Specification and Verification of a Transient Data Structure
Research internship proposal, M2

Arthur Charguéraud, Inria, Strasbourg, arthur.chargueraud@inria.fr
François Pottier, Inria, Paris, francois.pottier@inria.fr

2020–2021

1 Overview

With an ephemeral data structure, updates are destructive, meaning that prior versions of the data structure are lost after an operation. On the contrary, with a persistent data structure, all prior versions of the data structure remain valid after an update. A *transient data structure* is a persistent data structure that may be temporarily accessed in an ephemeral fashion, enabling performance gains.

In recent work, Arthur Charguéraud and François Pottier developed Sek [4], a transient data structure for sequences. This data structure offers, at the same time:

- ephemeral sequences, which support in-place updates;
- persistent sequences, which appear to be immutable, even though their implementation can involve mutable state and copy-on-write techniques;
- constant-time operations for converting between these two variants.

The purpose of this internship is to to formally verify the key ingredients of this transient data structure in Separation Logic. This work, if successful, would presumably be the first formal verification of a persistent data structure implemented with optimized imperative code.

2 Key Ideas of the Implementation

To focus on the core of the verification challenge, we propose to consider a simpler data structure, namely a transient stack data structure. This greatly reduces the size of the code and simplifies numerous technical details, while preserving the most interesting aspects of the problem, which have to do with managing the ownership of mutable memory blocks.

A stack is represented as a linked list of *chunks*, where a chunk is a fixed-capacity array. The OCaml data types involved in the representation of stacks are given below and explained next.

```
type 'a  chunk =
  { mutable data : 'a array;
    mutable size : int;
    default : 'a  }
type 'a pchunk =
  { support : 'a chunk;
    mutable view_size : int;
```

```
    version : int }
type 'a pstack =
  { pfront : 'a pchunk;
    ptail : 'a pchunk list;
    pversion_max : int }
type 'a stack =
  { mutable front : 'a chunk;
    mutable tail : 'a pchunk list;
    version_owned : int }
```

- A *chunk* (type `chunk`) consists of a fixed capacity array `data`, and of a `size` field that indicates how many elements are currently stored in this array. The unused cells store a default value. An ephemeral chunk also keeps at hand a `default` value that is used to overwrite a cell when it becomes conceptually empty. Chunks serve two purposes: first, they serve as stand-alone *ephemeral chunks*, which are pieces of an ephemeral stack; second, they serve as the *support* of persistent chunks, which are potentially-shared pieces of persistent stacks.

- A *persistent chunk* (type `pchunk`) consists of a "view" on a chunk, that is, a description of a prefix of a chunk. Thus, a persistent chunk carries a pointer to a chunk, its `support`, and an integer `view_size`, which indicates the size of the prefix of the support that is relevant. Finally, a persistent chunk carries a `version` number. This number is used by ephemeral stacks to keep track of which persistent chunks they own. Indeed, a persistent chunk is either definitely uniquely owned by a stack, or potentially shared between several stacks.

- A *persistent stack* (type `pstack`) essentially consists of a list of persistent chunks. For efficiency reasons, the front persistent chunk is stored in a separate field, `pfront`, while the remaining persistent chunks are stored in a linked list, `ptail`. A persistent stack also stores an upper bound `pversion_max` on the version numbers of its persistent chunks. This upper bound allows generating a fresh version number in constant time.

- An *ephemeral stack* (type `stack`) consists of a `front` chunk, which can be efficiently updated in place, and of a linked list `tail` of persistent chunks. A number of these persistent chunks may be uniquely owned by the ephemeral stack, in which case they keep the ability to be updated in place when they come to the front. The remaining persistent chunks are regarded as potentially shared, therefore cannot be modified; if they must be updated, then they are copied first (a copy-on-write operation). To keep track of which persistent chunks it owns, an ephemeral stack stores a version number, `version_owned`. The persistent chunks whose version number matches this number are uniquely owned.

