Specification and Verification of a Transient Stack

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

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

François Pottier
Inria
Paris, France
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. 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.

CCS Concepts: • **Theory of computation → Separation logic**; **Program specifications**; **Program verification**; **Data structures design and analysis**.

Keywords: transient data structure, program verification

ACM Reference Format:
Alexandre Moine, Arthur Charguéraud, and François Pottier. 2022. Specification and Verification of a Transient Stack. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '22), January 17–18, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3497775.3503677

CPP '22, January 17–18, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9182-5/22/01…$15.00
https://doi.org/10.1145/3497775.3503677

1 Introduction

The algorithms literature establishes a distinction between ephemeral and persistent data structures [23]. 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 have been popularized by Clojure; they appear in the Scala standard library and in several libraries implemented in C++, Python, and JavaScript (§7). In all of these systems, one uses persistent data structures by default, but a persistent data structure offers the possibility of creating a "transient", that is, an ephemeral copy of the data structure. The "transient" supports 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 [13], 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 [1]. At a high level, a sequence is represented as a balanced tree whose leaves consist 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, such as the block device drivers of Linux; they also appear in the double-ended queues of the C++ STL.

Sek extends Acar et al.'s work [1] 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 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 [1].

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. It does remove the difficulty of reasoning about bootstrapped trees; however, this aspect has been addressed already by Charguéraud [14, §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 [17]. To carry out the proof, we use CFML [12], 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). Our code and proofs are publicly available [49].

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.[1] More precisely, some operations have complexity $O(1)$, while some have complexity $O(K)$, where the parameter K is the capacity

[1]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

of a chunk. The value of K is fixed by the user. We give precise complexity bounds as part of the specification in §4. The constant factors for ephemeral operations are smaller than for their persistent counterparts, motivating the use of ephemeral operations when performance matters.

Crucially, we provide constant-time conversions between ephemeral and persistent stacks: this is the hallmark 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 eagerly 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 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 grey 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 `data` to a fixed-capacity array of size K, (2) the index `top` of the first empty slot in the `data` 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 grey cloud.

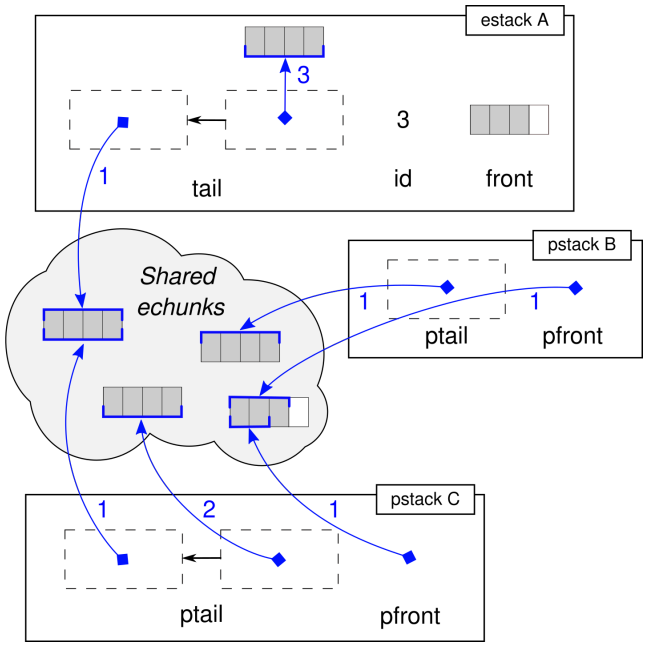

Figure 2. Internal representation of transient stacks

```
(* 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.

This view and ownership information 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 (the ownership policy

is described in detail further on), and (3) a view size.[2] 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 an ephemeral 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".

To summarize, an ephemeral chunk is either: (1) owned by an ephemeral stack, because it is the front chunk of the stack; or (2) owned by an ephemeral stack, because it is owned by a shareable chunk that is part of the tail of the stack; or (3) owned by 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. This is necessary in order to avoid memory leaks: keeping a pointer to an item that is no longer an element of the stack would prevent the GC from reclaiming this item. 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 (which is always a uniquely-owned 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 `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 grey (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

[2] The identifiers that determine ownership could also be stored in ephemeral chunks rather than in shareable chunks. This choice controls a tradeoff between minimizing the memory footprint of structures that involve a lot of sharing and decreasing the cost of accessing ownership identifiers.

