish operations that definitely do not affect the internal state and operations that may update it in a manner that preserves the *views* of other data structures on the shared state.
- A precise complexity analysis, accounting for the fact that the cost of operations may depend on the presence or absence of internal sharing.

The paper is laid out as follows. We present the API offered by our transient stack, explain how stacks are internally represented, and sketch how the main operations on stacks are implemented (§2). After a refresher on specification of ephemeral stacks in Separation Logic (§3), we present a specification of transient stacks (§4). Then, we give the definitions of the key “representation predicates” that describe our data structure, and highlight several aspects of the proof (§5). We discuss iterators (§6), review the related work (§7), and conclude (§8).

## 2 Transient Stacks

**Interface.** Our code takes the form of an OCaml module, which exposes two abstract data types:

- `estack` for *ephemeral* stacks,
- `pstack` for *persistent* stacks.

Each of these types comes with operations `create`, `push` and `pop`. These operations have constant time complexity.<sup>1</sup> More precisely, some of them have complexity  $O(1)$ , while some have complexity  $O(K)$ , where the parameter  $K$  is the capacity of a chunk. The value of  $K$  is fixed by the user. We provide precise complexity bounds as part of the specification in §4.

Crucially, we provide constant-time conversions between ephemeral and persistent stacks: this is the landmark of a transient data structure. The function `pstack_to_estack` creates a fresh ephemeral stack out of a persistent stack. The chunks that contain the items are not copied up front: they are copied in a lazy manner, as items are popped off the newly-created ephemeral stack. In the reverse direction, the function `estack_to_pstack` freezes an ephemeral stack: that is, it creates a persistent stack out of it. This operation is destructive: the ephemeral stack is invalidated and must no longer be accessed.

The types and operations discussed up to this point appear in Figure 1. In addition, our verified implementation includes `is_empty` and `peek` operations (not discussed in the paper), iterators on both kinds of stacks (§6), and two copy operations for ephemeral stacks (§A.1).

**Internal Representation.** We now explain how transient stacks are represented in memory. (An end user does not

<sup>1</sup>Throughout the paper, every complexity bound is *amortized*.

```
(* Basic operations on ephemeral stacks. *)
type 'a estack
val ecreate : 'a -> 'a estack
val epush   : 'a estack -> 'a -> unit
val epop    : 'a estack -> 'a

(* Basic operations on persistent stacks. *)
type 'a pstack
val pcreate : 'a -> 'a pstack
val ppush   : 'a pstack -> 'a -> 'a pstack
val ppop    : 'a pstack -> 'a pstack * 'a

(* Conversions. *)
val pstack_to_estack : 'a pstack -> 'a estack
val estack_to_pstack : 'a estack -> 'a pstack
```

Figure 1. OCaml API of transient stacks

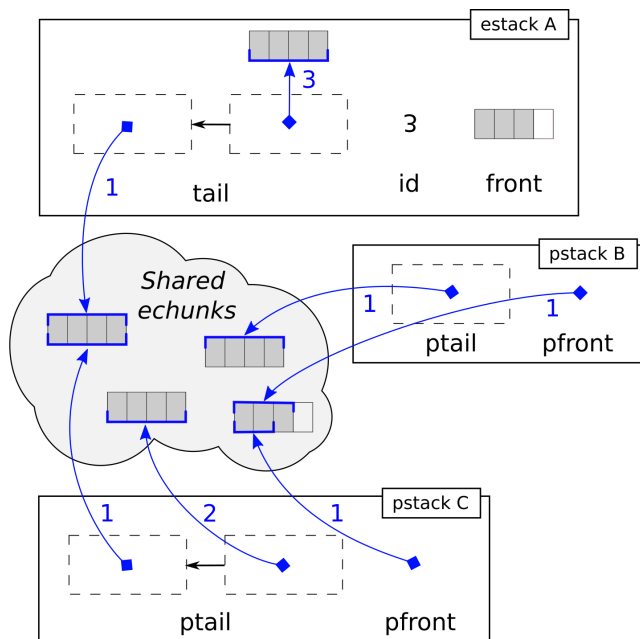


Figure 2. Internal representation of transient stacks

need this information.) Figure 2 presents a drawing where one ephemeral stack (A) and two persistent stacks (B, C) appear. The items contained in a stack are pictured as gray slots. They are stored in fixed-capacity arrays, depicted as groups of 4 slots.

An *ephemeral chunk* is a mutable data structure that holds (1) a pointer to a fixed-capacity array and (2) the number of items currently stored in this array. In practice,  $K$  typically lies between 16 and 256; in Figure 2, it is 4.

An ephemeral chunk is either *uniquely owned* by a specific ephemeral stack, or *shared* between an arbitrary number of ephemeral and persistent stacks. In Figure 2, the shared ephemeral chunks are represented inside the gray cloud.

```
(* Identifiers, used in runtime ownership tests *)
type id = unit ref

(* Ephemeral chunks: data in fixed-capacity arrays *)
type 'a echunk =
  { data: 'a array; mutable top: int; default: 'a }

(* Shareable chunks: views on ephemeral chunks *)
type 'a schunk =
  { support: 'a echunk; view: int; owner: id option }

type 'a estack = (* Ephemeral stacks. *)
  { mutable front: 'a echunk;
    mutable tail : 'a schunk list;
    mutable id   : id;
    mutable spare: 'a echunk option }

type 'a pstack = (* Persistent stack *)
  { pfront: 'a schunk; ptail: 'a schunk list }
```

Figure 3. Internal type definitions for transient stacks

An important point is that a stack does not necessarily “see” the entire content of an ephemeral chunk: its *view* of the chunk may be limited to a *prefix* of the chunk’s elements. These prefixes are depicted in Figure 2 by blue brackets.

The information described in the previous two paragraphs is held in *shareable chunks*. A shareable chunk consists of an immutable data structure that holds (1) a pointer to an ephemeral chunk, (2) an identifier that allows determining whether this chunk is currently uniquely owned or shared (this aspect is described in detail further on), and (3) a view size. In Figure 2, a shareable chunk is depicted as a small solid blue square, with an arrow pointing at a prefix of an ephemeral chunk.

A persistent stack is a nonempty list of shareable chunks, all of which are considered as “shared”. The front shareable chunk is stored in a field named `pfront`, while the remainder of the list is stored in a field named `ptail`. The dashed boxes in Figure 2 represent list cells.

The representation of an ephemeral stack is slightly more complex. Its front chunk is an *ephemeral* chunk, stored in a field named `front`. Its remaining items chunks form a list of *shareable* chunks, each of which may be either uniquely owned by this ephemeral stack or shared. This list is stored in the `tail` field.

A central aspect of the design is how we distinguish between uniquely-owned and shared chunks, that is, how we implement runtime ownership tests and transfers. Runtime ownership tests are needed by some of the operations that update an ephemeral stack: a uniquely-owned ephemeral chunk can be updated in place, whereas a shared ephemeral chunk must be copied before it is updated, unless the update is *monotonic*, a situation that is described later on in this section. Runtime ownership transfers are needed when a persistent stack is frozen: all of the shareable chunks in its

tail list must be marked as potentially shared. A challenge is to perform such a transfer in constant time. A naïve approach, where every shareable chunk is explicitly marked “uniquely owned” or “shared”, would require linear time. Our solution is to let every ephemeral stack and every shareable chunk carry an *identifier*. By convention, if an ephemeral stack and a shareable chunk carry the same identifier, then this stack owns this chunk; otherwise, this chunk is shared. This convention allows an ephemeral stack to abandon the ownership of all of its shareable chunks, in constant time, simply by changing its own identifier to a fresh one.

In Figure 2, the ephemeral stack (A) has identifier 3, stored in its `id` field. Its `tail` field contains a list of two shareable chunks. The first of these has identifier 3, and is therefore uniquely owned by this ephemeral stack, which is why it is drawn outside of the “shared cloud”. The second one has identifier 1, and is therefore not owned by this ephemeral stack, which is why it is drawn inside the “shared cloud”.

The internal type definitions that describe the layout of our data structures appear in Figure 3. Identifiers have type `unit ref`, which means that one can generate a fresh identifier and test the equality of two identifiers in constant time. The `default` field in ephemeral chunks contains a default value that is written into a slot that becomes conceptually empty. The `spare` field in ephemeral stacks contains an optional pointer to an empty ephemeral chunk. Keeping an empty chunk at hand, instead of letting the GC reclaim it, allows us to avoid certain “stuttering” scenarios that would lead to bad time complexity.

