Mechanized Verification of Fine-grained Concurrent Programs in Fine-grained Concurrent Separation Logic

User Manual and Code Commentary

Aleksandar Nanevski[1] Ilya Sergey[2] Ruy Ley-Wild[3]
Anindya Banerjee[1] Germán Andrés Delbianco[1,4]

[1]IMDEA Software Institute, Spain [2]University College London, UK
[3]LogicBlox, USA [4]Universidad Politécnica de Madrid, Spain

April 19, 2017

This manual describes the design and implementation of Fine-grained Separation Logic (FCSL) [5] as well as the case studies conducted in it [1, 9–11].

Section 1 of this manuscript lists the prerequisites for reading and executing FCSL code, Section 2 outlines the structure of the FCSL project, and Section 3 provides general instructions on compiling FCSL and measuring the line counts of the development. Section 4 serves as a guide through the mechanized implementation of the case studies.

Contents

1. Prerequisites

In the remainder of this document, we assume that the reader is using the provided VirtualBox VM image or has successfully installed Coq/Ssreflect as well as GNU Emacs/Aquamacs and Proof General. The VM image and the set-up instructions for the artifact are available by the following URL:

`http://software.imdea.org/fcsl`

In the case of the VM image, you can find the code of the project in the folder `~/fcsl-ecoop17` (use `fcsl` as username and password whenever required). We do not assume the reader to have any familiarity with Coq/Ssreflect to browse through the examples. In the case if the reader is feeling curious to experiment on her own and modify the proof scripts, we recommend to take a look at the lecture notes [8] for the basics of interactive proof development in Coq/Ssreflect.

For the description of semantic foundations of FCSL in prose, we address the reader to the appendices of the original FCSL paper [5].

2. Structure of the development

The project's structure is completely flat, with all files located currently in the same folder (`fcsl`). Besides the `README` file, which summarizes the contents of particular files, and the `Makefile`, the folder contains the files, corresponding to case studies, and to the core development of the FCSL logic itself. Below, we provide brief descriptions for the most interesting parts of the source codebase.

2.1 Case Studies

The following files implement the case studies/example programs, together with their specifications and verification scripts (more details are available in the corresponding referred sections further in the document).

2.1.1 Basic examples

Basic examples, described in the initial work on FCSL [5], described in detail in Section 4.1.

Folder: `Examples/Basic`

- `locks.v` — generic interface for locking structures (§4.1.1).
- `caslock.v` — an implementation of a CAS-based spin lock concurroid, its actions, lock/unlock procedures and lemmas for Hoare-style reasoning (§4.1.2).
- `ticketed.v` — An implementation of a ticketed lock concurroid, its actions, lock/unlock procedures and lemmas for Hoare-style reasoning (§4.1.3).
- `incrementor.v` — a toy example of a coarse-grained incrementor (§4.1.4).
- `alloc.v` — a simple coarse-grained memory allocator, parametrized by a lock implementation (§4.1.5).

2.1.2 Linearizable concurrent objects and universal constructions

Examples, motivating the use of subjective auxiliary histories, presented in the work [10], described in Section 4.2.

Folder: `Examples/Histories`

- `readpair.v` — an implementation of an atomic pair-snapshot data structure (§4.2.1).
- `treiber.v` — an implementation of Treiber's non-blocking stack (§4.2.2).
- `stack_framed.v` — application of FCSL's frame rule applied to the Treiber stack specifications (§4.2.2)
- `stack_seq.v` — a derivation of a sequential specification for the Treiber stack "push" and "pop" operations (§4.2.3).
- `stack_client.v` — a simple stack client: two-thread producer/consumer (§4.2.4).
- `flatcombine.v` — a generic implementation of the flat combiner, parametrized by specifications of sequential procedures, requiring exclusive access to the critical resource (§4.2.5).
- `stack_combine.v` — instantiation of the flat combiner with *sequential* stack push/pop procedures (§4.2.6).

2.1.3 Concurrent graph manipulations

Main example from the work [9], demonstrating handling of deep sharing in FCSL, described in Section 4.3.

Folder: `Examples/Graphs`

2.1.4 Non-linearizable concurrent objects and their clients

Examples from the paper on handling non-linearizable concurrent data structures in FCSL [11], described in detail in Section 4.4.

Folder: `Examples/NonLinearizable`

- `exchange.v` — an implementation of an elimination-based exchanger, specified via subjective histories, storing "twin" contributions.
- `exchange_client.v` — A client of the elimination-based exchanger, implementing the fair exchange of two sequences between two threads running in parallel in the absence of additional external interference.
- `qcounter.v` — an implementation and verification of a simple counting network, built via one binary balancer b and two atomic counters, c0 and c1, such that each of the counters stores only values of the corresponding parity. The given specification incorporates information about interference, observed in the past, and, hence, enables quantitative reasoning about the counter's clients.
- `qcounter_client1.v` — a client of the counting network, exercising the quiescent consistency and proving that in the presence of a quiescent state between two calls to the (quiescently-consistent) counter, the obtained results will come out in the natural order.
- `qcounter_client2.v` — A client of the counting network, used to demonstrate quantitative reasoning out of the provided spec, derived in the qcounter.v file. The specification ensures that the results of the two subsequent calls in the presence of interference come out of order by no more that 2 times the "size" of the interference.

2.1.5 Concurrent Data Structures Linked in Time

Folder: `Examples/Relink`

This folder contains the main case study of the *Linking in Time* technique for verifying concurrent data structures with non-fixed linearization points introduced in [1]. It consists of several files implementing the verification in FCSL of Jayanti's single writer/single scanner snapshot construction [3].

- `jayanti_snapshot.v` — Preliminaries. Lemmas for reasoning about coloring patterns of histories, and other history invariants.
- `jayanti_ghoststate.v` — Implementation of the auxiliary state for the snapshot object, its transitions, invariants and associated invariant preservation proofs.
- `jayanti_concurroid.v` — Implementation of the FCSL concurrent resource —a.k.a. *concurroid* for the snapshot.
- `jayanti_actions.v` — FCSL atomic actions for the snapshot object concurrent resource.
- `jayanti_stability.v` — Assertions used in the proof outline of the main methods and proofs of their stability under FCSL transitions.
- `jayanti_library.v` — Implementation and verification of the atomic commands (auxiliary code) and the snapshot library (i.e. scan and write procedures).
- `jayanti_clients.v` — Verification of the clients of the snapshot data structure, as presented in Section 4 of the paper [1].

2.2 FCSL metatheory

For the formalization of the metatheory, we have borrowed a number of libraries (`pred.v`, `prelude.v`, `ordtype.v`, `finmap.v`, `pperm.v`, `domain.v`, `array.v`) from the development of the verification framework for the sequential Hoare Type Theory [6] (which can be found in the folder `Heaps` of the project), so we don't describe them here.

Below, we list the libraries (contained in the folder `Core`), essential for the formulation of FCSL's semantics and the proof of soundness of its rules. The intuition behind the implementation can be acquired from the referred parts of the preceding papers on FCSL.

- `feaps.v` — PCM of heap maps.
- `state.v` — resource states implemented as triples of maps.
- `zigzag.v` — definition and helper lemmas about $\triangleleft$ (`<\`) and $\triangleright$ (`/>`) operators used in the fork-join closure [5].
- `worlds.v` — definition of concurroids, with properties on states and transitions.
- `auto_split.v` — some automation using overloaded lemmas.
- `inject.v` — meta theory required for proving the injection rule [5].
- `atomic.v` — definition and properties of atomic actions [5, Appendix A].
- `rely.v` — definition of the *Rely* predicate for a concurroid [5, Appendix B].
- `privstate.v` — concurroid for thread-local state [5, §4, Example 1].
- `footprint.v` — helper lemmas about disjointness of labels.
- `retract.v` — meta-theory required for proving the hiding rule [5, Appendix F.2].
- `schedule.v` — datatype of execution paths.
- `process.v` — definition of action trees [5, Appendices F.1, F.3 and F.4].
- `always.v` — definition of the modal predicates for adherence to a protocol [5, Appendices F.5 and F.7].
- `model.v` — implementation of denotational semantics of programs as sets of trees [5, Appendix F.6].
- `lemmas.v` — soundness of the rules; essentially each lemma is a case in the proof of the main soundness theorem from [5, Appendix F.7]. The lemmas also immediately serve as proof rules.
- `histories.v` — libraries for history PCMs the used in the specifications of optimistic fine-grained algorithms in [1, 10, 11]. The *standard* PCM history library is described in [10], whereas the *relinkable* history library is presented in [1].
- `stsep.v` — notation for traditional-style unary Hoare pre/postconditions.
- `getter.v` — generic getters for concurroid components.
- `seq_extras.v` — Algebraic surgery lemmas for lists, extending the original Ssreflect `seq` library.

3. Compilation

3.1 Building the project

In order to compile the whole project from scratch, which will ***automatically check the proofs of all theorems and specifications***, execute the following command from the command line in the project folder.

```
make clean; make
```

4. Commentary on the Case Studies

For further evaluation of the artifact, we invite the reader to take a tour of FCSL and explore particular example programs that we have verified. The most convenient way to do it is to use GNU Emacs (or Aquamacs in OS X) with the installed Proof General mode[1] for interactive proof construction in Coq. Opening any `.v`-file in the editor (use `Ctrl-X Ctrl-F` to navigate with Emacs) will automatically trigger the Proof General mode, making it possible to "step" through the file and examine intermediate steps of the proofs. Note, that in order to step through a particular file in Proof General, you will need to have all its dependencies (typically listed as `Require Import` arguments at the head of a file) compiled, which could be done in the case of FCSL via the supplied `make`-script (see §3.1).