A strength of this representation is that one can convert a stack into a persistent stack in time $O(K)$, where K denotes the capacity of the chunks. K is typically a small constant such as 16 or 32.

An implementation of all operations is given in the appendix.

3 Proof Strategy

The CFML system [5] supports the verification of OCaml programs through interactive proofs in the Coq proof assistant. To reason about mutable state, CFML leverages Separation Logic, which extends Hoare Logic with "local reasoning".

CFML has been used in the past decade to verify numerous data structures (vectors, hash tables, disjoint set forests, finger trees, etc.) and algorithms (Dijkstra's shortest paths algorithm, depth-first search, Eratosthenes' sieve, incremental cycle detection, etc.).

A central aspect of the internship is to express in CFML both the high-level principles of the chunk ownership policy and the low-level details of the management of version numbers. The key challenges of the proof include arguing that the data structures invariants are preserved by in-place updates and explaining the owership transfers that take place during the conversion operations.

4 Roadmap

Here is a possible roadmap for this internship.

1. As a warm-up for using CFML: verify the correctness of the data structure, restricted to its ephemeral interface. Then, verify the correctness of the data structure, restricted to its persistent interface.

2. As another, independent warm-up for specifying the data structure: specify and test the full data structure using Monolith [12], a tool for applying *fuzz testing* to an OCaml library. Doing so should help understand exactly what each operation is supposed to do, and which invariants are true.

3. Specify the data structure operations and invariants in the presence of both ephemeral and persistent instances. This requires explaining which pieces of data are uniquely owned, and which pieces are shared.

4. Verify the correctness of the implementation with respect this specification. This includes the verification of the ephemeral operations, of the persistent operations, and of the conversion operations.

5. Write and publish a research paper describing the work.

5 Extensions

There are several immediate directions for extending this work.

1. Generalize the chunks from a stack interface to a double-ended queue interface, and generalize the data structure from a list of chunks to a tree of chunks. This would result in a data structure that is much closer to what is actually found in Sek.

2. Generalize the specification and proofs to establish asymptotic time complexity bounds. Doing so can be achieved by leveraging the extension of CFML with “time credits” [6, 7].

3. Investigate how the proof could be ported to Iris [8], a very powerful evolution of Separation Logic. Exploiting Iris could lead to more concise specifications.

6 Prerequisites

Familiarity with the operational semantics of programming languages (MPRI 2.4), with Hoare logic and Separation Logic (MPRI 2.36.1), and with proof assistants (MPRI 2.7.1 and 2.7.2) is essential. A solid programming background, including fluency in OCaml, is also highly desirable.

7 Practical details

This internship will be co-supervised by Arthur Charguéraud and by François Pottier. It will take place from March 2021 to August 2021 approximately. It can take place *either* in Paris or in Strasbourg. Regardless of which physical location is chosen, regular video meetings with both advisors will be scheduled.

8 Reading list

To learn about Separation Logic for sequential programs, and about the foundations of CFML2, an obvious step is to read Charguéraud's educational pearl [3], as well as the associated all-in-Coq course material [2].

Learning about the logical foundations of Iris [8] is recommended.

Mehnert *et al.*'s paper on the verification of a snapshottable data structure [10] may be relevant. Krishna *et al.*'s approach to local reasoning about the global properties of a graph [9] should also be interesting.

For broader information on Separation Logic and its many variants, the surveys by O'Hearn [11] and by Brookes and O'Hearn [1] are recommended.

References

[1] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016.

[2] Arthur Charguéraud. Foundations of Separation Logic. August 2020. To-be-released volume from the Software Foundations series.

[3] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[4] Arthur Charguéraud, François Pottier, and Émilie Guermeur. Sek. `https://gitlab.inria.fr/fpottier/sek/`, 2020.

[5] Arthur Charguéraud. The CFML tool and library. `http://www.chargueraud.org/softs/cfml/`, 2019.