**Overview of the Operations.** To complete the presentation of transient stacks, we briefly describe the implementation. Our aim is to give the reader a taste of the arguments involved in reasoning about this data structure. It is however possible to skip this section upon first reading.

To pop an item out of a nonempty ephemeral stack, we pop it out of the front chunk (an ephemeral chunk). If the front chunk becomes empty, then we extract a shareable chunk out of the tail. If this shareable chunk is uniquely owned, then its support (the underlying ephemeral chunk) becomes the new front chunk. Otherwise, its content must be copied into a fresh ephemeral chunk. This allocation and copy require  $O(K)$  operations. Fortunately, this occurs infrequently (at most once per  $K$  pop operations), so one can prove that the amortized time complexity of pop is  $O(1)$ .

To push an item into an ephemeral stack, we push it into its front chunk. If the front chunk is full, however, then we must first move it into the tail, and install an empty front chunk in its place. Inserting the front chunk into the tail requires wrapping it in a new shareable chunk, which we mark “uniquely owned”, by assigning it the same identifier as the ephemeral stack. Allocating a fresh empty front chunk costs  $O(K)$ , but this is needed only when no spare empty chunk is at hand, and (therefore) occurs at most once per  $K$

( $\star [e \rightsquigarrow \text{Stack } L]$  is a notation for  $[\text{Stack } L \ e] \star$ )  
**Definition** `Stack` : `list A`  $\rightarrow$  `loc`  $\rightarrow$  `hprop` := ...

**Lemma** `create_spec` :  $\forall d$ ,  
 SPEC (`create`  $d$ )  
 PRE ( $\$c1$ )  
 POST ( $\text{fun } e \Rightarrow e \rightsquigarrow \text{Stack } \text{nil}$ ).

**Lemma** `push_spec` :  $\forall L \ e \ x$ ,  
 SPEC (`push`  $e \ x$ )  
 PRE ( $\$c2 \star e \rightsquigarrow \text{Stack } L$ )  
 POST ( $\text{fun } \_ \Rightarrow e \rightsquigarrow \text{Stack } (x::L)$ ).

**Lemma** `pop_spec` :  $\forall L \ e$ ,  
 $L \neq \text{nil} \rightarrow$   
 SPEC (`pop`  $e$ )  
 PRE ( $\$c3 \star e \rightsquigarrow \text{Stack } L$ )  
 POST ( $\text{fun } x \Rightarrow \exists L', \setminus [L = x::L'] \star e \rightsquigarrow \text{Stack } L'$ ).

**Figure 4.** Specification of a standard ephemeral stack

push operations, so the amortized time complexity of push is  $O(1)$ . This amortization argument is definitely nonobvious, providing motivation for a formal proof.

To pop an item off a persistent stack, we simply decrement the view field of its front shareable chunk. In case the view becomes empty, we extract a shareable chunk out of `ptail` and make it the new `pfront`. This takes time  $O(1)$ .

To push an item into a persistent stack, we first examine its front shareable chunk. If its view size is less than  $K$  and is equal to the number of items stored in the underlying ephemeral chunk, then the situation is favorable: there is a free slot in the shared chunk that can hold the incoming item. Visually (Figure 2), this corresponds to a situation where the blue bracket covers exactly the gray (occupied) slots and there remains at least one white (unoccupied) slot. In such a case, the new item is written in the free slot, in time  $O(1)$ . This update is performed in place, even though the chunk is shared, because it cannot affect the views of other stacks: it is a *monotonic update*. When the requirements for such a monotonic update are not met, a copy must be performed: a fresh shared chunk must be initialized with appropriate data. In this unfavorable case, the cost of the operation is  $O(K)$ .

To convert a persistent stack into an ephemeral one, we allocate and initialize a fresh ephemeral front chunk, and pick a fresh identifier for this ephemeral stack: this indicates that the new ephemeral stack does not own any of the shared chunks in its tail. This operation takes time  $O(K)$ .

To convert an ephemeral stack into a persistent one, we wrap its front chunk into a shareable chunk, in time  $O(1)$ . This ephemeral stack becomes logically invalid, and must no longer be used. Any shareable chunk that was uniquely owned by this ephemeral stack is now regarded as shared.



### 3 Ephemeral Stacks in Separation Logic

Before attempting to propose a specification for our transient stacks, we recall how an ephemeral stack is usually specified in Separation Logic. Later on (§4), we adapt this standard specification so that it applies to our ephemeral stacks. Throughout the paper, we use the concrete syntax of CFML [Charguéraud 2011].

A *heap predicate* of type `hprop`, also known as an *assertion*, describes a fragment of the mutable state. A *representation predicate* is a heap predicate that describes a mutable data structure. Its parameters are typically: (1) the mathematical object that this data structure represents and (2) the address of this data structure in memory.

Figure 4 gives the type of `Stack`, the representation predicate for ephemeral stacks. We introduce  $e \rightsquigarrow \text{Stack } L$  as a short-hand for `Stack L e`. This heap predicate represents the unique ownership of the stack, and at the same time asserts that the memory location  $e$  points to a valid stack whose elements form the list  $L$ , whose type is `list A`. Throughout the paper, we assume that the type  $A$  of the items is fixed. Our Coq definitions are polymorphic in this type.

The remainder of Figure 4 gives the specifications of the functions `create`, `push` and `pop`. We describe them next.

**Create.** The operation `create d` initializes an empty stack. Its argument  $d$  denotes a default value of type  $A$ . The specification takes the form `SPEC (create d) PRE P POST Q`, which is CFML notation for the triple  $\{P\}(\text{create } d)\{Q\}$ .

The precondition of `create` is  $\$c1$ , which means that `create` consumes  $c1$  time credits. A time credit [Charguéraud and Pottier 2019] is a permission to perform one function call or to execute one iteration of a loop. The function `create` consumes a constant number of credits, thus it admits constant time complexity. In Figure 4, the definition of  $c1$  is not shown. A concrete implementation of stacks would give its own definition of  $c1$ . We do so, for example, in Figure 6. By design, the reasoning rules of Separation Logic with Time Credits allow unused credits to be stored for later use. Therefore, all of the complexity bounds given in this paper are *amortized* [Tarjan 1985].

The postcondition of `create` is  $\text{fun } e \Rightarrow e \rightsquigarrow \text{Stack } \text{nil}$ . The name  $e$  denotes the result of the operation. Here, the value  $e$  is the address of the newly-created stack. The assertion  $e \rightsquigarrow \text{Stack } \text{nil}$  guarantees that this stack is valid and is empty.

**Push.** The specification of `push` indicates that, by invoking `push e x`, one transforms the heap from a state described by  $e \rightsquigarrow \text{Stack } L$  to one described by  $e \rightsquigarrow \text{Stack } (x::L)$ . Furthermore,  $c2$  time credits are consumed, reflecting (again) the fact that this operation has constant time complexity.

```
Variable A : Type. (* type of stack items *)
Parameter Memory : Type → Type. (* "cloud" contents *)
Parameter Shared : Memory A → hprop.
Parameter EStack : Memory A → list A → estack_A → hprop.

Parameter PStack : Memory A → List A → pstack_A → Prop.
```

Figure 5. Type of representation predicates in specifications

**Pop.** In the specification of `pop`, the hypothesis  $L \neq \text{nil}$  requires the stack to be nonempty.<sup>2</sup> The specification indicates that the state of the stack evolves from  $e \rightsquigarrow \text{Stack } L$  to  $e \rightsquigarrow \text{Stack } L'$ , where  $L'$  is the tail of the list  $L$ . It also indicates that the value  $x$  returned by `pop` must be the head of the list  $L$ . The variable  $L'$  is existentially quantified by  $\exists$ . The equality  $L = x::L'$ , which has type `Prop`, is lifted to the level of assertions by the square brackets  $\lfloor \dots \rfloor$ .

## 4 Specification of Transient Stacks

### 4.1 Abstract Predicates

We now introduce three abstract predicates that are involved in the specification of transient stacks.