```
(* [e ⤳ Stack L] is a notation for [Stack L e] *)
Definition Stack : list A → loc → hprop := ...

Lemma create_spec : ∀ d,
  SPEC (create d)
    PRE ($c1)
    POST (fun e ⇒  e ⤳ Stack nil).

Lemma push_spec : ∀ L e x,
  SPEC (push e x)
    PRE ($c2 ★  e ⤳ Stack L)
    POST (fun _ ⇒  e ⤳ Stack (x::L)).

Lemma pop_spec : ∀ L e,
  L ≠ nil →
  SPEC (pop e)
    PRE ($c3 ★  e ⤳ Stack L)
    POST (fun x ⇒  ∃ L',  \[L = x::L'] ★  e ⤳ Stack L').
```

Figure 4. Specification of a standard ephemeral stack

$O(1)$. 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 a monotonic update are not met, a copy must be performed: a fresh shared chunk must be initialized with suitable 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 $O(1)$. Any shareable chunk that was uniquely owned by this ephemeral stack is now regarded as shared. The ephemeral stack becomes logically invalid, and must no longer be used. Our specifications statically forbid the use of an invalid stack by a verified client: see §4.4.

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 [12], including the "dollar" notation for time credits. We introduce this syntax as we go.

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 ⤳ 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 $\{P\}(\texttt{create d})\{Q\}$.

The precondition of `create` is `$c1`: this means that `create` consumes `c1` time credits. A time credit [17] 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 has constant time complexity. In Figure 4, the definition of `c1` is not shown. A concrete implementation of stacks would fix the value of `c1`. We do so, for example, in Figure 6, where we instantiate `c1` with `K+4`. The postcondition of `create` is `fun e ⇒ e ⤳ Stack 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 ⤳ Stack nil` guarantees that this stack is valid and is empty.

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* [68].

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

Pop. In the specification of `pop`, the hypothesis `L ≠ nil` requires the stack to be nonempty.[3] The specification indicates that the state of the stack evolves from `e ⤳ Stack L` to `e ⤳ 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 `∃`. The equality `L = x::L'`, which has type `Prop`, is lifted to the level of assertions by the square brackets `\[...]`.

4 Specification of Transient Stacks

4.1 Abstract Predicates

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

[3] 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.

```
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

The set of shared chunks (that is, the "cloud" in Figure 2) is described by an assertion of the form `Shared M`. From the perspective of the user, `M` is an abstract entity: its type is `Memory`, a type whose definition is not revealed. (It is shown in §5.2.) 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 ⤳ EStack M L`, where `e` is a memory location and `L` is a list of items. Compared with `Stack` (§3), `EStack` takes an additional parameter `M`: this reflects the fact that an ephemeral stack is valid in relation to a certain shared state.

A persistent stack is described by a proposition (of type `Prop`) 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.

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 ⤳ Stack L` is replaced with `e ⤳ EStack M L` to account for the dependence of the data structure on a shared state. Second, the specification 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.[4] 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.

[4] **`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) {λy. (Q y) ∗ I}`.

```
Lemma ecreate_spec : ∀ M x,
 SPEC (ecreate x)
  PRE ($(K+4))
  POST (fun e ⇒ e ⤳ EStack M nil).

Lemma epush_spec : ∀ M L e x,
 SPEC (epush e x)
  PRE ($7 ★ e ⤳ EStack M L)
  INV (Shared M)
  POST (fun _ ⇒ e ⤳ EStack M (x::L)).

Lemma epop_spec : ∀ M L e,
 L ≠ nil →
 SPEC (epop e)
  PRE ($11 ★ e ⤳ EStack M L)
  INV (Shared M)
  POST (fun x ⇒ ∃ L', \[L = x::L'] ★ e ⤳ EStack M L').
```

Figure 6. Specification of ephemeral stacks

```
Lemma pcreate_spec : ∀ M x,
 SPEC (pcreate x)
  MONO M
  PRE ($(K+3))
  POST (fun M' p ⇒ \[PStack M' nil p]).

Lemma ppush_spec : ∀ M L p x,
 PStack M L p →
 SPEC (ppush p x)
  MONO M
  PRE ($(K+7))
  POST (fun M' p' ⇒ \[PStack M' (x::L) p']).

Lemma ppop_spec : ∀ M L p,
 PStack M L p →
 L ≠ nil →
 SPEC (ppop p)
  PRE ($7)
  INV (Shared M)
  POST (fun M' (p',x) ⇒ ∃ L',  \[L = x::L' ∧ PStack M' L' p']).
```

Figure 7. Specification of persistent stacks

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 ⤳ Stack L` as `e ⤳ EStack ∅ L`. The assertion `Shared ∅` happens to be equivalent to the "empty heap" assertion `\[]`, 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 contains the assertion `e ⤳ EStack M nil`, where `M` is chosen by the user: it may be a memory that is at hand, or the empty memory `∅`.