Stepping through a Coq file in Emacs/Proof General can be performed via the `Ctrl-C Ctrl-Enter` shortcut, which makes the interactive proof engine to process the file till the current position of the caret. Use `Ctrl-C Ctrl-C` to interrupt the interpretation process (when some part of the buffer remains colored light-red). A typical Emacs/Proof General layout[2] for interactive proof construction is shown in Figure 1. Use `Ctrl-C Ctrl-L` to restore the initial layout.

[1] https://github.com/emacsattic/proofgeneral

[2] The standard layout is the "smart" one, but different configurations of the buffers can be chosen through the [Coq] → [3 Windows mode layout] menu.

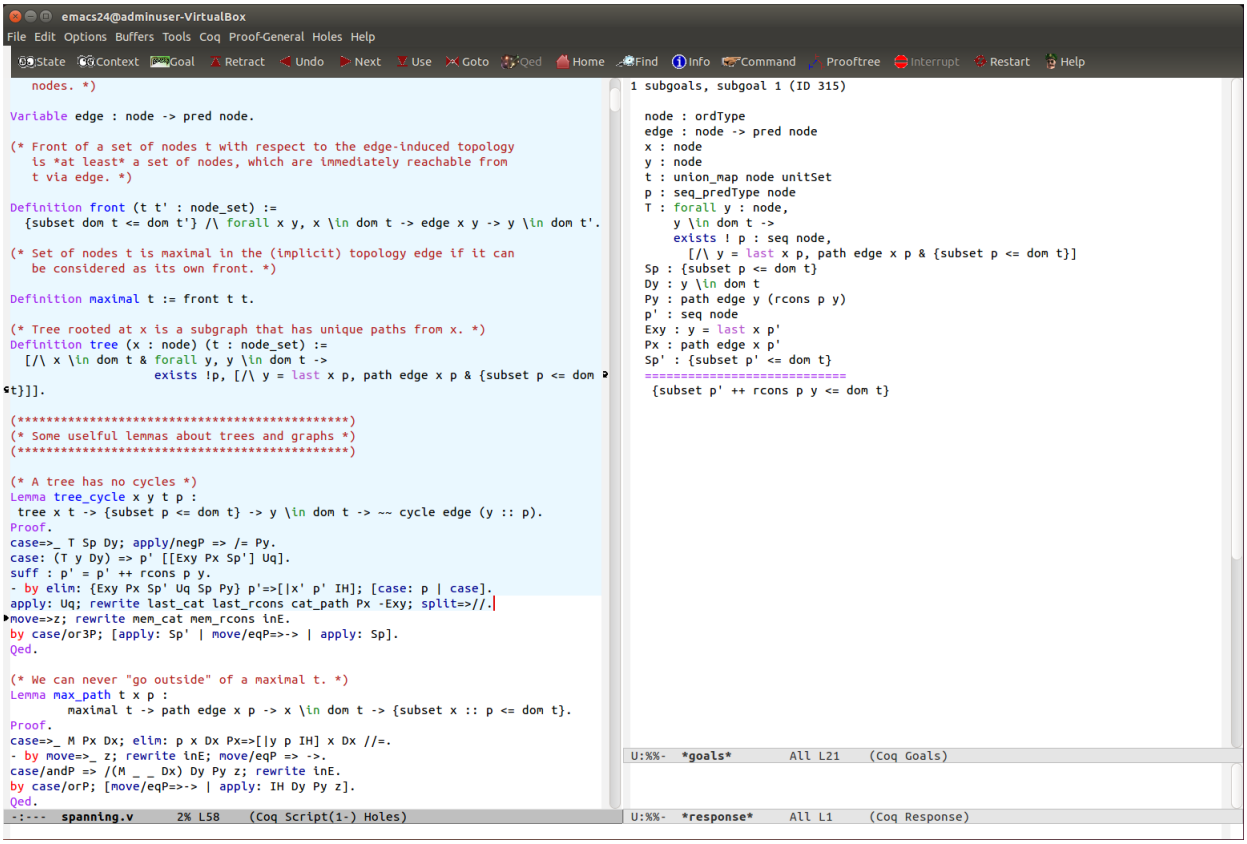

Figure 1: The hybrid layout of the Proof General mode in GNU Emacs 24 (in Ubuntu Linux 14.04) processing the `spanning.v` file. The left buffer indicates the current file, in which the blue part has been already processed by the Coq interactive interpreter. The top-right buffer displays the current proof context (above the `=======` line) and the goals remaining to be proved. The bottom-right buffer shows the responses of the interpreter.

4.1 Basic examples

Folder: `Examples/Basic`

4.1.1 Locking structures (`locks.v`)

Locks are the simplest fine-grained concurrent data structures, and we have implemented two instances of the locking mechanism: a CAS-based spinlock and a ticketed lock. The file `locks.v` defines a uniform basic structure for locks by packaging a number of values and facts about them in a dependent record as follows:

```
Structure lock_info := LockInfo {
  label_of: nat;
  pcm_of : encoded_pcm;
  inv_of : pcm_of -> Pred heap;
  precise_of : forall g, precise (inv_of g)
}.
```

The structure's fields are the label `label_of` of a concurroid implementing the lock; the client-specific PCM `pcm_of`, specifying the nature of the contributions that separate threads perform over the lock-protected part of the heap; the *lock invariant* `inv_of` that relates the cumulative contribution of all the threads and the heap, which is protected by the lock, and the proof `precise_of` that the invariant is *precise*, which is crucial for the consistency of the locking protocol in the classical Concurrent Separation Logic [7].

Abstract specification of a lock. What is a suitable abstract specification for a locking structure? To answer this question, we adopt the idea of specifying concurrent data structures via *abstract predicates* [2] and provide a lock interface in the

form of two abstract procedures: `lock` and `unlock`. Every data structure, implementing the lock protocol, will require to provide the implementation of these procedures. The lock interface is parametrized by two abstract predicates, constraining some part of the state: `is_lock` and `locked`. The former denotes the fact that the part of state implements belongs to the locking protocol, while the latter one identifies that the lock is currently being locked by the current thread.

Both predicates should satisfy a number of axioms, which are packaged into a dependent record in the section `LockStruct` (where the predicates are referred to as `locked_op` and `is_lock_op`). For instance, the following axiom expresses the lock's mutual exclusion property:

```
forall s gl ge, locked_op gl ge s -> locked_op ge gl (transp s)-> False
```

Specifically, it states that the situation when lock is held by the current thread (`locked_op gl ge s`) *and* it is simultaneously held by some other thread, which is expressed via a *transposition* of a concurroid's state `s` (`locked_op ge gl (transp s)`), implies the falsehood. The other property implies that if the state represents a locked lock, it also corresponds to a resource, which belongs to the lock protocol:

```
forall s gl ge, locked_op gl ge s -> is_lock_op gl ge s
```

These two predicates serve as parameters to specifications `lock_spec` and `unlock_spec`, which are supposed to be fulfilled by the locking and unlocking procedures, correspondingly. Let us consider the specification for locking, defined as follows:

```
Definition lock_spec is_lock locked : spec unit :=
  (* precondition predicate *)
  (fun i => exists (hl he : heap) (gl ge: U) i',
    i = hp ->> [hl, tt, he] \+ i' /\ is_lock gl ge i',
   (* postcondition predicate *)
   fun i y m => forall hl he (gl ge: U) i',
    i = hp ->> [hl, tt, he] \+ i' -> is_lock gl ge i' ->
    exists h ge' he' m',
      [/\ m = hp ->> [hl \+ h, tt, he'] \+ m',
          locked gl ge' m',
          valid (gl \+ ge') & I (gl \+ ge') h]).
```

The specification `lock_spec` takes predicates `is_lock` and `locked` as parameters. It states that the return type of the locking procedure is `unit`. While reminiscent to the specifications, shown in §4.3 for the spanning tree procedures, this spec bears two significant differences that are, in fact, typical rudiments from the earlier stages of FCSL development.

- ***Explicit concurroid structure***: The concurroid states in the pre- and postconditions are specified explicitly. For instance, the thread-local state concurroid is denoted as `hp ->> [hl, tt, he]`, where `hp` is its label, `hp` and `he` are heaps, that belong to this thread and its environment, correspondingly, and `tt` is its joint component, which is of `unit` type. With getters, which were developed later, we would be able to write the same part of the assertion as `pv_self i = hl /\ pv_other i = he` (actually, we could avoid mentioning `he` at all, since it is irrelevant in this case).
- ***Binary postconditions***: Unlike the postconditions in the specs we've seen before, the postcondition of `lock_spec` is *binary*, as it relates the pre-state `i` with a result `y` and the post-state `m` (the "unary" postconditions typically constrain just the result and the post-state). Binary postconditions are useful to capture the essence of logical (or, *ghost*) variables, that serve for the purpose of relating the pre- and postconditions [6]. For instance, the binary postcondition in `lock_spec` states that the thread-local heap, which initially was `hl`, at the end of the locking procedure became `hl \+ h`, *i.e.*, the heap `h` was acquired by the thread. With the notation for logical variables, used in the spanning tree example, we could have expressed the same relation by making `hl` to be a logical variable stated in the preamble of the specification as `{hl ...}`.

We can now read the specification `lock_spec`. In plain words, it postulates that the pre-state, observed by the thread can be split into two concurroids: the one for thread-local heaps (labelled `hp`), and another abstract concurroid for a lock (`i'`), such that `is_lock gl ge i'` holds and `gl`, `ge` are the contributions of this and other threads to the lock-protected resource, correspondingly. At the end of the locking procedure, the thread-local heap in the post-state `m` has been augmented by the heap `h`, released by the lock. Moreover, the lock is now in the locked state, which is captured by the predicate `locked gl ge' m'`, constrining the state `m'`, which belongs to the lock Finally, the acquired heap satisfies the lock-specific invariant `I (gl + ge')`, which we can now use for local reasoning.