The set of shared chunks (that is, the “cloud” in Figure 2) is described by an assertion of the form `Shared M`. Internally,  $M$  is a map of locations (addresses of ephemeral chunks) to lists of items (the items currently stored in a chunk). However, an end user need not know this. From the perspective of the user,  $M$  is an abstract entity: its type is `Memory`, a type whose definition is not revealed. The user can think of  $M$  as a name for the current state of the shared cloud.

An ephemeral stack is described by an assertion of the form  $e \rightsquigarrow \text{EStack } M L$ , where  $e$  is a memory location and  $L$  is a list of items. Compared with the predicate `Stack` of the previous section (§3), `EStack` takes  $M$  as an additional parameter: this reflects the fact that an ephemeral stack is valid relative to a certain shared state.

A persistent stack is described by a pure proposition<sup>3</sup> of the form `PStack M L p`, where  $p$  is a Coq record that describes the persistent stack. As in the previous paragraph,  $M$  is the shared memory, and  $L$  is the list of items contained in the stack. The parameter  $M$  is needed because the internal representation of a persistent stack involves shared chunks. The reader may wonder why `PStack M L p` is a proposition, as opposed to an assertion. This is possible because, provided the shared cloud is disregarded, a persistent stack is a pure data structure: the types `schunk` and `pstack` in Figure 3 are immutable record types. CFML automatically reflects immutable OCaml records as Coq records.

<sup>2</sup>This hypothesis could also be placed after the `PRE` keyword. By convention, we prefer to place pure hypotheses outside of triples. This improves the readability of complex specifications and saves a few proof steps.

<sup>3</sup>A pure proposition is an ordinary Coq proposition. Its type is `Prop`.

**Lemma** `ecreate_spec` :  $\forall M x,$   
 SPEC (`ecreate x`)  
 PRE  $(\$ (K+4))$   
 POST (`fun e  $\Rightarrow$  e  $\rightsquigarrow$  EStack M nil`).

**Lemma** `epush_spec` :  $\forall M L e x,$   
 SPEC (`epush e x`)  
 PRE  $(\$7 \star e \rightsquigarrow \text{EStack } M L)$   
 INV (`Shared M`)  
 POST (`fun _  $\Rightarrow$  e  $\rightsquigarrow$  EStack M (x::L)`).

**Lemma** `epop_spec` :  $\forall M L e,$   
 $L \neq \text{nil} \rightarrow$   
 SPEC (`epop e`)  
 PRE  $(\$11 \star e \rightsquigarrow \text{EStack } M L)$   
 INV (`Shared M`)  
 POST (`fun x  $\Rightarrow$   $\exists L', \backslash [L = x::L'] \star e \rightsquigarrow \text{EStack } M L'$` ).

**Figure 6.** Specification of ephemeral stacks

**Lemma** `pcreate_spec` :  $\forall M x,$   
 SPEC (`pcreate x`)  
 MONO `M`  
 PRE  $(\$ (K+3))$   
 POST (`fun M' p  $\Rightarrow$   $\backslash [\text{PStack } M' \text{ nil } p]$` ).

**Lemma** `ppush_spec` :  $\forall M L p x,$   
`PStack M L p  $\rightarrow$`   
 SPEC (`ppush p x`)  
 MONO `M`  
 PRE  $(\$ (K+7))$   
 POST (`fun M' p'  $\Rightarrow$   $\backslash [\text{PStack } M' (x::L) p']$` ).

**Lemma** `ppop_spec` :  $\forall M L p,$   
`PStack M L p  $\rightarrow$`   
 $L \neq \text{nil} \rightarrow$   
 SPEC (`ppop p`)  
 PRE  $(\$7)$   
 INV (`Shared M`)  
 POST (`fun M' (p',x)  $\Rightarrow$   $\exists L', \backslash [L = x::L' \wedge \text{PStack } M' L' p']$` ).

**Figure 7.** Specification of persistent stacks

Figure 5 gives the Coq types of the three predicates. The Coq types `estack_A` and `pstack_A` correspond to the OCaml types `'a estack` and `'a pstack`. The type `estack_A` is defined by CFML as a synonym for `loc`, the type of memory locations. The type `pstack_A` is defined by CFML as a Coq record type.

## 4.2 Specification of Ephemeral Stacks

The specification of the operations on ephemeral stacks appears in Figure 6. Compared with the specification of standard stacks given in Figure 4, it differs in two ways. First, as previously explained, the assertion  $e \rightsquigarrow \text{Stack } L$  is replaced with  $e \rightsquigarrow \text{EStack } M L$  to account for the dependence of the data structure on a shared state. Second, the specification

**Parameter** `Extend` : `Memory A  $\rightarrow$  Memory A  $\rightarrow$  Prop`.

**Lemma** `EStack_mon` :  $\forall e M M' L, \text{Extend } M M' \rightarrow$   
 $(e \rightsquigarrow \text{EStack } M L) \vdash (e \rightsquigarrow \text{EStack } M' L)$ .

**Lemma** `PStack_mon` :  $\forall p M M' L, \text{Extend } M M' \rightarrow$   
 $(\text{PStack } M L p) \rightarrow (\text{PStack } M' L p)$ .

**Figure 8.** Monotonicity lemmas

involves an additional line `INV (Shared M)`. This notation is syntactic sugar for two occurrences of the assertion `Shared M`, one in the precondition and one in the postcondition.<sup>4</sup> The presence and invariance of `Shared M` reflect the fact that an operation on an ephemeral stack requires access to the shared state, and does not modify or extend it.<sup>5</sup>

For the benefit of users who need only ephemeral stacks and never persistent stacks, it is possible to give to our ephemeral stacks the standard specification of Figure 4. To that end, it suffices to define  $e \rightsquigarrow \text{Stack } L$  as  $e \rightsquigarrow \text{EStack } \emptyset L$ . The assertion `Shared  $\emptyset$`  happens to be equivalent to the “empty heap” assertion  `$\backslash []$` , so it can be elided. We have verified that our code admits this simplified specification.

The precondition of `ecreate` does not require a shared memory `Shared _`. The postcondition provides  $e \rightsquigarrow \text{EStack } M \text{ nil}$ , where `M` is chosen by the user: it may be a memory that is at hand, or the empty memory  `$\emptyset$` .

The precondition of `ecreate` requires  $K + 4$  time credits, reflecting the cost of allocating and initializing an array of size  $K$ . Ideally, we would like to write  $O(K)$  in the specification, instead of  $K + 4$ , and to set things up so that  $K$  is not  $O(1)$ , because we wish to distinguish between  $O(1)$  and  $O(K)$ . This is made possible in principle by Guéneau’s work [2019]. Yet, we leave this aspect to future work, so as to avoid dealing at this stage with an additional layer of technicality.

## 4.3 Specification of Persistent Stacks

Persistent stacks, contrary to ephemeral stacks, do operate on the shared state. To achieve persistence, it is fundamental that the shared state evolve *monotonically*: any view on a prefix of a shared chunk must be preserved when the shared state evolves. The manner in which the shared state evolves is modeled by a preorder `Extend M M'`. The definition of this preorder is not revealed to the user. All that matters is that an evolution of the shared state preserves the meaning of

<sup>4</sup>More generally, `SPEC (f x) PRE P INV I POST Q`, where `I` is of type `hprop`, is a notation for the triple  $\{P * I\} (f x) \{ \lambda y. (Q y) * I \}$ .

<sup>5</sup>The operations on ephemeral stacks affect or create only uniquely-owned chunks. They never modify or extend the shared state. In an extension of Separation Logic with read-only permissions, such as Charguéraud and Pottier’s [2017], these operations could presumably require only read-only access to the shared state.

**Lemma** `pstack_to_estack_spec` :  $\forall M L p$ ,  
`PStack M L p`  $\rightarrow$   
`SPEC (pstack_to_estack p)`  
`PRE $(2*K+7)`  
`INV (Shared M)`  
`POST (fun e  $\Rightarrow$  e  $\rightsquigarrow$  EStack M L).`

**Lemma** `estack_to_pstack_spec` :  $\forall M L e$ ,  
`SPEC (estack_to_pstack e)`  
`MONO M`  
`PRE $(2 * e  $\rightsquigarrow$  EStack M L)`  
`POST (fun M' p  $\Rightarrow$  \[PStack M' L p]).`

**Figure 9.** Specification of the conversions

existing ephemeral stack and persistent stacks: this is expressed by the two lemmas shown in Figure 8. There, the symbol  $\vdash$  denotes entailment of assertions.