The precondition of `ecreate` requires $K + 4$ time credits, reflecting the cost of allocating and initializing an array of

```
Parameter Extend : Memory A → Memory A → Prop.

Lemma EStack_mon : ∀ e M M' L, Extend M M' →
 (e ⤳ EStack M L) ⊢ (e ⤳ EStack M' L).

Lemma PStack_mon : ∀ p M M' L, Extend M M' →
 (PStack M L p) → (PStack M' L p).
```

Figure 8. Monotonicity lemmas

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 [30], which develops techniques to handle formal asymptotic complexity with time credits. 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 existing ephemeral stack and persistent stacks: this is expressed by the two lemmas shown in Figure 8. There, the symbol ⊢ 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 : ∀ M L p x,
 PStack M L p →
 SPEC (ppush p x)
   PRE ($(K+7) ★ Shared M)
   POST (fun p' ⇒ ∃ M', \[Extend M M'] ★ Shared M'
                  ★ \[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.[5] The specifications in Figure 7 illustrate the use of this syntactic sugar.

[5]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 → hprop` and a preorder $\leq_T$. The notation **SPEC** `(f x)` **MONO** `M` **PRE** `P` **POST** `Q` stands for the triple {R M ★ P} (f x) {λ y. ∃ M', [M ≤$_T$ M'] ★ R M' ★ (Q M' y)}.

```
Lemma pstack_to_estack_spec : ∀ M L p,
 PStack M L p →
 SPEC (pstack_to_estack p)
   PRE ($(2*K+7))
   INV (Shared M)
   POST (fun e ⇒ e ⤳ EStack M L).

Lemma estack_to_pstack_spec : ∀ M L e,
 SPEC (estack_to_pstack e)
   MONO M
   PRE ($2 ★ e ⤳ EStack M L)
   POST (fun M' p ⇒ \[PStack M' L p]).
```

Figure 9. Specification of the conversions

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 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 predicates presented in §4 are defined and give insights about the key points of the proof.

5.1 Representation of Ephemeral Chunks

The predicate `c ⤳ 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[6] 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 `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`

[6]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 EChunk_inv
 (L:list A) (D:list A) (top:int) (default:A) : Prop :=
 { efront : ∀ i, 0 ≤ i < top → D[i] = (rev L)[i];
   etail : ∀ i, top ≤ i < K → D[i] = default;
   elength : length L = top ∧ length D = K;
   etop : 0 ≤ top ≤ K }.

Definition EChunk (L:list A) (c:echunk_ A) : hprop :=
 ∃ (data:loc) (D:list A) (top:int) (default:A),
   c ⤳ `{ data' := data; top' := top; default' := default }
 ★ data ⤳ Array D
 ★ \[EChunk_inv L D top default].
```

Figure 10. Representation of ephemeral chunks

```
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

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 §A.3).

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. We write $\emptyset$ for the empty memory, a record whose components are both empty.

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 $x \rightsquigarrow \mathtt{Ref}\ () * y \rightsquigarrow \mathtt{Ref}\ () \vdash x \neq y$ 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.[7]

[7] `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].
```

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 create an empty shared memory out of thin air, using the entailment `\[] ⊢ Shared ∅`. Although the specification of every operation 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: `Shared M1 ★ Shared M2 ⊢ ∃ M, Shared M ★ \[Extend M1 M ∧ 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 ⤳ SChunk 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 ⤳ SChunkUniquelyOwned L` denotes 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: 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 specifications of "push" and "pop" operations on shared chunks appear in Appendix A.4.

Finally, the assertion `s ⤳ SChunkMaybeOwned M id L` depicts 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 ⤳ SChunkUniquelyOwned L`. Otherwise, it is equivalent to `\[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.

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

```
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

```
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

```
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

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 ⇝ 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 ⇝ 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.

The combinator `ListOf`, a *higher-order representation predicate* [14], 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.[8]

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 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 [68]. 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 are part of a stack's internal representation.