We next instantiate the general locking data structure for two specific locking protocols and instantiate abstract predicates `is_lock` and `locked` for them.

4.1.2 CAS-lock (caslock.v)

The intuition behind the CAS-based lock protocol is described in detail in the original FCSL paper [5], which uses it as a running example.

The implementation of the lock first refines the general lock structure `lock_info` from `locks.v` by adding one more parameter: an address of the "flag pointer", which holds the boolean value indicating the fact of locking:

```
Structure lockT := LockT {
  lock_info_of :> lock_info;
  lock_of : ptr
}.
```

The coherence predicate describes the valid states of the lock, which are either "locked" (the protected heap is gone, the locking flag is `true`) or "unlocked", in which case the lock concurroid has the protected heap `h` in each joint state, and the heap satisfies the client-provided precise invariant.

```
Definition validRes s :=
  exists (mL mE : lift mutexUnlifted) (gL gE : U) (b : bool) h,
    s = l ->> [(mL, gL), lk :-> b \+ h, (mE, gE)] /\ P (mL \+ mE) (gL \+ gE) b h.

Definition cohRes P := [Pred s : state | [/\ validH s, validSO s & validRes P s]].

Definition locked m (g : U) b (h : heap) := [/\ m = up own, b = true & h = Unit].
Definition unlocked m g b h := [/\ m = up nown, b = false & I g h].
Definition either m g b h := locked m g b h \/ unlocked m g b h.

Definition coh := cohRes either.
```

The definition of the CAS-lock coherence predicate (section `LockCoh`) includes the proofs of the standard concurroid axioms, already mentioned in §4.3.

In addition to trivial (*i.e.*, identity-like) internal transitions, the lock concurroid features external transitions implementing communication via heap ownership transfer. In particular, the *acquire* transition (described in detail in [5, §4]) corresponds to the lock concurroid getting a heap satisfying the invariant `I (gL' + gE)` and placing it to its own joint component, as defined by the following relation between the initial and the final states (`s` and `s'`, correspondingly):[3]

```
Definition acq_rel s s' :=
  exists gL gL' gE : U,
    [/\ s = l ->> [(up own, gL), lk :-> true, (up nown, gE)],
        s' = l ->> [(up nown, gL'), lk :-> false \+ h, (up nown, gE)],
        valid (gL' \+ gE), valid (lk :-> false \+ h) &
        I (gL' \+ gE) h] /\ s \In coh.
```

Simultaneously, the value of the flag pointer `lk` is being changed from `true` to `false` and the value of the *mutex auxiliary resource* changes from `(up own)` to `(up nown)` in the self-part, indicating that the lock is no longer held by this thread.

The *release* transition is symmetric and corresponds to the lock giving up ownership on the heap `h` from its joint part, simultaneously with changing the value of the flag pointer, but preserving the value of the client specific auxiliary value `gL`:

```
Definition rel_rel s s' :=
  exists gL gE : U,
    [/\ s = l ->> [(up nown, gL), lk :-> false \+ h, (up nown, gE)],
        s' = l ->> [(up own, gL), lk :-> true, (up nown, gE)] &
        s \In coh].
```

The two defined transitions form a *communication channel*, such the heap may be transferred through it. Together with the coherence predicate, the transitions are next packaged in the definition of the concurroid `CLock` at the end of the `Exports` section of the `LockGuar` module.

[3] At the moment of implementing the locking protocols, the machinery for state getters (see the file `getter.v`) hasn't yet been implemented, which is why the definitions of concurroid states and transitions look a bit different from those of the spanning tree example. For the same reason, the proofs of the lemmas about locks are significantly larger, as they haven't been optimized using getters.

Next, we define so-called *pre-actions*, which implement the machinery of the corresponding acquire/release transitions of the lock concurroid, connecting it to the operational semantics of machine commands, such as `CAS` and `write`. This is done in the sections `LockReleaseAct` and `LockAcquireAct`. Later, the sections `CASAct` and `UnlockAct` turn this pre-actions into full-fledged actions that span a *composite* concurroid `(entangle (Priv hp) (CLock lkT))`, obtained by connecting acquire/release transitions of its constituents. Following the transitions with different "polarity", such actions perform a transfer of the heap from the lock's joint state to the thread-local state (described by the concurroid `Priv hp`) and vice versa: see the definitions of `cas_step` and `unlock_step` relations.[4]

Similarly to the development of the spanning tree concurroid in §4.3 we next give stable specifications to the atomic actions, operating with the lock state. For instance, the `CAS`-like operation, which, when successful, performs the heap ownership transfer from the lock-protected state to the thread-local state is given the following spec (with explicit concurroid structure and binary posts) in the form of a structural lemma:

```
Lemma val_cas1 (hL hE : heap) (mL mE : lift mutexUnlifted) (gL gE : U) (b : bool) h (r : cont bool) :
       (* if we succeed *)
       (forall hE' gE' h',
          valid (gL \+ gE') ->
          I (gL \+ gE') h' ->
          hp ->> [hL \+ h', tt, hE'] \+
           l ->> [(up own, gL), lk :-> true, (up nown, gE')] \In Mod.coh V ->
          r true (hp ->> [hL \+ h', tt, hE'] \+
                  l ->> [(up own, gL), lk :-> true, (up nown, gE')])) ->
       (* if we fail *)
       (forall hE' mE' gE' (b' : bool) h',
          hp ->> [hL, tt, hE'] \+
           l ->> [(mL, gL), lk :-> b' \+ h', (mE', gE')] \In Mod.coh V ->
          r false (hp ->> [hL, tt, hE'] \+
                   l ->> [(mL, gL), lk :-> b' \+ h', (mE', gE')])) ->
       verify (hp ->> [hL, tt, hE] \+
               l ->> [(mL, gL), lk :-> b \+ h, (mE, gE)])
              (act (CAS hp lkT)) r.
```

The first of the two "continuations" postulates that in the case of the success (the result is `true`) the heap `h'` satisfying `I (gL + gE')` has been transferred to the thread-local state. The second continuation states that in the case of failure the heap hasn't been transferred, and we don't know anything specific about it (since it could have been acquired by another thread by that moment). This weakening is required to make such specification stable.

We then define the actual meaning of the `locked` and `is_lock` predicates for the CAS-lock and prove that such definitions satisfy the axioms, imposed by the abstract lock structure. The `c_is_lock` predicate simply describes that the part of the state, constrained by it, satisfied the coherence predicate of CAS-lock:

```
Definition c_is_lock (gl ge : U) s :=
  exists (ml me: lift mutexUnlifted) (b: bool) (h: heap),
    s = l ->> [(ml, gl), flag :-> b \+ h, (me, ge)] /\
    s \In Mod.coh (CLock lkT).
```

The `c_locked` predicate makes sure that this thread currently holds the lock (*i.e.*, its self-component has `(up own)` element of the mutual exclusion PCM):

```
Definition c_locked (gl ge : U) s :=
  s = l ->> [(up own, gl), flag :-> true, (up nown, ge)] /\
  s \In Mod.coh (CLock lkT).
```

Finally, with these two predicates, we can give specifications to the locking and unlocking procedures of the CAS-lock. Both procedures operate on the composite concurroid, obtained as an entanglement of the thread-local state and the concurroid implementing the locking protocol.

```
Notation W := (entangle (Priv hp) (CLock lkT)).
```

[4] With the introduction of state getters, this pattern of defining actions operating with the composite concurroid (e.g., `(entangle (Priv hp) (CLock lkT))`), has been rendered obsolete, and such actions are defined directly in the later case studies, such as Treiber stack (§4.2.2).

The locking definition is a simple CAS-loop over the locking flag pointer (which is set using the lock-specific action `CAS`, defined and specified earlier in the file):

```
Definition cas_lock_spec : spec unit := @lock_spec hp lkT c_is_lock c_locked.

Program Definition cas_lock_proc : STbin W (cas_lock_spec) :=
 Do (ffix (fun (loop : unit -> STbin W cas_lock_spec) (_ : unit) =>
      Do (b <-- act (CAS hp lkT);
          if b then ret tt else loop tt)) tt).
```

The unlocking procedure is a single invocation of the `unlock` action, which is operationally equivalent to writing into the locking flag. However, in order to invoke it, the current thread should hold the lock, which is ensured by the precondition:

```
Definition cas_unlock_spec : spec unit := @unlock_spec hp lkT f c_is_lock c_locked.

Program Definition cas_unlock : STbin W (cas_unlock_spec) := Do (act (unlock hp pf)).
```

4.1.3 Ticketed lock (`ticketed.v`)

The ticketed lock (described in detail in [5, Appendix E]) implements a different protocol with better fairness guarantees, in which the threads, competing for the lock, first "draw" tickets, allowing them to request the access to the resource in the "first-come, first-served" order.