Every operation on persistent stacks requires a shared state described by `Shared M` and produces a shared state described by `Shared M'`, for some  $M'$  such that `Extend M M'` holds. For example, the operation `ppush p x`, shown below, features a pure precondition `PStack M L p` and a pure postcondition `PStack M' (x::L) p'`, where  $p'$  denotes the result stack.

**Lemma** `ppush_spec_without_syntactic_sugar` :  $\forall M L p x$ ,  
`PStack M L p`  $\rightarrow$   
`SPEC (ppush p x)`  
`PRE $(K+7) * Shared M)`  
`POST (fun p'  $\Rightarrow$   $\exists M'$ , \[Extend M M'] * Shared M'`  
 $\star$  \[PStack M' (x::L) p']).

Because many specifications involve this pattern where the shared state evolves monotonically, we introduce an ad hoc notation: we write `MONO M` before the precondition, and bind  $M'$  as an extra argument of the postcondition.<sup>6</sup> The specifications in Figure 7 illustrate the use of this syntactic sugar.

#### 4.4 Specification of the Conversion Operations

The specification of the conversion operations appears in Figure 9. The function `pstack_to_estack` expects a persistent stack whose logical model is a list  $L$  and produces a new ephemeral stack whose logical model is also  $L$ . The precondition establishes that the operation has constant time complexity. Because this operation reads but does not modify the shared state, the specification `INV (Shared M)` suffices.

The function `estack_to_pstack` requires an ephemeral stack whose logical model is  $L$  and produces a persistent

<sup>6</sup>When expanding the notation `MONO`, Coq determines which predicate `Shared` and which preorder `Extend` are desired, based on the type of  $M$ , thanks to a type class. In general, the `MONO` notation applies to any  $M$  of type  $T$  for which there is a predicate  $R$  of type  $T \rightarrow \text{hprop}$  and a preorder  $\leq_T$ . The notation `SPEC (f x) MONO M PRE P POST Q` stands for the triple:

$$\{R M \star P\} (f x) \{\lambda y. \exists M', [M \leq_T M'] \star R M' \star (Q M' y)\}.$$

**Record** `EChunk_inv`  
`(L:list A) (D:list A) (top:int) (default:A) : Prop :=`  
`{ efront :  $\forall i, 0 \leq i < \text{top} \rightarrow D[i] = (\text{rev } L)[i]$ ;`  
`etail :  $\forall i, \text{top} \leq i < K \rightarrow D[i] = \text{default}$ ;`  
`elength : length L = top  $\wedge$  length D = K;`  
`etop :  $0 \leq \text{top} \leq K$  }.`

**Definition** `EChunk (L:list A) (c:echunk_ A) : hprop :=`  
 $\exists$  (data:loc) (D:list A) (top:int) (default:A),  
 $c \rightsquigarrow$  '{ data' := data; top' := top; default' := default }  
 $\star$  data  $\rightsquigarrow$  Array D  
 $\star$  \[EChunk\_inv L D top default].

**Figure 10.** Representation of ephemeral chunks

stack whose model is also  $L$ . The original ephemeral stack is invalidated, that is, the permission to access it is lost: indeed, the assertion `EStack M L` does not appear in the postcondition. The shared state evolves monotonically during the operation, because the uniquely-owned chunks of the ephemeral stack become shared and are moved into the shared state.

## 5 Overview of the Proof

We now explain how the abstract predicates presented earlier (§4) are defined, and give insights about the key points of the proof.

### 5.1 Representation of Ephemeral Chunks

The predicate  $c \rightsquigarrow$  `EChunk L` describes an ephemeral chunk at address  $c$ , whose items form the list  $L$ . Its definition appears in Figure 10. Two points-to assertions claim the unique ownership of the chunk record<sup>7</sup> and of its data array. The auxiliary proposition `EChunk_inv`, a conjunction of pure propositions, captures the pure part of the invariant. It states that the data array has size  $K$ , that its first  $\text{top}$  elements are the elements of the list  $L$  (in reverse order, because the front of the array corresponds to the bottom of the stack), and that the remaining slots of the array store the `default` value.

A chunk can be viewed as a stack of bounded capacity. It is convenient to define the operations `empty`, `push`, and `pop` at the level of chunks, so as to use them as building blocks in the implementation of stacks. The specification of these operations is essentially the same as in the case of a standard stack (Figure 4). The only difference is that `push` requires the chunk not to be full (specifications are given in Appendix A.3).

<sup>7</sup>The field names associated with OCaml records appear in Coq with an extra “quote” symbol, which is automatically added by CFML to avoid name clashes. Thus, whereas the OCaml record type `echunk` has three fields `data`, `top`, `default`, the Coq record type `echunk_` has three fields `data'`, `top'`, `default'`.

**Record Memory** : **Type** :=

```
{ echunks : map (echunk_ A) (list A)
  ids : map (ref_ unit) unit }.
```

**Definition** Shared (M:Memory A) : hprop :=  
Group EChunk M.echunks ★ Group Ref M.ids.

**Definition** Extend (M M':Memory A) : Prop :=  
let E := M.echunks in  
let E' := M'.echunks in  
(∀ x, x ∈ dom E → (x ∈ dom E') ∧ (Suffix E[x] E'[x]))  
∧ (dom M.ids) ⊆ (dom M'.ids).

**Figure 11.** Representation of the shared memory

## 5.2 Representation of the Shared State

Next, we present the definition of the predicate Shared M and of the preorder Extend M M'. We must begin by defining the type Memory, which is the type of M. These definitions appear in Figure 11. (The type ref\_ unit is defined by CFML as loc.)

The definition of the type Memory indicates that a memory M is a record that holds two components. First, it holds a map of locations (addresses of ephemeral chunks) to lists of items (items stored in the chunks). This map describes the content of the shared cloud. Second, it holds a set of the identifiers that may appear in shared shareable chunks. This is the set of the identifiers that are no longer associated with an ephemeral stack.

Keeping track of the set of shared identifiers is useful for the following reason. When we generate a fresh identifier for an ephemeral stack, we must prove that this identifier is distinct from every identifier carried by an existing shareable chunk—otherwise, an unintended “ownership capture” would occur. Our solution is to let the shared state include the unique ownership of all shared identifiers. Thus, when we generate a new identifier, the standard Separation Logic axiom  $x \rightsquigarrow \text{Ref } () * y \rightsquigarrow \text{Ref } () \vdash x \neq y$  allows us to prove that no such collision can occur.

The predicate Shared M, which appears next in Figure 11, claims the ownership of the ephemeral chunks and identifiers described by the memory M. Its definition is straightforward; it relies on the Group combinator, an iterated separating conjunction over a finite map.<sup>8</sup>

Figure 11 ends with the definition of Extend M M'. This preorder describes how the shared state evolves. It asserts that (1) a chunk that is allocated in M is also allocated in M'; (2) if in state M this chunk contains a list of items L, then in state M' it must contain a list that extends L; and (3) the set of shared identifiers grows over time.

Initially, the user may forge an empty shared memory out of thin air, using the entailment  $\vdash \text{Shared } \emptyset$ . Then,

<sup>8</sup>Group R G is defined as follows:

```
Definition Group (a A:Type) (R:A → a → hprop) (G:map a A) : hprop :=  
(Map.fold (fun x X acc → x ∼ R X ★ acc) \[] G) ★ \[finite G].
```

although the specification of every operation on stacks mentions a single instance of the predicate Shared, a user may wish to work with several instances of this predicate, and may eventually need to merge two such instances into one. We prove an entailment lemma that allows this:  $\text{Shared } M1 \star \text{Shared } M2 \vdash \exists M, \text{Shared } M \star \backslash[\text{Extend } M1 \ M \wedge \text{Extend } M2 \ M]$ .

## 5.3 Shareable Chunks

A central aspect of our contribution is the setup of the representation predicates for shareable chunks. Several concerns must be taken into account. On the one hand, we wish to verify the elementary operations on shareable chunks only once, under the assumption that it is permitted to access to the *support* of the shareable chunk—that is, the underlying ephemeral chunk. The predicate SChunk captures this assumption. On the other hand, we need to describe the ownership policy that applies to the support of a shareable chunk. Within a persistent stack, the support of every shareable chunk is owned by the shared state. Within an ephemeral stack, the support of a shareable chunk may be owned either by the stack or by the shared state, depending on the outcome of the comparison between the identifier of the shareable chunk and the identifier of the stack. These situations are described by two more predicates, SChunkShared and SChunkMaybeOwned.