[8] 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.
```

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. Our specification must impose this restriction. Yet, it cannot expose precisely when a copy takes 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.

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 [62] 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. (If the iterator already points to the last item, calling `move` makes it point to a virtual item that lies "one-past-the-end".) The operations `get` and `set` read and replace the current item. The operation `finished` returns `true` if the iterator points past the last item.

```
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

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.[9] 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 ⤳ 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 ⤳ EStack M L`, we introduce a new predicate, `e ⤳ EStackInState st M L`, where a name `st` appears. They are related by the equivalence law:

`(e ⤳ EStack M L) ⊣⊢ (∃ st, e ⤳ EStackInState st M L).`

A declarative reading of this law is that "a stack is a stack in some internal state". From a more operational point of view,

[9] 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 : ∀ st M L e,
 SPEC (iter_on_estack e)
  PRE ($2)
  INV (e ⤳ EStackInState st M L)
  POST (fun it ⇒ it ⤳ Iterator st 0).

Lemma finished_spec_ephemeral : ∀ st M L e it i,
 SPEC (finished it)
  PRE ($1)
  INV (e ⤳ EStackInState st M L ★ it ⤳ Iterator st i)
  POST (fun b ⇒ b = isTrue (i = length L)).

Lemma get_spec_ephemeral : ∀ st M L e it i,
 i ≠ length L →
 SPEC (get it)
  PRE ($3)
  INV (Shared M ★ e ⤳ EStackInState st M L
      ★ it ⤳ Iterator st i)
  POST (fun x ⇒ x = L[i]).

Lemma move_spec_ephemeral : ∀ st M L e it i,
 i ≠ length L →
 SPEC (move it)
  PRE ($4 ★ it ⤳ Iterator st i)
  INV (e ⤳ EStackInState st M L)
  POST (fun _ ⇒ it ⤳ Iterator st (i+1)).

Lemma set_spec_with_sharing : ∀ st M L e it i x,
 i ≠ length L →
 SPEC (set it x)
  PRE ($(length L + K + 10)
      ★ it ⤳ Iterator st i
      ★ e ⤳ EStackInState st M L)
  INV (Shared M)
  POST (fun _ ⇒ ∃ st', it ⤳ Iterator st' i
         ★ e ⤳ EStackInState st' M (L[i:=x])).