The ticketed lock augments the generic lock structure `lock_info` by adding two more fields to it:

```
Structure tlockT := TLockT {
  owner_of : ptr;
  next_of : ptr;
  lock_info_of :> lock_info
}.
```

The field `owner_of` stands for the pointer, whose current target value is the "counter", displaying the ticket, the owner of which is currently being served, or may come and get the lock-protected resource. The field `next_of` defines a pointer, whose target value is the next vacant ticket to be "drawn" by some thread. This leads to three generic states in which the ticketed lock can be: in addition to "locked" and "unlocked" state, there adds a "transit" state, meaning that there is a thread, which is just about to be served, as it has the right ticket, although it hasn't acquired the resource yet, hence the lock is not yet taken. This results in the following definition of the ticketed lock's coherence:

```
Definition validDispenser P s :=
  exists (n1 n2 : nat) (fL fE : nat_set) (gL gE : U) (b : bool) (h : heap),
    [/\ s = dl ->> [(fL, gL), owner :-> (n1, b) \+ next :-> n2 \+ h, (fE, gE)],
        n1 <= n2 & P n1 n2 (fL \+ fE) (gL \+ gE) b h].

Definition cohRes P :=
  [Pred s : state | [/\ validH s, validSO s & validDispenser P s]].

Definition locked n1 (f : nat_set) b (h : heap) :=
  [/\ n1 \in dom f, b = true & h = Unit].

Definition transit n1 n2 (f : nat_set) (g : U) b h :=
  [/\ n1 \in dom f, b = false, n1 < n2 & I g h].

Definition unlocked (n1 n2 : nat) (g : U) b h :=
  [/\ n1 = n2, b = false & I g h].

Definition either n1 n2 (f : nat_set) g b h :=
  dom f =i [pred x | n1 <= x < n2] /\
  [\/ locked n1 f b h | if n1 < n2 then transit n1 n2 f g b h else unlocked n1 n2 g b h].

Definition coh := cohRes either.
```

The boolean value `b` in the target value of the `owner` pointer in the joint part of the ticketed lock concurroid's state bears logical meaning, indicating, whether the lock is locked, in order to distinguish between the `locked` and `transit` states. The `transit` state differs from the `unlocked` state, as the latter assumes no threads waiting to be served (the ticket `n1` of the counter is the same as the ticket `n2` at the dispenser). The self/other parts of the concurroid, besides the client-specific contributions `gL` and `gE` contain the sets of currently held tickets, implemented as finite sets of type `nat_set`: `fL` and `fE`. Indeed, final sets form a PCM with disjoint union as an operation. The coherence predicate `either` ensures that all currently held by threads tickets are in the range `n1 <= x < n2`. This design solution is describe in more detail in [5, Appendix E.2].

The ticketed lock concurroid features three transitions. `int_rel` describes "drawing" of a ticket and corresponds operationally to a fetch-and-increment command. Similarly to the CAS-lock, the acquire transition `acq_rel` corresponds to *unlocking* (operationally, this is just a read command), and the release transition `rel_rel` defines the locking (it changes the value of the logical flag `b`, but operationally can be effectively considered as a no-op).

The layout of defining the actions of the ticketed lock protocol (some of them, inded, cover the entanglement of the concurroid for the thread-local state and the one of the ticketed lock, *e.g.*, the `TickTryLockAct` action) follows the same structure as for the CAS-lock (§4.1.2).

The instances of abstract lock predicates are defined for the ticketed lock as follows:

```
Definition t_is_lock (gl ge : U) s :=
  exists (fl fe: nat_set) (n1 n2: nat) (b: bool) (h: heap),
    s = l ->> [(fl, gl), owner :-> (n1, b) \+ next :-> n2 \+ h, (fe, ge)] /\ s \In Mod.coh (TLock lkT).

Definition t_locked (gl ge: U) s :=
  (exists (fl fe : nat_set) (n1 n2: nat),
    s = l->>[(fl \+ #n1, gl), owner:->(n1, true) \+ next:->n2, (fe, ge)]) /\ s \In Mod.coh (TLock lkT).
```

In particular, `t_locked` requires the logical flag `b` to be `true` and the current thread to own the ticket `n1`, which is currently being served. These two facts together make sure that this thread indeed holds the lock. The locking program first "draws" atomically a ticket and then spins in the loop until this ticket is displayed on the counter (checked by the action `ttryLock`).

```
Program Definition tlock : STbin W (tlock_spec) :=
 Do (n <-- extend _ (act (tdraw lkT)); (* draw a ticket *)
     ffix (fun (loop : unit -> STbin W (t_loop_spec n)) (_ : unit) =>
       Do (b <-- act (ttryLock hp lkT n); (* check the counter *)
           if b then ret tt else loop tt)) tt).
```

The derived specification, therefore, conforms to the abstract spec of locking from `locks.v`, instantiated with specific predicates `t_is_lock` and `t_locked`. The definition of the unlocking procedure is straightforward.

4.1.4 Coarse-grained incrementor (`incrementor.v`)

Incrementor is a toy coarse-grained concurrent program, described as a running example for introducing FCSL [5, §3]. It is a parametrized by a lock structure, which protects a heap, consisting of a single pointer `x`, pointing to a natural number. The concurrent thread can add arbitrary non-negative amounts to the value, given that it acquires the lock first, so the pointer's value will be modified in a race-free manner. These added amounts are "subjective contributions", made to the lock-protected resource by concurrent threads, hence, the incrementor-specific PCM is a set of natural number with addition as join operation and zero as a unit element.

The lock invariant for an incrementor is, thus, defined as follows, stating that the heap is a single entry of the pointer `x`, whose target value is the cumulative contribution `g` of the threads, operating with the resource.

```
Definition incr_inv (g : nat) := [Pred h | h = x :-> g].
```

We next prove that this variant is indeed *precise*, *i.e.*, it unambiguously picks out an area of heap [7]:

```
Lemma incr_inv_prec (g : [encoded_pcm of nat]) : precise (incr_inv g).
Proof. by move => h1 h2 t1 t2 V1 E; move=>->->. Qed.
```

The incrementor procedure is given specification using the familiar unary pre- and post-conditions with `gs` being a logical variable, binding the initial amount of self-contribution to the incrementor resource by this particular thread.

```
Program Definition incr n :
 {gs}, STsep [W]
        (fun i => exists sl go,
          i = hp ->> [Unit, tt, pv_other i] \+ sl /\ is_lock gs go sl,
         fun (_: unit) m => exists sl go,
          m = hp ->> [Unit, tt, pv_other m] \+ sl /\
          is_lock (gs + n) go sl) :=
  Do (lock;;
      tmp <-- [priv_read hp nat x];
      [priv_write hp x (tmp + n)];;
      unlock (@pfF n)).
```

The precondition, given in a *small footprint* states that the local heap is empty (*i.e.*, it is `Unit`), and the self-contribution is `gs` (which is captured by the abstract predicate `is_lock gs go sl`). At the end, the local heap is empty again, but the local contribution has been changed to `(gs + n)`, which is precisely the subjective effect of the incrementor.

The body of the incrementor procedure first acquires the lock via the `lock` procedure, transferring the pointer to be incremented to the thread-local heap. It then reads the current pointer's value via the action `priv_read`, defined over the thread-local state concurroid and "injected" into the entanglement via the `[...]` macro. Next, it writes the new value `tmp + n` into the pointer using the `priv_write` action. Finally, it unlocks the lock using the `unlock` procedure, which requires a provably *local* function as an argument. The intuition behind this requirement is defined in the initial paper on Subjective Concurrent Separation Logic [4]. Here, such function (`fun x => x + n`) is provided by the proof of the following lemma:

```
Lemma pfF n : @local [encoded_pcm of nat] (fun x => x + n).
Proof. by move=>m y; rewrite /= -addnAC. Qed.
```

4.1.5 Coarse-grained allocator (`alloc.v`)

A coarse-grained concurrent memory allocator, used by many algoritms and data structures, is another client of the abstract lock library.

The allocator is implemented as a lock-protected chunk of memory, representing a pool of available entries and mirrored as a logical list of pointers with a "sentinel" pointer, which points to the list of available entries. The allocation is, therefore, implemented as taking the first available entry in the pool and moving it logically to the thread-local state, simultaneously with changing the target value of the sentinel pointer, so it would point to the next available memory entry. In the case if the pool is empty, the allocator loops until some other thread frees memory. The `free` operation simply returns the deallocated memory cell and registers it as a new logical "head" of the memory pool.

The allocator's lock invariant is implemented via the two following definitions:

```
Fixpoint alloc_inv' avl :=
  [Pred h | if avl is x::xs
            then exists h', h = x :-> tt \+ h' /\ alloc_inv' xs h'
            else h = Unit].

Definition alloc_inv (g : unit) : Pred heap :=
  [Pred h | exists (avl: seq ptr) h',
             [/\ valid h, h = sent :-> avl \+ h' & alloc_inv' avl h']].
```

The recursive predicate `alloc_inv'` specifies the memory pool with respect to the logical list `avl`, while `alloc_inv` augments it with a sentinel pointer `sent`. Next, we prove that `alloc_inv` is precise, so we can instantiate a lock with it.

The key component of the allocation procedure is the function `try_alloc`, whose result is of `option` type, indicating the possibility of a failure. The function first acquires the lock and then checks whether the pool is empty. If it is not, it detaches the first available element `p`, unlocks and returns the result `Some p`. Otherwise, it unlocks and returns `None`.

```
Program Definition try_alloc :
  {hS : heap}, STsep [W]
  (fun i => exists st (h0: heap), i = hp ->> [hS, tt, h0] \+ st,
   fun v m => if v is Some p
              then exists st' (h0': heap) B (b: B), m = hp ->> [p :-> b \+ hS, tt, h0'] \+ st'
```

```
            else exists st' (h0': heap), m = hp ->> [hS, tt, h0'] \+ st') :=
  Do (lock;;
      avl <-- [priv_read hp _ sent];
      if avl is p::ps
      then [priv_write hp sent ps];;
           unlock pfF;;
           ret (Some p)
      else unlock pfF;;
           ret None).
```

The main allocation procedure `alloc_proc`, defined next, is a generally-recursive procedure, whose specification `alloc_tp` is also its "loop invariant".

```
Definition alloc_tp :=
  {hS : heap}, STsep [W]
    (fun i => exists st (h0 : heap), i = hp ->> [hS, tt, h0] \+ st,
     fun p m => exists st' (h0' : heap) B (b : B), m = hp ->> [p :-> b \+ hS, tt, h0'] \+ st').

Program Definition alloc_proc :=
  ffix (fun (loop : unit -> alloc_tp) xx =>
    Do (res <-- try_alloc;
        if res is Some p
        then ret p
        else loop tt)) tt.
```

The procedure spins in the loop until the allocation is successful. Its type indicates that in the post-state the initial thread-local heap `hS` is increased by adding an entry of a pointer `p :-> b`, which points to an unspecified value `b`.