Moreover, in the case where the support is uniquely owned, we maintain the invariant that the view of the shareable chunk on its support is full. That is, the view coincides with the data; it is never a strict prefix of the data. Some of our auxiliary predicates carry a Boolean parameter align which indicates when this requirement is imposed.

The representation predicates used for shareable chunks are defined in Figure 12, and are described next.

The proposition SChunk\_inv align S L s captures the internal invariant of a shareable chunk s. The parameter S is the list of all items stored in the underlying ephemeral chunk. The parameter L is the suffix of length s.view of the list L; it represents the limited view that this shareable chunk has onto its support. If the Boolean parameter align is true, then the lists L and S must coincide.

The assertion  $s \rightsquigarrow \text{SChunk } \text{align } S \ L$  describes a shareable chunk that owns its support. As explained earlier, an operation on a shareable chunk needs access to its support, which is why this assertion is useful. However, ownership of the support is temporary: we typically extract it from somewhere else (either an ephemeral stack or the shared state) and return it afterwards.

The assertion  $s \rightsquigarrow \text{SChunkUniquelyOwned } L$  describes a shareable chunk s that is uniquely owned. It is a specialization of SChunk with an “alignment” constraint; that is, align is instantiated with true. In contrast with the assertions that follow, it does not depend on the shared memory M.

The pure proposition SChunkShared M L s describes a shareable chunk s that appears in a persistent stack. Its support (an ephemeral chunk) must be owned by the shared state:



```

Record SChunk_inv
  (align:bool) (S L:list A) (s:schunk_ A) : Prop :=
  let v := s.view' in
  { ssize : 0 ≤ v ≤ length S;
    slist : Suffix L S;
    slength : length L = v;
    salig : align → v = length S; }.

Definition SChunk
  (align:bool) (S L:list A) (s:schunk_ A) : hprop :=
  s.support' ∼ EChunk S
  ★ \[SChunk_inv align S L s].

Definition SChunkUniquelyOwned
  (L:list A) (s:schunk_ A) : hprop :=
  s ∼ SChunk true L L.

Definition SChunkShared
  (M:Memory A) (L:list A) (s:schunk_ A) : Prop :=
  s.support' ∈ dom (M.echunks)
  ∧ SChunk_inv false (M.echunks)[s.support'] L s.

Definition SChunkMaybeOwned
  (M:Memory A) (id:id_) (L:list A) (s:schunk_ A) : hprop :=
  If s.owner' = Some id
  then s ∼ SChunkUniquelyOwned L
  else \[SChunkShared M L s]

```

**Figure 12.** Representation of shareable chunks

thus, the address of its support must appear in the domain of the map  $M.echunks$ . The expression  $(M.echunks)[s.support']$  denotes the list of all elements stored in the support. This is why we instantiate the parameter  $S$  of  $SChunk\_inv$  with this list. Meanwhile, the list  $L$  represents the limited view of this shareable chunk onto its support.

The assertion  $s \sim SChunkMaybeOwned M id L$  describes a shareable chunk  $s$  that appears in an ephemeral stack whose identifier is  $id$ . If the identifier of  $s$  is equal to  $id$ , then this assertion boils down to  $s \sim SChunkUniquelyOwned L$ . Otherwise, it is equivalent to  $\backslash[SChunkShared M L s]$ , a pure assertion, which does not imply the ownership of the support.

Regarding the verification of the elementary operations on shareable chunks, two aspects are worth mentioning. First, we must verify that an operation on a uniquely-owned shareable chunk maintains the alignment property. Second, we must verify that the cost of a push operation is  $O(1)$  when alignment holds, whereas it is  $O(K)$  otherwise.

#### 5.4 Persistent Stacks

Figure 13 presents the representation predicate for persistent stacks, which takes the form  $PStack M L p$ . Throughout the figure,  $L$  denotes the list of all items in the stack,  $LF$  is the list of items in the front chunk, and  $LS$ , a list of lists, describes the items in the remaining (shareable) chunks.

```

Record Stack_inv
  (L:list A) (LF:list A) (LS:list (list A)) : Prop :=
  { stack_list : L = LF ++ concat LS;
    stack_tail_full : Forall (fun xs ⇒ length xs = K) LS;
    stack_tail_nil : LF = nil → LS = nil }.

Definition valid_id (M : Memory A) (id:option Id.t_) :=
  match id with
  | None ⇒ True
  | Some x ⇒ x ∈ dom M.ids end.

Record PStack_inv
  (M:Memory A) (L:list A) (p:pstack_ A)
  (LF:list A) (LS:list (list A)) : Prop :=
  let pf := p.pfront' in
  let pt := p.ptail' in
  { pstack_stack : Stack_inv L LF LS;
    pstack_chunks :
      Forall2 (SChunkShared M) (LF::LS) (pf::pt)
    pstack_version :
      Forall (fun p ⇒ valid_id M p.owner') (pf::pt) }.

Definition PStack
  (M:Memory A) (L:list A) (p:pstack_ A) : Prop :=
  ∃ LF LS, PStack_inv M L p LF LS.

```

**Figure 13.** Representation of persistent stacks

The proposition  $Stack\_inv L LF LS$  asserts that the list  $L$  is indeed the concatenation of  $LF$  with all of the lists in  $LS$ , each of which must have length  $K$ . (That is, every shareable chunk in the tail must be full.) It also enforces an invariant that allows a fast emptiness test: if the front chunk is empty, then the whole stack is empty.

We distinguish the predicates  $Stack\_inv$  and  $PStack\_inv$  because the former is used also for ephemeral stacks. The additional properties imposed by  $PStack\_inv$  capture the fact that every shareable chunk in the tail is in fact shared: that is, (1) it is described by the predicate  $SChunkShared$ , and (2) its identifier must be a member of the set of shared identifiers  $M.ids$ , a property that is expressed by the predicate  $valid\_id$ .

#### 5.5 Ephemeral Stacks

The representation predicate  $e \rightsquigarrow EStack M L$  is defined at the bottom of Figure 14, and is explained next. First, a points-to assertion claims the ownership of the 4-field record at address  $e$ . Second, the assertion  $front \rightsquigarrow EChunk LF$  claims the ownership of the ephemeral front chunk, whose items form the list  $LF$ . Third, the tail is described as a list of shareable chunks, each of which may or may not be uniquely owned, as prescribed by the predicate  $SChunkMaybeOwned M id$ , where  $id$  is the identifier of this ephemeral stack.

```

Definition potential_push
(NF:int) (spare:option (echunk_ A)) : int :=
  match spare with
  | None => 1 + NF
  | Some _ => 0 end.

Definition potential_pop
(NF:int) (tail:list (schunk_ A)) (id:id_) : int :=
  match tail with
  | nil => 0
  | p::_ => If p.owner' = Some id
            then 0
            else 2 + K - NF end.

Record EStack_inv
(M:Memory A) (id:id_) (tail:list (schunk_ A))
(L:list A) (LF:list A) (LS:list (list A)) : Prop :=
{ estack_stack : Stack_inv L LF LS;
  estack_tail_ids : Forall (fun p =>
    valid_id M p.owner' ∨ p.owner' = Some id) tail }.

Definition EStack
(M:Memory A) (L:list A) (e:estack_ A) : hprop :=
  ∃ front tail spare id LF LS,
  e ∼ { front' := front; tail' := tail;
        spare' := spare; id' := id }
  ★ front ∼ EChunk LF
  ★ tail ∼ ListOf (SChunkMaybeOwned M id) LS
  ★ spare ∼ OptionOf (EChunk nil)
  ★ id ∼ Ref tt
  ★ \[EStack_inv M id tail L LF LS]
  ★ $(potential_push NF spare + potential_pop NF tail id).

```

**Figure 14.** Representation of ephemeral stacks

The combinator `ListOf`, a *higher-order representation predicate* [Charguéraud 2016], is standard in CFML. The combinator `ListOf R LS xs` denotes the fold operation of the representation predicate `R`, pairwise over the items from the lists `LS` and `xs`, over the separating logic monoid.<sup>9</sup>

Coming back to the definition of `EStack`, the next assertion, `spare ∼ OptionOf (EChunk nil)`, claims the ownership of the spare chunk if it is present, and indicates that this spare chunk must be empty. The combinator `OptionOf`, not shown, is analogous to `ListOf`.