Lemma set_spec_without_sharing : ∀ st L e it i x,
 i ≠ length L →
 SPEC (set it x)
  PRE ($6 ★ e ⤳ EStackInState st ∅ L)
  INV (it ⤳ Iterator st i)
  POST (fun _ ⇒ e ⤳ EStackInState st ∅ (L[i:=x])).
```

Figure 17. Specification of iterators

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 to name 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.

To call `push e` or `pop e`, one must provide the assertion `e ⤳ EStack M L`. If an assertion `e ⤳ EStackInState st M L` 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 an update of this list at index `i`. An operation on an iterator `it` requires `it ⤳ Iterator st i` and `e ⤳ EStackInState st M L`, 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 mentioned 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 $O(1)$ and does *not* invalidate concurrent iterators, as the internal state remains unmodified.

As long as one works with an ephemeral stack without performing any conversion or copy-with-sharing operation, the shared state `M` remains empty, so every `set` operation is guaranteed to be cheap and to preserve the validity of all existing iterators.

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

7.1 Verification Frameworks and Techniques

Verification Frameworks. A traditional approach to program verification involves annotating the code with specifications and invariants, using a tool (based on Hoare logic) to produce proof obligations, then transmitting these proof obligations to automated theorem provers. Müller and Shankar [51] provide a survey of this "deductive" form of program verification. Practical systems include Boogie [6], KeY [3], and Why3 [25].

Separation Logic [64] improves on Hoare Logic by allowing more modular forms of reasoning about mutable state. O'Hearn [57] gives a comprehensive survey of its applications. Separation Logic can be exploited by automated

and semi-automated tools, such as Infer [10], VeriFast [38], and Viper [50], or embedded in interactive proof assistants. For example, CFML [15] and Iris [39] are embedded in Coq; Steel [26] is embedded in ***F**** [67].

Another approach to program verification involves first verifying a pure program, expressed in the logic of a proof assistant. Then, out of this high-level code, one extracts executable code, expressed either in a functional programming language (such as OCaml or Haskell) or in a low-level imperative language. In some cases, this refinement process is guided by the user and involves interactive proofs. For example, Bedrock [20] refines Coq definitions down to assembly language; the Isabelle Refinement Framework refines Isabelle/HOL code down to imperative SML [42, 43] or LLVM IR [33].

Verification of Time Bounds. Worst-case bounds on execution time can sometimes be obtained via an automated analysis [37]. However, fully automated analyses are often limited to polynomial or poly-log bounds, and usually cannot be applied to algorithms that involve a nontrivial complexity argument.

These limitations may be overcome by performing interactive proofs. *Time credits* are a lightweight yet very expressive extension to Separation Logic that allows establishing worst-case time complexity bounds. Time credits, introduced in Separation Logic by Atkey [4] and in a type system by Pilkiewicz and Pottier [59], have been formalized in Coq and used in practice (in CFML) by Charguéraud and Pottier [16, 17]. Several extensions of time credits have been studied. Guéneau [30] introduces negative time credits and investigates the use of the big-O notation at scale. Mével et al. [52] introduce time receipts, a related notion, and encode them in Iris. Haslbeck and Lammich [33] introduce multiple currencies of time credits. Haslbeck and Nipkow [34] compare the expressiveness of several program logics that can establish time complexity bounds.

Verification Involving Monotonicity Arguments. Our specification explicitly exposes the existence of a shared internal state, whose nature remains abstract, and whose evolution is monotonic. This is a common pattern. Monotonic state has been studied in several contexts, beginning with ghost monotonic references [59] or simple type systems equipped with monotonic references [27], and continuing with more powerful dependent type systems and program logics, such as ***F**** [2] and Iris [69].

It is also common practice to *hide* the existence of this internal state. In a sequential setting, this can be done by using the anti-frame rule [61]. In a concurrent setting, this can be done by creating an Iris invariant [39, 69]. CFML does not allow hiding an internal state in such a way. This may be less convenient for end users, but we do not believe that this is a severe problem. Indeed, 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 it was hidden, we would have to record the absence or presence of sharing in some other way.

7.2 Verification of Data Structures

Verification of Data Structures. There is a large panel of work on the verification of data structures and algorithms. Nipkow et al. [56] present a survey that focuses on verified textbook algorithms. Here, we highlight a few results, so as to give an idea of the kind of data structures that state-of-the-art program verification tools can handle.

In the area of purely functional data structures, Sozeau [65] formalizes finger trees in Coq, and Danielsson [22] uses Agda to verify the amortized time complexity of the `dequeue` operation of finger trees, using Okasaki's variant of the banker's method [58] to annotate each thunk with its cost. Charguéraud [11] uses CFML to verify about half of Okasaki's book [58], which contains a large collection of purely functional data structures. Chen et al. [18] verify the FSCQ file system in Coq, and extract executable Haskell code. More recently, Nipkow et al. [54] use Isabelle/HOL to verify a number of textbook functional algorithms, and introduce manually-crafted cost functions to reason about time complexity [53, 55].

In the realm of imperative data structures, Chlipala [19] uses Bedrock to verify hash tables and binary search trees. Charguéraud and Pottier [17] use CFML to verify Union-Find, including its amortized complexity bound; Guéneau et al. [31] use it to verify a state-of-the-art incremental cycle detection algorithm. The Isabelle Refinement Framework is used by Lammich [44] to verify Introsort (which is used in the GNU C++ Standard Library) and Pdqsort. Mohan et al. [48] use VST to verify C implementations of Dijkstra's, Kruskal's, and Prim's algorithms. Their work includes the verification of a priority queue, including the bottom-up construction of the heap and the operations for increasing or decreasing a key.