The procedure `free_proc`, specified by the Hoare type `free_tp`, is similar, although, it never fails, and, hence, doesn't need to re-iterate. What it does is simply writes the unit value `tt` to pointer to be deallocated, acquires the lock, reads the current value of the sentinel and augments it by the pointer `p`, which is, therefore, "returned" to the allocator resource.

```
Definition free_tp (p : ptr) :=
  {hS : heap}, STsep [W]
               (fun i => exists st (h0: heap) B (b: B),
                   i = hp ->> [p :-> b \+ hS, tt, h0] \+ st,
                fun (_ : unit) m => exists st' (h0' : heap) B (b : B),
                   m = hp ->> [hS, tt, h0'] \+ st').

Program Definition free_proc p : free_tp p :=
    Do ([priv_write hp p tt] ;;
        lock;;
        avl <-- [priv_read hp _ sent];
        [priv_write hp sent (p::avl)];;
        unlock pfF).
```

4.2 Linearizable concurrent objects and universal constructions

Folder: `Examples/Histories`

4.2.1 Atomic Pair Snapshot (`readpair.v`)

An atomic pair snapshot algorithm is the primary example illustrating specification and verification method in FCSL, based on the notion of *time-stamped histories* (or just histories). The main intuition on histories and the snapshot algorithm is presented in [10, §2].

Intuitively, histories specify atomic changes in the essential part of the concurrent resource's state. Each such atomic change is assigned a timestamp (a natural number), so the histories can be viewed as partial maps from timesamps to atomic changes, which makes then an instance of a PCM, fitting naturally into the subjective model of FCSL's concurrent resources. We typically consider *complete* histories, in which there are no "gaps" between subsequent timestamps. We also require the histories to be *continuous*, meaning that each next atomic change starts from the state, at which the previous

action left the resource. The library with the implementation of histories and their main properties is provided by the file `histories.v`.

The atomic pair snapshot is represented as a pair of pointers, `x` and `y`, which point to the values of a client-specific type `encoded_set`. These parameters, together with a label `label_of` for the concurroid are packaged in the following structure:

```
Structure readpairT := ReadPairT {
  label_of : nat;
  eset_of : encoded_set;
  x_of : ptr;
  y_of : ptr }.
```

The coherence predicate for the atomic snapshot concurroid is defined as follows:

```
Definition coh :=
  [Pred s | exists (hs ho : hist st) (tvx tvy : nat * A),
    [/\ s = l ->> [hs, x :-> tvx \+ y :-> tvy, ho],
        validH s,
        [(tvx, tvy.2) in hs \+ ho at last_key (hs \+ ho)],
        stamps_mono (hs \+ ho) /\ stamps_inj (hs \+ ho) /\
        stamps_bound (hs \+ ho) tvx.1 &
        well_formed (hs \+ ho)]].
```

In particular, it states that the joint state consists of two pointer entries, `x :-> tvx` and `y :-> tvy`, and self/other parts are histories of atomic changes. It also requires the last entry in the history to be exactly the current pair of values in the snapshot: `[(tvx, tvy.2) in hs + ho at last_key (hs + ho)]`. Furthermore, the coherence predicate requires the snapshot histories to be monotone and injective with respect to allocated "version numbers" (`stamps_mono`) as well as well-formed in the general sense (*i.e.*, continuous and complete). All these properties are explained in [10, §2].

The atomic smapshot concurroid supports two internal transitions: writing atomically to `x` and `y`. Each transition, besides writing to the appropriate value and bumping up its version number, also augments the history with a new entry (which is ascribed to *this* thread by registering it in its *self*-component), indicating the new atomic change. For instance, the relation, implementing the atomic write to `x`, is defined as follows:

```
Definition writex_rel s s' :=
  exists (pf : s \In coh) (pf' : s' \In coh),
    let: (tx, vx) := ([getx pf].1, [getx pf].2) in
    let: (vx', vy) := ([getx pf'].2, [gety pf].2) in
    let: h := self s \+ other s in
    [/\ self s' = self s \+ fresh h \\-> ((tx, vx, vy), (tx.+1, vx', vy)),
        other s' = other s, [getx pf'].1 = tx.+1 & [gety pf'] = [gety pf]].
```

Here `[getx pf]` and `[gety pf]` are "getters" for the corresponding actual values of the pointers `x` and `y` in the concurroid's joint parts. One can see that the eventual self-history (`self s`) is augmented by the new entry `fresh h \\-> ((tx, vx, vy), (tx.+1, vx', vy))`, where `fresh h` is a new timestamp. While the new value of `y` remains the same (`[gety pf'] = [gety pf]`), the "version" of `x` is incremented by one: `[getx pf'].1 = tx.+1`. The transition, describing writing into `y`, is defined similarly.

We next define two actions for atomically reading the values of `x` and `y` (sections `ReadXAct` and `ReadYAct`, correspondingly). The actions are later given stable specifications under the names of standalone programs `readx_proc` and `ready_proc`. The procedure, which returns an atomic snapshot (*i.e.*, the pair of values, which indeed were present in the structure at some point of time, as reflected by the history), is specified via the type `read2_tp` and implemented as follows:

```
Definition read2_tp :=
  {h}, STsep [W] (fun i => self i = Unit /\ [h <<= other i],
                  fun vxy m => exists t tx,
                    [/\ last_key h <= t,
                        self m = Unit, [h <<= other m] &
                        [(tx, vxy.1, vxy.2) in other m at t]]).

Program Definition read_pair :=
  ffix (fun (loop : unit -> read2_tp) xx =>
```

```
Do (tvx <-- readx_proc;
    tvy <-- ready_proc;
    tmpx <-- readx_proc;
    if tmpx.1 == tvx.1 then ret (tvx.2, tvy.2)
    else loop tt)) tt.
```

The body of the function is a loop, which performs a conservative check at the end, making sure that the version (and, hence, the value) of `x` hasn't changed while the procedure was reading from `y`. If it did, the procedure re-iterates, otherwise it returns the pair of the values it has read.

The postcondition of the specification `read2_tp` postulates that the result `vxy` indeed corresponds to some entry in the other-history (`other m`), *i.e.*, it an entry, which was contributed to the total history by another thread. The fact that each entry corresponds to a snapshot's value at some point of time implies that the result is a *correct* atomic snapshot. Moreover, the timestamp `t`, corresponding to the entry read, happened to be the actual one *during* the procedure execution, which is denoted by the conjuncts `last_key h <= t` and `[h <<= other m]`. The specification is given in the *small footprint*, assuming that the current thread hasn't written anything to the structure itself (`i = Unit`, `m = Unit`), but as the paper [10] demonstrates, a more general specification can be obtained from this one by applying the FCSL's *framing rule*, which allows for this thread previously writing to the history itself. In this case, the result will correspond to the entry, contributed either by this or by some other thread.

4.2.2 Treiber stack (`treiber.v`)

Treiber stack is another major example, described in the paper [10, §4]. The intuition for the implementation of the Treiber concurroid is similar to the design of an atomic snapshot protocol (§4.2.1): "effectful" changes (*e.g.*, push and pop) are reflected in self/other history (depending on which thread has performed them), while the read-like actions simply reflect on the actual history and state.

Internally, Treiber stack is implemented as a single-linked list with a single entry point: a "sentinel" pointer `sent`, through which the concurrent threads modify the stack's contents, while performing the *push* and *pop* operations. The single-linked list structure is described via the following standard recursive predicate over heaps:

```
Fixpoint llist A p (xs : seq A) {struct xs} :=
  if xs is x::xt then
    [Pred h | exists r h', h = p :-> (x, r) \+ h' /\ h' \In llist r xt]
  else [Pred h | p = null /\ h = Unit].
```

We instantiate the generic getter structure (defined in `getter.v`), so we would be able to extract components of the concurroid, associated with the label `tb` and the auxiliaries of type `U` (which abbreviates the PCM of histories):

```
Definition tbg := @Getter tb U heap_tp_Encoded Unit.
```

The coherence predicate for the Treiber concurroid makes use of the familiar notation for self/joint/other state via the corresponding getters. It states that the joint part of the Treiber's state consists of the sentinel pointer entry `sent :-> p`, a heap `h`, implementing a single-linked list with the head pointer `p` and a *garbage* heap `g`, where the popped nodes reside.[5] The coherence predicate also states that the actual logical contents of the stack `ls` correspond to the latest change in the combined self/other history `H`, expressed as `[ls in H at last_key H]`. Finally, the coherence requires the history `H` to be complete ans *stack-continuous*, meaning that the new entries correspond to actual push and pop operations (more explanations on the design of the concurroid are provided in [10, §4]).

```
Definition coh :=
  [Pred s | let: H := self s \+ other s in
            exists p h g ls,
              [/\ s = tb ->> [self s, joint s, other s],
                  joint s = sent :-> p \+ h \+ g,
                  llist p ls h, [ls in H at last_key H],
                  stacks_cont H /\ complete H &
                  validH s]].
```

[5] Treber stack never deallocates, and this memory leak is by design of the data structure in order to avoid the ABA problem.