The assertion `id ∼ Ref tt` asserts that an ephemeral stack owns its identifier. When an ephemeral stack is converted

<sup>9</sup>The definition of `ListOf` is as follows.

```

Fixpoint ListOf (A B:Type)
(R:A → B → hprop) (LS:list A) (xs:list B) : hprop :=
  match LS,xs with
  | nil, nil => \[]
  | L::LS, x::xs => x ∼ R L ★ xs ∼ ListOf R LS
  | _, _ => \[False] end.

```

into a persistent stack, the ownership of this identifier is transferred to the shared state—a purely logical operation.

The proposition `EStack_inv M id tail L LF LS` captures the pure invariants associated with ephemeral stacks. It includes those that are common with persistent stacks, described by `Stack_inv`, and adds the property `estack_tail_ids`, which states that every shareable chunk in the tail carries an identifier that is either `id`, the identifier of this ephemeral stack, or a shared identifier.

Last but not least, the definition of `EStack` includes a number of time credits. They represent the *potential* of the data structure, in the traditional terminology of amortized complexity analysis [Tarjan 1985]. These credits have been set aside and can be used to pay for infrequent expensive operations. Roughly, `potential_push NF spare` is needed to pay for the next allocation of a fresh front chunk, unless an empty spare chunk is already at hand; and `potential_pop NF tail id` accounts for the cost of the copy operation that is required if a shared chunk must be extracted out of the tail. The function `potential_pop` is nonzero only when the tail is nonempty and begins with a shared shareable chunk.

## 6 Iterators

The library `Sek`, which motivates our work, features efficient iterators. Several iterators may concurrently operate on the same sequence. For example, one can use two iterators to copy items from one sequence segment into another sequence segment, a “blit” operation. In this paper, we consider iterators that are similar to `Sek`’s. They are implemented as mutable records and contain direct pointers into chunks that a part of a stack’s internal representation.

An iterator on an ephemeral stack supports a set operation which replaces the current item, that is, the item to which the iterator currently points. If the current item lies in a uniquely-owned shareable chunk, the write operation can be performed in place, in constant time. If it lies in a shared shareable chunk, a copy is necessary: a new chunk and a new shareable chunk must be allocated. In the latter case, the tail of the stack, a singly-linked list of shareable chunks, must also be updated with this new shareable chunk. Because this update can take place at an arbitrary depth in the list, it requires linear time. (In `Sek`, which uses a tree structure, it requires only logarithmic time.)

When a chunk is copied, the existing iterators that point directly to this chunk become invalid, and must no longer be used. Unfortunately, one cannot expect the user to know precisely when a copy take place and which existing iterators might be affected: after all, the existence of chunks, and the manner in which chunks are shared, are supposed to be implementation details. We address this problem via an over-approximation: our specification states that updating an ephemeral stack invalidates *all* existing iterators on this stack, except the iterator used to perform the update.

```

type 'a iterator
val iter_on_estack : 'a estack -> 'a iterator
val finished : 'a iterator -> bool
val move : 'a iterator -> unit
val get : 'a iterator -> 'a
val set : 'a iterator -> 'a -> unit

```

**Figure 15.** API of iterators on ephemeral stacks

```

type 'a iterator =
{ mutable focused : 'a echunk;
  mutable fview : int;
  mutable fid : id;
  rest : 'a schunk list ref;
  mutable traveled : int;
  uestack : 'a estack }

```

**Figure 16.** Implementation of iterators on ephemeral stacks

We use the word “invalidation” in a *static* sense: our specification forbids the use of an iterator that is considered invalid, and does not specify what happens at runtime if the user violates the rule and uses such an iterator anyway. If desired, we could add *dynamic* validation instructions, so as to keep track of validity and fail at runtime if the user attempts to use an invalid iterator.

The key ideas of our formalization of iterators are the following. First, to specify when iterators are invalidated, we follow Pottier [2017] and expose (abstract names for) internal states in representation predicates. Second, in order to allow direct pointers deep into the data structure, we let these concrete states include the addresses of internal chunks. Third, we distinguish two different use cases for `set`, and present a specification that is tailored for the common special case where an ephemeral stack owns all of its chunks, that is, for the special case where there is no sharing at all. By using conditionals in pre- and postconditions, we are able to establish two specifications for `set` while examining its source code only once.

Our formalization also includes iterators on persistent sequences. To avoid duplication of work, we factor the aspects that are common to iterators on ephemeral and persistent sequences. In the interest of space, we discuss only iterators on ephemeral sequences.

**Interface.** Figure 15 presents the API of iterators on ephemeral stacks. When it is created, an iterator points to the item at the top of the stack. Calling `move` moves the iterator to the next item, and can move it past the last item. The operations `get` and `set` read and replace the current item. The operation `finished` returns true if the iterator points past the last item.

**Implementation.** Figure 16 presents the internal representation of iterators. The field `focused` is a pointer to the ephemeral chunk that contains the current item. The field `fview` contains the index of the current item within this chunk. This index is updated by `move`. The field `fid` is a local copy of the identifier of the shareable chunk which contains this ephemeral chunk.<sup>10</sup> This identifier is used to enable a dynamic ownership test during a `set` operation. The field `rest` holds a suffix of the tail of the ephemeral stack. It is used to keep track of the chunks that remain to be traversed; its head chunk is extracted when the traversal of the currently focused chunk is completed. The field `traveled` keeps track of the number of chunks that have already been completely traversed. The field `uestack` is a pointer to the ephemeral stack that the iterator traverses. These two pieces of information are needed during a `set` operation, when the chunk under focus must be copied and the new chunk must be inserted at this position in the tail of the ephemeral stack.

**Specification.** An iterator is described by an assertion of the form  $it \rightsquigarrow \text{Iterator } st \ i$ , where  $it$  is the address of the iterator,  $i$  is the index of the current item, and  $st$  is an abstract name for the current internal state of the stack.

In addition to the predicate  $e \rightsquigarrow \text{EStack } ML$ , we introduce a new predicate,  $e \rightsquigarrow \text{EStackInState } st \ ML$ , where a name  $st$  appears. They are related by the equivalence law:

$$(e \rightsquigarrow \text{EStack } ML) \dashv\vdash (\exists st, e \rightsquigarrow \text{EStackInState } st \ ML).$$

A declarative reading of this law is that “a stack is a stack in some internal state”. From a more operational point of view, by using this law from left to right and by eliminating the existential quantifier, one introduces a fresh abstract name  $st$  for the current internal state of the stack. By introducing an existential quantifier and using this law from right to left, one forgets the name of the current internal state.

The ability of naming the internal state of a stack is useful, because it allows us to state that “the iterator  $it$  is valid as long as the stack remains in the state  $st$ ”, that is, as long as the stack is not updated.

In order to call `push e` or `pop e`, one must provide the assertion  $e \rightsquigarrow \text{EStack } ML$ . If an assertion of the form  $e \rightsquigarrow \text{EStackInState } st \ ML$  is currently at hand, then one must use the above equivalence law from right to left, thereby forgetting the name of the current internal state of the stack  $e$ . One can then no longer prove that the stack is in the state  $st$ . Therefore, all existing iterators on this stack become unusable, as intended, since we wish to view all existing iterators as invalid after the stack has been updated.

The specification of iterators appears in Figure 17. We write  $L[i]$  for the  $i$ -th element of the list  $L$ , and  $L[i:=x]$  for

<sup>10</sup>When the current item belongs to the front chunk, the field `fid` is set to the identifier of the ephemeral stack, to reflect ownership of that chunk.