Verification of Iterators on Collections. Containers, also known as collections, are an important class of data structures. A ubiquitous operation consists in iterating over the elements of a collection. This operation can be presented to the end user in various guises such as first-order iterators and higher-order iteration schemes. Regardless of which is chosen, writing a sound, expressive specification for this operation is surprisingly tricky. A number of authors have worked on the specification of iterators, including Krishnaswami et al. [41], Haack and Hurlin [32], Lammich and Meis [45], Polikarpova et al. [60], Filliâtre and Pereira [24], and Pottier [62]. The manner in which we describe iterator invalidation in the specification, using an abstract state token, follows Pottier's work [62].

Verification of Persistent Data Structures. Our persistent stacks are *fully persistent*: any version can be read

and updated, producing a new version. There exist several weaker notions of persistence. Driscoll et al. [23] describe *partial persistence*, where old versions can be read but not updated. Conchon and Filliâtre [21] describe *semi-persistence*, which allows read and write access only to the current version (that is, the most recently created version) and to its ancestors. They propose an ad hoc program logic, equipped with an automated decision procedure, to check that a client program makes legal use of a semi-persistent data structure.

Mehnert et al. [47] use Separation Logic to specify and verify a Java implementation of "snapshotable trees". These trees support read-only snapshots. They also support iterators, which cannot mutate the underlying tree and are invalidated by an update of this tree. The authors use an abstract predicate that describes one tree together with all of its snapshots and iterators. The shared state is hidden in this abstract predicate and is not visible to the end user. The 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.

7.3 Chunk-Based and Transient Data Structures

Data Structures involving Chunks. Storing elements in *chunks*, rather than storing them individually, allows for close-to-optimal memory usage and leads to greatly improved constant factors in the time complexity of many operations. For example, the C++ STL includes implementations of *deques* represented as circular vectors of fixed-sized chunks and *ropes* represented as binary trees of variable-sized chunks. Chunks can also be used in tree-shaped data structures at the leaves or at every node.

Catenable and Splittable Sequence Data Structures. A general-purpose representation of *strings* (or sequences) should offer efficient operations for sequential and random access, for concatenating two strings, and for extracting substrings (or, equivalently, splitting strings). One challenge is to design a data structure with good asymptotic complexity. A further challenge is to design one that moreover has good performance in practice.

Regarding the first challenge, Kaplan and Tarjan [40] present *functional catenable sorted lists*, which support "push" and "pop" operations in time $O(1)$, while also supporting splitting and concatenation in logarithmic time. Their structure is an instance of a finger search tree, a type of tree that has been extensively studied since the 1970s [28]. Hinze and Paterson [36] present *finger trees*, achieving similar bounds with a simpler implementation.

In the aforementioned structures, constant-time operations suffer from large constant factors. Acar et al. [1] measure that, when pushing and popping integer values, a carefully optimized C++ implementation of finger trees is over 20 times slower than an STL deque. To tame these constant factors, the Haskell package "yi" [7] instantiates finger trees with chunks of characters in the leaves. However, the repeated concatenation of small strings results in a degenerate tree where each chunk stores only two items [1, §2].

Acar et al. [1] present an ephemeral *chunked sequence* that achieves constant time "push" and "pop" operations at either end, logarithmic-time splitting and concatenation, and requires space $(2 + O(1)/K) \cdot n$ in the worst case to store n elements.

Transient Data Structures. *Transience* has been popularized by the Clojure programming language [35], based on the seminal ideas of Bagwell [5]. A definition of transience is given by L'Orange [46]: in his view, a collection is persistent by default, but the programmer can work with a *transient* version of the collection, that is, an ephemeral copy. The motivation for transience is to support efficient batched updates.

We give a slightly different, yet essentially equivalent, definition of transience: a *transient* data structure is one that offers an ephemeral variant, a persistent variant, and efficient conversions between the two.

Clojure's transients have inspired the authors of many other programming languages and libraries. Transient vectors and hash-maps can be found in Scala's standard library, in the JavaScript library `immutable.js` [9], in the Haskell library `persistent-vectors` [63], and in the Python library `pyrsistent` [29].

Relaxed Radix Balanced (RRB) vectors and trees [66] provide logarithmic-time `split` and `concat` operations. RRB vectors have been implemented in the C++ library `immer` by Bolívar Puente [8].

We are not aware of other work verifying the correctness or complexity of a transient data structure.

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.

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 could". 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 Conversions That Preserve Their Argument

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 Chunks

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 : ∀ align S L s x,
 ~ IsFull L →
 SPEC (schunk_push s x)
  PRE ($(2 + if align then 0 else K+2)
      ★ s ⤳ SChunk align S L)
  POST (fun s' ⇒ \[s'.owner' = s.owner' ∨ s'.owner' = None]
    ★ (if align
       then s' ⤳ SChunk align (x::S) (x::L)
       else hor
         (s' ⤳ SChunk align (x::S) (x::L)
          ★ \[s.support' = s'.support'])
         (s ⤳ SChunk align S L
          ★ s' ⤳ 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 : ∀ M L s,
 SChunkShared M L s →
 L ≠ nil →
 SPEC (schunk_pop s)
  PRE ($3)
  INV (Shared M)
  POST (fun '(s',x) ⇒
   ∃ L', \[SChunkShared M L' s' ∧ L = x::L'
        ∧ s'.owner' = s.owner']).

Lemma schunk_push_spec : ∀ M L s x,
 SChunkShared M L s →
 ~ IsFull L →
 SPEC (schunk_push s x)
  MONO M
  PRE ($(K+4))
  POST (fun M' s' ⇒ \[SChunkShared M' (x::L) s'
        ∧ (s'.owner' = s.owner' ∨ s'.owner' = None)] ).
```