The Treiber concurroid has two transitions: for pushing and popping elements. The pop-transition is an *internal* one, as it doesn't release any heap to outside, but only changes the value of the sentinel pointer `sent`, so it points to the next pointer `p'` in the single-linked list, implementing the stack:

```
Definition pop_rel s s' :=
  let: H := self s \+ other s in
  exists (p p' : ptr) e (h : heap) ls,
    [/\ joint s = sent :-> p \+ p :-> (e, p') \+ h,
        joint s' = sent :-> p' \+ p :-> (e, p') \+ h,
        self s' = self s \+ fresh H \\-> (e :: ls, ls),
        other s' = other s,
        s \In coh & s' \In coh].
```

The push-transition is an acquire transition, as it requires a memory entry, corresponding to the new top-element of the stack (`p' :-> (e, p)`) to be transferred to the joint part of the concurroid from outside:

```
Definition push_rel s s' :=
  let: H := self s \+ other s in
  exists (p p' : ptr) e (h : heap) ls,
    [/\ h' = p' :-> (e, p),
        joint s = sent :-> p \+ h,
        joint s' = sent :-> p' \+ p' :-> (e, p) \+ h,
        self s' = self s \+ fresh H \\-> (ls, e::ls),
        other s' = other s &
        s \In coh /\ s' \In coh].
```

It also changes the target value of `sent`, so it would point to the new stack top.

We next implement four actions operating with the stack and making use of its transitions.

- The `trypop` action (section `TryPopAct` of the file) implements the CAS-like operation, trying to logically remove the top element of the stack. It is safe to run *only* if the "guessed" current top element of the stack is in the Treiber's joint state (as an "alive" stack element or as a garbage). This is expressed by the `trypop_safe` predicate. The action succeeds if the top pointer of the stack is still the one that was guessed (`p`), and fails otherwise, implementing, therefore the logic of the CAS operation.
- The `read_sent` action (section `ReadSentAct`) implements the logic of reading the current target value of the sentinel pointer. The result of this action is guaranteed to be some stack node, which was there at some point of time (but by now it might be already in the garbage). This action corresponds to `read` command operationally.
- The `read_node` action reads content of a stack node. Since the contents of stack nodes never change according to the concurroid transitions (the only changing value is the one of the sentinel pointer), the result of this operation produces stable knowledge.
- The `trypush` action (section `TryPushAct`) spans the entanglement of the concurroid `Priv` for thread-local state and the Treiber concurroid. Operationally equivalent to CAS, it changes the value of `sent` and transfers the new memory entry from `Priv`'s *self* to the Treiber concurroid in the case of success; otherwise, it does nothing.

Section `Stability` establishes several important facts about stability of the results of some actions (in particular, `read_node`) with respect to the concurroid interference.

The body of the Treiber's push makes use of the allocator (§4.1.5), and, hence, spans *three* primitive concurroids: the one of the lock, protecting the allocator, the Treiber concurroid and the concurroid for thread-local state. The body of push first allocates a new memory entry and then initiates the loop, in which it first reads the current top pointer `p'` of the stack, writes it as the *next* element to the newly-allocated node (`[priv_write pv p (e, p')]`), and then tries to push the new node to the stack. It re-iterates until `[try_push p p' e]` returns `true`.

```
Program Definition push e :
  {h hS}, STsep [V] (fun i => [/\ tb_self i = Unit, pv_self i = hS & [h <<= tb_other i]],
                     fun (_ : unit) m => exists t ls,
                       [/\ tb_self m = t \\-> (ls, e :: ls), pv_self m = hS,
                           [h <<= tb_other m] & last_key h < t]) :=
  Do (p <-- [alloc pv al asent alk];
```

```
ffix (fun (loop : unit -> push_loop_tp e p) xx =>
  Do (p' <-- [read_sentinel T];
      [priv_write pv p (e, p')];;
      ok <-- [try_push p p' e];
      if ok then ret tt else loop tt)) tt).
```

The side-effect of push is captured in its postcondition, which states that the self-history (`tb_self`) has changed from `Unit` to the singleton entry `t \\-> (ls, e :: ls)`, which corresponds to the push-entry in the history.

The `pop` procedure is implemented similarly, as a loop, although its specification is more subtle, since it accounts for the possibility of encountering an empty stack during the execution, and, hence, in the other-history (`[[::] in other m at t]`), in which case the procedure's result `v` is `None`. In the case of success, the contribution of `pop` is described by the pop-entry `t \\-> (e :: ls, ls)`, which is added to the self-history.

```
Definition pop_tp :=
  {h}, STsep [W] (fun i => self i = Unit /\ [h <<= other i],
                  fun v m =>
                    [/\ [h <<= other m] &
                        if v is Some e then
                          exists t ls, last_key h < t /\
                                       self m = t \\-> (e :: ls, ls)
                        else exists t, [/\ last_key h <= t, self m = Unit &
                                           [[::] in other m at t]]]).
```

Similarly to the atomic pair snapshot (§4.2.1), specifications of `push` and `pop` are given in the small footprint, assuming that the current thread so far has made *no contribution* to the history (`tb_self i = Unit`). We will show how to relax this restriction in the next section.

Framing stack specifications (`stack_framed.v`) The specifications for `push` and `pop` derived so far don't account for the possibility of the thread being specified to contribute to the stack history *before* executing these commands. This requirement can be lifted by applying FCSL's *frame rule* to the specifications we have by now [10]. The file `stack_framed.v` demonstrates how to do it.

As an example, let us consider the "framed" specification for `push`, namely, `push_framed_tp`:

```
Definition push_framed_tp e :=
  {h hs hS}, STsep [V] (fun i => [/\ tb_self i = hs, pv_self i = hS & [h <<= hs \+ tb_other i]],
                   fun (_ : unit) m => exists t ls,
                     [/\ tb_self m = hs \+ t \\-> (ls, e :: ls), pv_self m = hS,
                         [h <<= hs \+ tb_other m] &
                         last_key h < t]).
```

We can see that now the self-history of the Treiber stack is allowed to be *any arbitrary history* (`tb_self i = hs`). However, this change also required us to modify the other conjuncts in the pre- and postcondition in order to keep the whole specification consistent and provable. In particular, the ultimate self-history at the end of `push` now contains `hs`, so the final self-history looks as `tb_self m = hs + t \\-> (ls, e :: ls)`. Moreover, the two conjuncts `[h <<= hs + tb_other m]` and `last_key h < t` imply that the fresh timestamp `t` allocated for the push-entry is positioned strictly "later" than the initial history `hs`.

The key steps of the corresponding framing proofs are the application of the `frame_do` and `frame'` lemmas, defined in `getter.v` file. Unsurprisingly, the body of the framed operation `push_framed` is just a call to `push`, wrapped into FCSL's `Do` constructor, allowing one to weaken the specification for the program at the right-hand-side of the definition:

```
Program Definition push_framed e : push_framed_tp e := Do (push pv al asent alk e).
```

A similar derivation is done for the `pop` procedure.

4.2.3 Sequential stack (`stack_seq.v`)

Section 4 of the paper [10] shows how to obtain a *sequential* specification of a stack from the framed concurrent specifications, derived in §4.2.2. We omit the detailed intuition behind the proof, as it is described in that paper and focus on the implementation instead.

A procedure has a sequential specification if it operates solelyon the thread-local states (possibly, also using an allocator), without touching other concurroids. However, given enough thread-local states, one can *allocate* lexically-scoped concurroids in it using FCSL's hiding mechanism, which was already employed for verifying the spanning tree algorithm (§4.3). The file `stack_seq.v` demonstrates how to make use of hiding to implement a sequential stack from Treiber stack.

Sequential push and pop operations are obtained by "wrapping" calls to Treiber's push and pop operation into a hiding constructor via the `TreiberDecorate` structure, defined at the end of `treiber.v` file. Let us consider the implementation of the sequential push as an example.

```
Notation V := (entangle (Priv pv) AllocM).
Definition push_seq_tp (e: A) :=
  {ls}, STsep [V] (fun i => exists (p: ptr) hs,
                    [/\ pv_self i = sent :-> p \+ hs,
                        l \notin sdom i &
                        llist p ls hs],
                   fun (_ : unit) m => exists (p: ptr) hs g,
                     [/\ pv_self m = sent :-> p \+ hs \+ g &
                         llist p (e::ls) hs]).

Program Definition push_seq e : push_seq_tp e :=
  Do (priv_hide (TreiberDecorate T) (Unit, Unit)
                (cast (T:=fun m => ST m _) (entangleAC _ _ _)
                      (push_framed pv al asent alk e))).
```

From the specification type `push_seq_tp` we see that the procedure `push_seq` affects only the concurroid `Priv pv` for the thread-local state and a generic allocator concurroid `AllocM` (it requires the latter one to allocate new nodes for the stack). In order to run the concurred operation `push_framed` within the hiding section (`priv_hide`), the thread needs to allocate this concurroid and assign it a fresh label `l`, which is ensured by `l \notin sdom i` in the precondition. Moreover, the precondition guarantees that the thread-local state is sufficient to be donated for creating the joint part of Treiber concurroid (`pv_self i = sent :-> p + hs`). The argument `(Unit, Unit)` of hiding, instantiates the initial self-history and the initial garbage with empty values. The wrapper `(cast (T:=fun m => ST m _) ...)` is necessary to statically *cast* the concurroid, in which `push_framed` "lives", to the one, resulting from hiding. The cast is justified by rearranging its primitive concurroids via associativity and commutativity of entanglement [5], hence the `(entangleAC ...)` argument.