[6] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *European Symposium on Programming (ESOP)*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, April 2018.

[7] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal proof and analysis of an incremental cycle detection algorithm. In *Interactive Theorem Proving (ITP)*, volume 141 of *Leibniz International Proceedings in Informatics*, pages 18:1–18:20, September 2019.

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

[9] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. Local reasoning for global graph properties. In *European Symposium on Programming (ESOP)*, volume 12075 of *Lecture Notes in Computer Science*, pages 308–335. Springer, April 2020.

[10] Hannes Mehnert, Filip Sieczkowski, Lars Birkedal, and Peter Sestoft. Formalized verification of snapshotable trees: Separation and sharing. In *Verified Software: Theories, Tools and Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 179–195. Springer, January 2012.

[11] Peter W. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[12] François Pottier. Monolith. `https://gitlab.inria.fr/fpottier/monolith/`, 2020.

A Implementation

```
(* Auxiliary functions. *)

let empty_chunk d =
  { data = Array.make capacity d; size = 0; default = d }

let empty_pchunk d =
  { support = empty_chunk d; view_size = 0; version = 0 }

let chunk_of_pchunk p =
  let d = p.support.default in
  let t = Array.init capacity (fun i ->
    if i < p.view_size then p.support.data.(i) else d
  ) in
  { data = t; size = p.view_size; default = d }

let chunk_push c x =
  c.data.(c.size) <- x;
  c.size <- c.size + 1

(* Conversion functions. *)

let persistent_to_ephemeral s =
  { front = chunk_of_pchunk s.pfront; tail = s.ptail; version_owned = s.pversion_max + 1 }

let ephemeral_to_persistent s =
  let p = { support = s.front; view_size = s.front.size; version = s.version_owned } in
  { pfront = p; ptail = s.tail; pversion_max = s.version_owned }

(* Ephemeral operations. *)

let empty d =
  { front = empty_chunk d; tail = []; version_owned = 0 }

let push s x =
  if s.front.size = capacity then begin
    let p = { support = s.front;
              view_size = capacity;
              version = s.version_owned } in
    s.tail <- p :: s.tail;
    s.front <- empty_chunk s.front.default;
  end;
  chunk_push s.front x

let pop s =
  let n = s.front.size in
  if n = 0 then raise Not_found;
  let n' = n - 1 in
  s.front.size <- n';
  let x = s.front.data.(n') in
  s.front.data.(n') <- s.front.default;
  if n' = 0 then begin
    match s.tail with
```

```
    | [] -> ()
    | p::ps ->
        s.tail <- ps;
        s.front <- if p.version = s.version_owned
                   then p.support
                   else chunk_of_pchunk p
  end;
  x

(* Persistent operations. *)

let pempty d =
  { pfront = empty_pchunk d; ptail = []; pversion_max = 0 }

let ppush s x =
  if s.pfront.view_size = capacity then begin
    let c = empty_chunk s.pfront.support.default in
    chunk_push c x;
    let p = { support = c; view_size = 1; version = 0 } in
    { s with pfront = p; ptail = s.pfront :: s.ptail }
  end else begin
    let p = s.pfront in
    let n = p.view_size in
    if n = p.support.size then begin
      chunk_push p.support x;
      let p' = { p with view_size = n+1 } in
      { s with pfront = p' }
    end else begin
      let c = chunk_of_pchunk p in
      chunk_push c x;
      let p' = { support = c; view_size = n+1; version = 0 } in
      { s with pfront = p' }
    end
end

let ppop s =
  let n = s.pfront.view_size in
  if n = 0 then raise Not_found;
  let n' = n - 1 in
  let x = s.pfront.support.data.(n') in
  let p' = { s.pfront with view_size = n' } in
  let s' =
    if n' > 0 then
      { s with pfront = p' }
    else
      match s.ptail with
      | [] -> { s with pfront = p' }
      | p::ps -> { s with pfront = p; ptail = ps }
    in
  s', x
```