A.5 A 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 : ∀ e st M L it i x,
 i ≠ length L →
 SPEC (set it x)
  PRE ($(If M = ∅ then 6 else length L + K + 10)
     ★ e ⤳ EStackInState st M L ★ it ⤳ Iterator st i)
  INV (Shared M)
  POST (fun _ ⇒ ∃ st', e ⤳ EStackInState st' M (L[i:=x])
               ★ it ⤳ Iterator st' i ★ \[M = ∅ → st = st']).
```

References

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2014. Theory and Practice of Chunked Sequences. In *Algorithms (ESA)*, Vol. 8737. Springer, 25–36.

[2] 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.

[3] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. 2014. The KeY Platform for Verification and Analysis of Java Programs. In *Verified Software: Theories, Tools and Experiments*, Dimitra Giannakopoulou and Daniel Kroening (Eds.). Springer International Publishing, Cham, 55–71.

[4] Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* 7, 2:17 (2011).

[5] Phil Bagwell. 2001. *Ideal Hash Trees*. Technical Report. EPFL.

[6] Mike Barnett, Bor-Yuh Evan Chang, Rob DeLine, and Bart Jacobs. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*. Springer.

[7] Jean-Philippe Bernardy. 2021. The Haskell yi package. http://hackage.haskell.org/package/yi-0.6.2.3/docs/src/Data-Rope.html.

[8] Juan Pedro Bolívar Puente. 2017. Persistence for the masses: RRB-vectors in a systems language. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 16:1–16:28.

[9] Lee Byron. 2021. *Immutable.js library for JavaScript*.

[10] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (NFM) (Lecture Notes in Computer Science, Vol. 6617)*. Springer, 459–465.

[11] Arthur Charguéraud. 2010. Program Verification Through Characteristic Formulae. In *International Conference on Functional Programming (ICFP)*. 321–332.

[12] Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *International Conference on Functional Programming (ICFP)*. 418–430.

[13] Arthur Charguéraud and François Pottier. 2021. *Sek*.

[14] Arthur Charguéraud. 2016. Higher-order representation predicates in separation logic. In *Certified Programs and Proofs (CPP)*. 3–14.

[15] Arthur Charguéraud. 2021. The CFML tool and library. http://www.chargueraud.org/softs/cfml/.

[16] Arthur Charguéraud and François Pottier. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science, Vol. 9236)*. Springer, 137–153.

[17] 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.

[18] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Symposium on Operating Systems Principles (SOSP)*. 18–37.

[19] Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *Programming Language Design and Implementation (PLDI)*. 234–245.

[20] Adam Chlipala. 2013. The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In *International Conference on Functional Programming (ICFP)*. 391–402.

[21] 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.

[22] Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Principles of Programming Languages (POPL)*.

[23] 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.

[24] Jean-Christophe Filliâtre and Mário Pereira. 2016. A Modular Way to Reason About Iteration. In *NASA Formal Methods (NFM) (Lecture Notes in Computer Science, Vol. 9690)*. Springer, 322–336.

[25] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—Where Programs Meet Provers. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 125–128.

[26] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021.

Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–30.

[27] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. 2013. Rely-Guarantee References for Refinement Types over Aliased Mutable Data *(PLDI '13)*. 73–84.

[28] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. 1977. A New Representation for Linear Lists *(STOC '77)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/800105.803395

[29] Tobias Gustafsson. 2021. *Pyrsistent library for Python*.

[30] Armaël Guéneau. 2019. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*. Ph.D. Dissertation. Université de Paris.

[31] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. 2019. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *Interactive Theorem Proving (ITP) (Leibniz International Proceedings in Informatics, Vol. 141)*. 18:1–18:20.

[32] Christian Haack and Clément Hurlin. 2009. Resource Usage Protocols for Iterators. *Journal of Object Technology* 8, 4 (2009), 55–83.

[33] Maximilian P. L. Haslbeck and Peter Lammich. 2021. For a Few Dollars More - Verified Fine-Grained Algorithm Analysis Down to LLVM. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 12648)*. Springer, 292–319.