**Lemma** `iter_on_estack_spec` :  $\forall st ML e,$   
`SPEC (iter_on_estack e)`  
`PRE ($2)`  
`INV (e  $\rightsquigarrow$  EStackInState st ML)`  
`POST (fun it  $\Rightarrow$  it  $\rightsquigarrow$  Iterator st 0).`

**Lemma** `finished_spec_ephemeral` :  $\forall st ML e it i,$   
`SPEC (finished it)`  
`PRE ($1)`  
`INV (e  $\rightsquigarrow$  EStackInState st ML  $\star$  it  $\rightsquigarrow$  Iterator st i)`  
`POST (fun b  $\Rightarrow$  b = isTrue (i = length L)).`

**Lemma** `get_spec_ephemeral` :  $\forall st ML e it i,$   
 $i \neq \text{length } L \rightarrow$   
`SPEC (get it)`  
`PRE ($3)`  
`INV (Shared M  $\star$  e  $\rightsquigarrow$  EStackInState st ML`  
 $\star$  it  $\rightsquigarrow$  Iterator st i)  
`POST (fun x  $\Rightarrow$  x = L[i]).`

**Lemma** `move_spec_ephemeral` :  $\forall st ML e it i,$   
 $i \neq \text{length } L \rightarrow$   
`SPEC (move it)`  
`PRE ($4  $\star$  it  $\rightsquigarrow$  Iterator st i)`  
`INV (e  $\rightsquigarrow$  EStackInState st ML)`  
`POST (fun _  $\Rightarrow$  it  $\rightsquigarrow$  Iterator st (i+1)).`

**Lemma** `set_spec_with_sharing` :  $\forall st ML e it i x,$   
 $i \neq \text{length } L \rightarrow$   
`SPEC (set it x)`  
`PRE ($(\text{length } L + K + 10)`  
 $\star$  it  $\rightsquigarrow$  Iterator st i  
 $\star$  e  $\rightsquigarrow$  EStackInState st ML)  
`INV (Shared M)`  
`POST (fun _  $\Rightarrow$   $\exists st', it \rightsquigarrow$  Iterator st' i`  
 $\star$  e  $\rightsquigarrow$  EStackInState st' M (L[i:=x])).

**Lemma** `set_spec_without_sharing` :  $\forall st L e it i x,$   
 $i \neq \text{length } L \rightarrow$   
`SPEC (set it x)`  
`PRE ($6  $\star$  e  $\rightsquigarrow$  EStackInState st  $\emptyset$  L)`  
`INV (it  $\rightsquigarrow$  Iterator st i)`  
`POST (fun _  $\Rightarrow$  e  $\rightsquigarrow$  EStackInState st  $\emptyset$  (L[i:=x])).`

**Figure 17.** Specification of iterators

an update of this list at index  $i$ . Operation on an iterators it require two assertions  $it \rightsquigarrow \text{Iterator } st \ i$  and  $e \rightsquigarrow \text{EStackInState } st \ ML$ , both of which refer to a common internal state  $st$ . The operations `get` and `set` additionally require `Shared M`, because they need access to the shared chunks.

As announced earlier, we provide two specifications for `set`. The first specification indicates that `set` has complexity  $O(n + K)$ , where  $n$  is the length of the stack. It is always applicable, but pessimistic. The second specification can be exploited only in the special case where there is no sharing,

that is, where  $M$  is the empty memory  $\emptyset$ . In this case, `set` has complexity  $O(1)$ . As long as the user exploits ephemeral stacks only,  $M$  remains  $\emptyset$ , and `set` is guaranteed to be cheap.

**Proof.** Internally, an internal state  $st$  is defined as a record of (1) the address of the stack, (2) its unique identifier, and (3) the list of its shareable chunks. This list contains the addresses of all ephemeral chunks involved in the representation of this stack and all of the identifiers used to determine which chunks are uniquely owned by this stack.

In order to verify `set` just once, we derive the two specifications in Figure 17 from a single specification that uses conditionals. This specification is shown in Appendix A.5.

## 7 Related Work

Our persistent stacks are an example of *full persistence*, where any version of a persistent structure can be read and modified. There exist several weaker notions of persistence. Driscoll et al. [1989] describe *partial persistence*, where old versions can be read but not updated. *Semi-persistence* [Conchon and Filliâtre 2008] allows read and write access only to the current version and its ancestors.

Separation Logic [Reynolds 2002] has been used to verify a wide variety of data structures and real-world programs; see the survey by O’Hearn [2019].

Mehnert et al. [2012] use Separation Logic to specify and verify a Java implementation of “snapshotable trees”. These trees support read-only snapshots and read-only iterators. Because a snapshot cannot exist in the absence of the original tree, there is no need for a state that is shared between several structures. Instead, the authors use an abstract predicate that globally describes one tree together with all of its snapshots and iterators. Sharing is handled in a purely internal manner. The verified implementation does not mutate shared data, but the authors believe that their proof technique also applies to more efficient implementations where shared data is modified.

Pottier [2017] uses Separation Logic to specify and verify a simple mutable data structure, namely a hash table, as well as a form of iterator, which he refers to a *cascade*, actually a delayed list. We follow his approach to specifying iterator invalidation.

Charguéraud and Pottier [2019] use Separation Logic with Time Credits to verify the correctness and complexity of the Union-Find data structure. Guéneau [2019] introduces negative time credits and investigates the use of the big- $O$  notation at scale. In this paper, we use neither of these features; it would be desirable, in future work, to use big- $O$  notation in our specifications.

Our specification explicitly imposes the existence of a shared internal state, whose nature remains abstract, and whose evolution is monotonic. This is a common pattern, which has been studied in several papers [Ahman et al. 2018; Pilkiewicz and Pottier 2011; Timany and Birkedal 2021]. It is



also common practice to *hide* the existence of this internal state entirely by using the anti-frame rule [Pottier 2008] or an Iris invariant [Jung et al. 2018; Timany and Birkedal 2021]. CFML does not allow hiding an internal state in such a way. Because of this, the end user perhaps loses some comfort, but we do not believe that this is a severe problem. In fact, the fact that the internal state is not hidden allows us to distinguish the cases  $M = \emptyset$  and  $M \neq \emptyset$  in the specification of the set operation on iterators (Figure 17). If the internal state was hidden, we would have to record the absence or presence of sharing in some other way.

## 8 Conclusion

We have verified a scaled-down version of the Sek library, which presents the same nontrivial chunk ownership policy and the same optimization patterns in the implementation of iterators as the full-fledged Sek. Our verified implementation represents roughly 250 lines of OCaml code, and involves roughly 1400 lines of specifications and auxiliary definitions, and 1800 lines of proofs. In the future, we look forward to tackle the challenge of verifying the actual Sek library.

## References

- Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2014. [Theory and Practice of Chunked Sequences](#). In *Algorithms (ESA)*, Vol. 8737. Springer, 25–36.
- Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. [Recalling a witness: foundations and applications of monotonic state](#). *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 65:1–65:30.
- Arthur Charguéraud. 2011. [Characteristic Formulae for the Verification of Imperative Programs](#). In *International Conference on Functional Programming (ICFP)*. 418–430.
- Arthur Charguéraud and François Pottier. 2021. *Sek*.
- Arthur Charguéraud. 2016. [Higher-order representation predicates in separation logic](#). In *Certified Programs and Proofs (CPP)*. 3–14.
- Arthur Charguéraud and François Pottier. 2017. [Temporary Read-Only Permissions for Separation Logic](#). In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 260–286.
- Arthur Charguéraud and François Pottier. 2019. [Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits](#). *Journal of Automated Reasoning* 62, 3 (March 2019), 331–365.
- Clojure. 2021. Transient Data Structures. <https://clojure.org/reference/transients>.
- Sylvain Conchon and Jean-Christophe Filliâtre. 2008. [Semi-persistent Data Structures](#). In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 4960)*. Springer, 322–336.
- James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1989. [Making Data Structures Persistent](#). *J. Comput. System Sci.* 38, 1 (1989), 86–124.
- Armaël Guéneau. 2019. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*. Ph.D. Dissertation. Université de Paris.
- Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. [Iris from the ground up: A modular foundation for higher-order concurrent separation logic](#). *Journal of Functional Programming* 28 (2018), e20.
- Hannes Mehnert, Filip Sieczkowski, Lars Birkedal, and Peter Sestoft. 2012. [Formalized Verification of Snapshotable Trees: Separation and Sharing](#). In *Verified Software: Theories, Tools and Experiments (Lecture Notes in Computer Science, Vol. 7152)*. Springer, 179–195.
- Peter W. O’Hearn. 2019. [Separation logic](#). *Commun. ACM* 62, 2 (2019), 86–95.
- Alexandre Pilkiewicz and François Pottier. 2011. [The essence of monotonic state](#). In *Types in Language Design and Implementation (TLDI)*.
- François Pottier. 2008. [Hiding local state in direct style: a higher-order anti-frame rule](#). In *Logic in Computer Science (LICS)*. 331–340.
- François Pottier. 2017. [Verifying a hash table and its iterators in higher-order separation logic](#). In *Certified Programs and Proofs (CPP)*. 3–16.
- John C. Reynolds. 2002. [Separation Logic: A Logic for Shared Mutable Data Structures](#). In *Logic in Computer Science (LICS)*. 55–74.
- Robert Endre Tarjan. 1985. [Amortized Computational Complexity](#). *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318.
- Amin Timany and Lars Birkedal. 2021. [Reasoning about monotonicity in separation logic](#). In *Certified Programs and Proofs (CPP)*. 91–104.