The last conjunct `(llist p ls hs)` of the precondition ensures that the heap `hs` implements a list (*i.e.*, a sequential stack). In the postcondition, the last conjunct `llist p (e::ls) hs` guarantees that the resulting heap implements the list with the new element `e` on the top. However, it also accounts for some possible garbage `g` that could be introduced by the Treiber machinery (we could prove a stronger specification, ruling out the garbage in this case, but this would require to implement a significantly more complicated definition of the decorator predicate).

The main burden of the proof stems from the necessity to reflect on the fact that the singleton history with just one push-entry combined with the absence of interference (since we didn't fork new threads within the hiding) are sufficient to infer the exact shape of the resulting stack. The detailed explanation of the proof can be found in §4 of the paper [10].

4.2.4 Producer/Consumer (`stack_client.v`)

An interesting client program, which makes use of the Treiber stack and its framed specifications (see §4.2.2), is a concurrent producer/consumer protocol. The program runs two threads in parallel, such that they exchange elements through a shared stack with one thread only pushing (the producer) and another only popping (the consumer). The full functional correctness of this program will require us to prove that by the end of the execution all the elements pushed by the producer thread were successfully popped and accommodated by the consumer in tis locally-owned array.

The verification of this example is conducted purely out of history-based specifications of the Treiber stack (§4.2.2), this is why the file `stack_client.v`, implementing the producer/consumer protocol, starts from defining a number of predicates on stack-specific histories as well as their properties.

In particular, the recursive predicate `pushed` extracts the sequence *all* elements `xs` that have ever been pushed into the history `h` (including those that have been there initially):

```
Definition only_popped (h : hist) :=
  forall t s1 s2, find t h = Some (s1, s2) ->
```

```
  [/\ t = 0, s1 = s2 & s2 = [::]] \/
  exists v, s1 = v :: s2.

Fixpoint pushed (h : hist) (xs : seq A) : Prop :=
  if xs is x :: xs' then
    exists t s h', [/\ h = t \\-> (s, x::s) \+ h' &
                       pushed h' xs']
  else only_popped h.
```

More precisely, for a non-empty sequence of elements `xs = x :: xs'` of elements, the `pushed` predicate finds a corresponding push-entry in the history `h`. If `xs` is empty, the predicate makes sure that the remaining history contains only pop-entries. The definition of the predicate `popped` is similar. Together, both these predicates are described more explicitly as equations (23) in the paper [10].

The file then formulates several lemmas about `pushed` and `popped` predicates, which establish their properties under union of disjoint histories (`h1 \+ h2`), permutation of elements in `xs`, and addition of new entries into the history. In particular, the lemmas `pushed_mono` and `popped_mono` imply **Lemma 1** from [10].

The following definition specifies the elements that are currently in the stack (*i.e.*, the elements of the stack, corresponding to the latest history entry):

```
Definition in_stack (h: hist) (xs: seq A) : Prop :=
  h = Unit /\ xs = [::] \/
  [xs in h at last_key h].
```

The main lemma `push_pop_all` relates the pushed and popped entries of the history `h`, which is `complete` and `stack_continuous` (the paper [10] calls this property stacklike) and the actual elements present in the stack, as inferred from the history `h`.

```
Lemma push_pop_all h (L1 L2 : seq A) :
       complete h -> stacks_cont h -> pushed h L1 -> popped h L2 ->
       exists (L: seq A), in_stack h L /\ perm L1 (L2 ++ L).
```

We can see now that if `L = [::]` (*i.e.*, the eventual stack is empty), one can infer that `L1` and `L2` are permutations of each other, which makes **Lemma 2** from [10] to follow straightforwardly.

We proceed to the definitions of the producer/consumer procedures and their specifications. For instance, the consumer is defined as follows (with its postcondition extracted into the separate definition `consumer_post`):

```
Let consumer_post Ac :=
  fun (_ : unit) m =>
      let: H:= tb_self m in
      exists f,
      [/\ is_array Ac f (pv_self m),
      pushed H [::] & popped H (fgraph f)].

Definition consumer_tp (Ac : {array [finType of 'I_n] -> A}) :=
  forall j,
  STsep [W] (fun i =>
               let: H:= tb_self i in
               exists f,
               [/\ is_array Ac f (pv_self i), j <= n,
                   pushed H [::] & popped H (take j (fgraph f))],
              fun (_ : unit) m => consumer_post Ac tt m).

Program Definition consumer Ac : consumer_tp Ac :=
  ffix (fun (loop : consumer_tp Ac) i =>
    Do (if i == n
        then ret tt
        else t <-- [pop_framed T];
             if t is Some v then
```

```
      [priv_write pv (Ac .+ i) v];;
      loop (i.+1)
    else loop i)).
```

The consumer procedure and its spec are parametrized by the consumer array structure `Ac`. We have chosen to implement arrays as heap fragments, whose contents are described by a *finite function*, mapping indices, ranging from `0` to the specified array length, to the actual elements of type `A`, as defined by the type macro `[finType of 'I_n]` (the length of the array is bound by the variable `n` in section `ClientPrograms`):

```
Definition is_array p := @shape [finType of 'I_n] A (Array p).
```

The definition of the record `Array` is implemented in the `array.v` file, inherited from the implementation of Hoare Type Theory [6]. For the given finite function `f`, representing the array contents, the expression `(take j (fgraph f))` constructs a list of results of the function on inputs from `0` to `j`. With this notation, the pre- and postconditions of `consumer` can be directly mapped to its mathematical specification (24) in the paper [10].

The obligations, arising from the specifications of the `produce` and `consume` procedures are proved using the familiar FCSL machinery. Next, the procedure `prodcons_parcomp` combines the call to produced and consumer using the parallel composition operator and return the unit result `tt`:

```
Program Definition prodcons_parcomp Ap Ac : prodcons_parcomp_tp Ap Ac :=
  Do (par (producer Ap 0) [consumer Ac 0];;
      ret tt).
```

The key step of the proof of the specification of `prodcons_parcomp`, which combines the postcondition of the producer and the consumer into a complete information about the history of the shared stack, is the application of the parallel composition rule, implemented as a structural lemma `par_do`. The application of the lemma requires providing explicit postconditions for the spawned threads, which we do by supplying it with the arguments `r1` and `r2`:

```
apply: (par_do (r1:=producer_post Ap fp) (r2:=consumer_post Ac)) E _ _ _.
```

Finally, the procedure `producer_consumer` employs the hiding mechanism to exclude any external interference and make the producer and the consumer to be the only threads communicating through the share stack. This makes it possible to prove the following specification for it:

```
Definition producer_consumer_tp (Ap Ac: {array [finType of 'I_n] -> A}) :=
 {fp}, STsep [U] (fun i =>
           exists ip ic fc,
            [/\ l \notin sdom i,
                pv_self i = sent :-> null \+ ip \+ ic,
                is_array Ap fp ip & is_array Ac fc ic],
           fun (_ : unit) m =>
           exists mp mc mst h (p : ptr) fc,
            [/\ pv_self m = mp \+ mc \+ mst,
                is_array Ap fp mp, is_array Ac fc mc,
                mst = sent :-> p \+ h &
                perm (fgraph fp) (fgraph fc)]).
```

Specifically, the precondition states that the thread-local state currently contains the entry with the sentinel pointer `sent :-> null`, which will be required to allocate a new Treiber concurroid structure, as well as two regions of memory, `ip` and `ic`, which are both arrays, as described by the conjuncts `(is_array Ap fp ip)` and `(is_array Ap fp ip)`. The label `l` for the Treiber concurroid to be allocated shouldn't be present in the state yet (`l \notin sdom i`). The postcondition, among other things, states that that the contents of the producer array remained unchanged `(is_array Ap fp mp)`, and the contents `fp` and `fc` of the arrays `Ap` and `Ac` at the end of the procedure execution are identical modulo permutation `(perm (fgraph fp) (fgraph fc))`, which is exactly what we wanted to prove.

4.2.5 Flat combiner (`flatcombine.v`)

Flat combiner (FC) is a complex higher-order concurrent data structure, implementing synchronization of several threads accessing a critical resource and leveraging the helping/work stealing mechanism to achieve better cache locality. The

detailed intuition behind the formalization of the flat combiner protocol and its concurroid is described in [10, §5]. The implementation is provided by the file `flatcombine.v`.

The flat combiner structure `flat_combineT` with the constructor `FlatCombineT` contains a number of fields, capturing the components of the protocol, its parameters and their properties. In particular, a sequential data structure, serving as a flat combiner argument, should provide a number of methods to operate with it. To simplify encoding in Coq, each avaialable method is assigned a *code* from the fixed set of codes (the `fc_code` field of the structure). For each such code, the input type and the output type of the method should be specified (the fields `fc_inT` and `fc_outT`). The *interpretation* `fc_interp` for each code `c` returns a sequential procedure (*i.e.*, the actual implementation), associated with this code, such that the outcome of this procedure is consistent with the *validity predicate* `fc_R`, which relates the code, its corresponding input and output types, the initial and the final contribution.

```
Structure flat_combineT := FlatCombineT {
  ...
  fc_code : Type; (* code for each method *)
  fc_inT : fc_code -> Type; (* input type of a coded method *)
  fc_outT : fc_code -> Type; (* output type of a coded method *)
  (* Specification of method c wrt. input/output, initial auxiliaries and contribution *)
  fc_R : forall c, fc_inT c -> fc_outT c -> fc_pcm -> fc_pcm -> Prop;
  fc_interp : forall pv al asent alk c (v : fc_inT c),
    {g}, STsep [entangle (Priv pv) (AllocCAS al asent alk)]
      (fun i => fc_inv g (self (pvg pv) i),
       fun r m => exists g',
         [/\ valid (g \+ g'), fc_inv (g \+ g') (self (pvg pv) m) & fc_R v r g g']);
  fc_uniq : ...
}.
```

Finally, the `fc_uniq` is the proof that for each coded procedure, operating with the sequential data structure, with which the flat combiner is instantiated, the contribution is uniquely determined for each input and output values. This property is crucial for establishing useful stable specification of the `do_help` action, defined later. Notice, that for each procedure operating with the argument data structure `fc_interp` fixes the concurroid it can only operate with: `[entangle (Priv pv) (AllocCAS al asent alk)]`, which indicates that the procedure "looks like a sequential one" (in fact, it might allocate concurroids via hiding and fork new lexically-scoped threads internally).