[34] Maximilian P. L. Haslbeck and Tobias Nipkow. 2018. Hoare Logics for Time Bounds: A Study in Meta Theory. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 10805)*. Springer, 155–171.

[35] Rich Hickey. 2006. The Clojure programming language. https://clojure.org/.

[36] Ralf Hinze and Ross Paterson. 2006. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming* 16, 2 (2006), 197–217.

[37] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Principles of Programming Languages (POPL)*. 359–373.

[38] Bart Jacobs and Frank Piessens. 2008. *The VeriFast Program Verifier*. Technical Report CW-520. Department of Computer Science, Katholieke Universiteit Leuven.

[39] 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.

[40] Haim Kaplan and Robert E. Tarjan. 1996. Purely Functional Representations of Catenable Sorted Lists *(STOC '96)*. ACM, New York, NY, USA, 202–211. https://doi.org/10.1145/237814.237865

[41] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. 2009. Design Patterns in Separation Logic. In *Types in Language Design and Implementation (TLDI)*. 105–116.

[42] Peter Lammich. 2016. Refinement Based Verification of Imperative Data Structures. In *Certified Programs and Proofs (CPP)*. 27–36.

[43] Peter Lammich. 2019. Refinement to Imperative HOL. *Journal of Automated Reasoning* 62, 4 (April 2019), 481–503.

[44] Peter Lammich. 2020. Efficient Verified Implementation of Introsort and Pdqsort. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 307–323.

[45] Peter Lammich and Rene Meis. 2012. A Separation Logic Framework for Imperative HOL. *Archive of Formal Proofs* (2012).

[46] Jean Niklas L'Orange. 2014. *Improving RRB-Tree Performance through Transience*. Master's thesis. Department of Computer and Information Science, Norwegian University of Science and Technology. https://hypirion.com/thesis.pdf.

[47] 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.

[48] Anshuman Mohan, Wei Xiang Leow, and Aquinas Hobor. 2021. Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 12760)*. Springer, 801–826.

[49] Alexandre Moine, Arthur Charguéraud, and François Pottier. 2021. *Specification and verification of a transient stack: source code and proofs*.

[50] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering*. IOS Press, 104–125.

[51] Peter Müller and Natarajan Shankar. 2021. The First Fifteen Years of the Verified Software Project. In *Theories of Programming: The Life and Works of Tony Hoare*, Cliff B. Jones and Jayadev Misra (Eds.). ACM / Morgan & Claypool, 93–124.

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

[53] Tobias Nipkow. 2015. Amortized Complexity Verified. In *Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science, Vol. 9236)*. Springer, 310–324.

[54] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer, and Bohua Zhan. 2021. *Functional Algorithms, Verified!*

[55] Tobias Nipkow and Hauke Brinkop. 2019. Amortized Complexity Verified. *Journal of Automated Reasoning* 62, 3 (2019), 367–391.

[56] Tobias Nipkow, Manuel Eberl, and Maximilian P. L. Haslbeck. 2020. Verified Textbook Algorithms: a Biased Survey. In *Automated Technology for Verification and Analysis (ATVA) (Lecture Notes in Computer Science, Vol. 12302)*. Springer, 25–53.

[57] Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95.

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

[59] Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Types in Language Design and Implementation (TLDI)*.

[60] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. 2015. A Fully Verified Container Library. In *Formal Methods (FM) (Lecture Notes in Computer Science, Vol. 9109)*. Springer, 414–434.

[61] François Pottier. 2008. Hiding local state in direct style: a higher-order anti-frame rule. In *Logic in Computer Science (LICS)*. 331–340.

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

[63] Tristan Ravitch. 2020. *Persistent Vector library for Haskell*.

[64] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. 55–74.

[65] Matthieu Sozeau. 2007. Program-ing Finger Trees in Coq. In *International Conference on Functional Programming (ICFP)*. 13–24.

[66] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB vector: a practical general purpose immutable sequence. In *International Conference on Functional Programming (ICFP)*. 342–354.

[67] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages (POPL)*. 256–270.

[68] Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318.

[69] Amin Timany and Lars Birkedal. 2021. Reasoning about monotonicity in separation logic. In *Certified Programs and Proofs (CPP)*. 91–104.