## A Additional Specifications

### A.1 Copy Functions

This section presents two functions for creating copies of ephemeral stacks: one for making a totally independent copy in linear time, and one for making a copy with maximal sharing in constant time. These two functions admit the same functional specification, but lead to different time bounds.

```
(* Copy functions *)
val cp_without_sharing : 'a estack -> 'a estack
val cp_with_sharing    : 'a estack -> 'a estack
```

The function `cp_without_sharing` takes as input an ephemeral stack, and returns a totally independent ephemeral stack, which uniquely owns all of the chunks involved in its tail. This function copies all the elements and runs in linear time.

The function `cp_with_sharing` takes as input an ephemeral stack, and returns another ephemeral stack that shares with the first one all the chunks used to represent the tail. During this operation, the identifier of the input stack is replaced with a fresh identifier, to ensure that the input stack loses the unique ownership of all the chunks from its tail—these chunks are “migrated into the cold”. The copy-with-sharing operation only needs to copy elements from the front chunk, that is, at most  $K$  elements.

Both of these functions are useful. Indeed, a copy without sharing guarantees constant time complexity for set operations performed via iterators (see §6). A copy with sharing, on the contrary, allows to obtain two independent ephemeral structures without paying upfront a linear cost.

The two functions are specified as shown below.

```
Lemma cp_with_sharing_spec : ∀ e M L,
  SPEC (cp_with_sharing e)
  MONO M
  PRE $(3*K + 4) ★ e ∼ EStack M L
  POST (fun M' e' ⇒ e' ∼ EStack M' L ★ e ∼ EStack M' L).
```

```
Lemma cp_without_sharing_spec : ∀ e M L,
  SPEC (cp_without_sharing e)
  PRE $(4*(length L) + 2*K + 4)
  INV (Shared M ★ e ∼ EStack M L)
  POST (fun e' ⇒ e' ∼ EStack ∅ L).
```

### A.2 Preserving Conversions

The conversion operation `estack_to_pstack` destructs the ephemeral stack that it freezes. However, it is sometimes useful to take a persistent snapshot of an ephemeral structure without destroying it. To that end, we introduce the operation `estack_to_pstack_preserving`.

This operation returns a persistent stack with the same contents as the ephemeral stack, without modifying the logical contents of the ephemeral stack. Internally, the operation first builds a shallow copy of the ephemeral stack, then freezes the copy, thereby preserving the original. Concretely:

```
let estack_to_pstack_preserving e =
  estack_to_pstack (cp_with_sharing e)
```

The specification appears below. It follows from the specifications of `estack_to_pstack` and `cp_with_sharing`.

```
Lemma estack_to_pstack_preserving_spec : ∀ e M L,
  SPEC (estack_to_pstack_preserving e)
  MONO M
  PRE $(3*K + 7) ★ e ∼ EStack M L
  POST (fun M' p ⇒ \[PStack M' L p] ★ e ∼ EStack M' L).
```

### A.3 Specifications for Ephemeral Chunk

The specification of operations on ephemeral chunks are essentially the same as the specification of standard ephemeral stacks, with the only difference that the push operation features a precondition asserting that the chunk is not already full, reflecting the fact that chunks have a bounded capacity.

```
Definition IsFull A (L:list A) : Prop :=
  length L = K.
```

```
Lemma echunk_empty_spec : ∀ x,
  SPEC (echunk_empty a)
  PRE $(K+1)
  POST (fun c ⇒ c ∼ EChunk (@nil A)).
```

```
Lemma echunk_push_spec : ∀ L c x,
  ~ (IsFull L) →
  SPEC (echunk_push c x)
  PRE $(1 ★ c ∼ EChunk L)
  POST (fun _ ⇒ c ∼ EChunk (x::L)).
```

```
Lemma echunk_pop_spec : ∀ L c,
  L ≠ nil →
  SPEC (echunk_pop c)
  PRE $(3 ★ c ∼ EChunk L)
  POST (fun x ⇒ ∃ L', c ∼ EChunk L' ★ \[L = x::L']).
```

### A.4 Specifications for Shareable Chunks

We present below two specifications for two operations on shareable chunks.

The first specifications use the `SChunk` predicate, which captures the ownership of its support ephemeral chunk.

```
Lemma schunk_pop_spec : ∀ align S L s,
  L ≠ nil →
  SPEC (schunk_pop s)
  PRE $(3 ★ s ∼ SChunk align S L)
  POST (fun '(s',x) ⇒ ∃ L', s' ∼ SChunk false S L'
    ★ \[L = x::L' ∧ s'.support' = s.support'
    ∧ s'.owner' = s.owner']).
```

**Lemma** `schunk_push_spec` :  $\forall$  align S L s x,  
 $\sim$  IsFull L  $\rightarrow$   
SPEC (schunk\_push s x)  
PRE ( $\$(2 + \text{if align then } 0 \text{ else } K+2)$ )  
 $\star s \rightsquigarrow$  SChunk align S L)  
POST (fun s'  $\Rightarrow$   $\backslash$ [s'.owner' = s.owner'  $\vee$  s'.owner' = None])  
 $\star$  (if align  
then s'  $\rightsquigarrow$  SChunk align (x::S) (x::L)  
else hor  
(s'  $\rightsquigarrow$  SChunk align (x::S) (x::L)  
 $\star$   $\backslash$ [s.support' = s'.support'])  
(s  $\rightsquigarrow$  SChunk align S L  
 $\star$  s'  $\rightsquigarrow$  SChunk true (x::L) (x::L))))).

Note that `schunk_push_spec` uses the `hor` predicate, which is the Separation Logic disjunction (i.e., disjunction lifted to heap predicates).

Next we present specifications for the same functions, but in terms of the `SChunkShared` predicate, which does not own its support. These specifications are derived from the previous ones.

**Lemma** `schunk_pop_spec` :  $\forall$  M L s,  
SChunkShared M L s  $\rightarrow$   
L  $\neq$  nil  $\rightarrow$   
SPEC (schunk\_pop s)  
PRE ( $\$3$ )

INV (Shared M)  
POST (fun '(s',x)  $\Rightarrow$   
 $\exists$  L',  $\backslash$ [SChunkShared M L' s'  $\wedge$  L = x::L'  
 $\wedge$  s'.owner' = s.owner'])).

**Lemma** `schunk_push_spec` :  $\forall$  M L s x,  
SChunkShared M L s  $\rightarrow$   
 $\sim$  IsFull L  $\rightarrow$   
SPEC (schunk\_push s x)  
MONO M  
PRE ( $\$(K+4)$ )  
POST (fun M' s'  $\Rightarrow$   $\backslash$ [SChunkShared M' (x::L) s'  
 $\wedge$  (s'.owner' = s.owner'  $\vee$  s'.owner' = None)]).

## A.5 Common Specification for the Set Operation

In §6, we presented two specifications for the set operation. The two specifications are derived from one common specification, shown below, with respect to which the code is verified.

**Lemma** `set_spec` :  $\forall$  e st M L it i x,  
i  $\neq$  length L  $\rightarrow$   
SPEC (set it x)  
PRE ( $\$(\text{If } M = \emptyset \text{ then } 6 \text{ else } \text{length } L + K + 10)$ )  
 $\star e \rightsquigarrow$  EStackInState st M  $\star$  it  $\rightsquigarrow$  Iterator st i)  
INV (Shared M)  
POST (fun \_  $\Rightarrow$   $\exists$  st', e  $\rightsquigarrow$  EStackInState st' M (L[i:=x])  
 $\star$  it  $\rightsquigarrow$  Iterator st' i  $\star$   $\backslash$ [M =  $\emptyset$   $\rightarrow$  st = st'])).