The request-help-acknowledge protocol, implementing the work-stealing machinery, is encoded using the auxiliary algebraic datatype `req`, defined as follows:

```
Structure req (T : flat_combineT) :=
  NoReq (* thread is not making a request *)
| Req (c : fc_code T) of fc_inT c (* thread wants to run method c, with input iT c *)
| Ans (c : fc_code T) of fc_outT c. (* thread got back result of c *)
```

These three constructors corresponds to the states, depicted in the figure on page 20 of the paper [10]. The structure of the concurroid's state and its coherence predicate are reminiscent to the one of locks (§4.1.1), and are described in detail in the paper [10], so we don't focus on them here.

The FC concurroid has five kinds of transitions, two of which are "standard" external ones, inherent for the locking/unlocking protocol, and the other three implement the work stealing/acknowledging mechanism. We will quickly outline the transitions, referring to the sections in the code, defining them:

- `ReqHelpTrans` — an internal transition implementing requesting the help by the thread, whose thread id is `tid`. The corresponding relation's conjunct `(nat_of_ord tid \in dom (get_tid (self s)))` indicates that the thread should indeed hold the id it requests help for.
- `CollectHelpTrans` — an internal transition corresponding to collecting the help (*i.e.*, the result and the generic client-specific auxiliary contribution) by a thread with the corresponding thread id. This transition has two possible outcomes, depending on whether the thread has been helped already (in which case it successfully collects the results and writes the `NoReq` value to the helper array cell) or not (in which case everything remains unchanged).
- `DoHelpTrans` — an internal transition, that can be invoked only by the thread holding the lock and "donating" some part of its own contribution to a thread, which requested the help, by writing to the corresponding cell of the helping array.

- `AcquireTrans` — an external transition, corresponding to releasing the lock, and, hence, changing the ownership over the critical resource heap back to the FC concurroid's joint part.
- `ReleaseTrans` — an external transition, corresponding to successful taking the FC lock by the current thread and transferring the protected heap to the thread-local state.

We don't explain the definitions of actions in detail, as they either correspond to specific internal transitions, or implement the heap ownership transfer via a `CAS`-like operation, accommodating the FC's acquiring transition in the manner, similar to the lock implementation (§4.1.2). The section `Stability` stability establishes a number of facts about assertions about FC's state, that are stable with respect to interference and gives stable Hoare-style specifications to the actions defined previously (see definitions `req_help`, `do_help`, `try_collect` *etc*).

The definition of the main FC program is split into several parts. First, we define, specify and verify the main "helping loop" of the program, which is performed by the thread, once it has become a *combiner*, *i.e.*, it has acquired the lock, and has made a request itself before:

```
Definition help_loop_tp := {h s g}, STsep [W]
  (fun i => exists hr,
     [/\ pv_self i = h \+ hr, fc_self i = s,
         get_mtx (fc_self i) = up own, made_request tid g v i &
         I (getU (fc_self i \+ fc_other i) \+ sum_up (fc_getg i)) hr],
   fun (_ : unit) j => exists hr,
     [/\ pv_self j = h \+ hr, fc_self j = s, made_request tid g v j &
         I (getU (fc_self j \+ fc_other j) \+ sum_up (fc_getg j)) hr]).

Program Definition help_loop :=
  ffix (fun (help_loop : nat -> help_loop_tp) t =>
    Do (if @decP _ (t < n) idP is left pf return _ then
          x <-- [read_request tid (Ordinal pf)];
          if x is Req m' v' then
            w' <-- [framed_interp v'];
            [do_help (Ordinal pf) w'];;
            help_loop t.+1
          else help_loop t.+1
        else ret tt)) 0.
```

As the specification `help_loop_tp` indicates, the lock is owned (`get_mtx (fc_self i) = up own`) at the beginning and at the end of the loop. Another conjunct of the loop invariant states that the data structure-specific invariant `I` holds on the resource heap `hr` with respect to *all* the combined pieces of auxiliary state at the beginning and at the end of each loop iteration: `I (getU (fc_self i + fc_other i) + sum_up (fc_getg i)) hr`.

Next, we implement the logic of acquiring the lock and, in the case of success, helping all the threads, by implementing the program `cas_help`, which runs `help_loop` when the FC lock is acquired, or simply returns otherwise.

```
Program Definition cas_help : {h s g}, STsep [W]
  (fun i => [/\ pv_self i = h, fc_self i = s, get_mtx (fc_self i) = up nown &
               made_request tid g v i],
   fun (_ : unit) j =>
           [/\ pv_self j = h, fc_self j = s &
               made_request tid g v j]) :=
  Do (x <-- [cas T pv];
      if x then
        help_loop;;
        [unlock T pv]
      else ret tt).
```

The pre- and postconditions contain the conjunct `(made_request tid g v j)`, which indicates that the thread has made a request for help, but hasn't yet tried to collect the result (so it doesn't know whether it has been helped or not).

As the next step, we implement the "external" loop. In this loop, the thread, which is assumed to have requested the help for itself already, first tries to acquire the lock and help everyone via `cas_help`. Independently of its success or failure, it then tries to collect the help for itself. If the help is present (the result `ow` is `Some w`), the procedure returns the result, otherwise it re-iterates, since the thread hasn't been helped yet.

```
Definition cas_loop_tp := {h s g}, STsep [W]
  (fun i => [/\ pv_self i = h, fc_self i = s, get_mtx s = up nown &
               made_request tid g v i],
   fun (w : fc_outT m) j =>
     exists g' g1,
     [/\ pv_self j = h, fc_self j = (get_tid s, up nown, getU s \+ g'),
         [g1 <<= getU (fc_self j \+ fc_other j) \+ sum_up (fc_getg j)],
         [g <<= g1], fc_R v w g1 g' & no_reqs j]).

Program Definition cas_loop :=
  ffix (fun (cas_loop : unit -> cas_loop_tp) xx =>
    Do (cas_help;;
        ow <-- [try_collect tid v];
        if ow is Some w return _ then ret w
        else cas_loop tt)) tt.
```

Finally, the `flat_combine` procedure first requests the help to the thread `tid` via the `req_help` command, and then enters the `cas_loop` attempting to collect the help and help the others.

```
Program Definition flat_combine : {h s g}, STsep [W]
  (fun i => [/\ pv_self i = h, fc_self i = s,
               [g <<= getU (fc_self i \+ fc_other i) \+ sum_up (fc_getg i)],
               get_mtx s = up nown & no_reqs i],
   fun (w : fc_outT m) j =>
     exists g' g1,
     [/\ pv_self j = h, fc_self j = (get_tid s, up nown, getU s \+ g'),
         [g1 <<= getU (fc_self j \+ fc_other j) \+ sum_up (fc_getg j)],
         [g <<= g1], fc_R v w g1 g' & no_reqs j]) :=
   Do ([req_help tid v];;
       cas_loop).
```

The postcondition of `flat_combine` indicates that upon termination the thread has acquired an additional contribution `g'`, which is expressed through its auxiliary state (`getU s + g'`). Moreover, this contribution is properly related (`fc_R v w g1 g'`) to the input value `v`, result value `w` and the "intermediate" cumulative value `g1` of all auxiliaries, which happened to be observed during the execution of the procedure (`g` captures the "initial" cumulative value of all auxiliaries):

```
g <<= g1 <<= getU (fc_self j \+ fc_other j) \+ sum_up (fc_getg j)
```

More intuition on this specification is provided in [10, §5].

4.2.6 Instantiating flat combiner for a stack (`stack_combine.v`)

We now instantiate the flat combiner procedure with a sequential stack, not adapted for concurrent use and requiring exclusive access to be modified.

The file `stack_combine.v` defines all the components, necessary for instantiating the `FlatCombineT` constructor of the FC structure. The stack comes with two methods, push and pop, each is associated with the corresponding code, defined by the following structure:

```
Structure code := pushc | popc.
```

The definitions of input/output types and general specification for stack operations (`inT`, `outT`, `specR`) return the appropriate types and validity predicates, depending on the code provided. Finally, the interpretation function `interp` returns an appropriate *interpretation* (*i.e.*, the actual program code) for push ot pop (`seq_push` and `seq_pop`) depending on the code `c`.

All these components are used to instantiate the flat combiner with the stack structure:

```
Definition T :=
  @FlatCombineT a lk n fc [encoded_pcm of hist st] inv prec_inv
               code inT outT specR interp uniqR.
```

Notice, that the actual implementation of sequential push and pop *are not* provided by this module, as `seq_push` and `seq_pop` defined as variables (*i.e.*, they are *assumed* to be provided by the client of this module), which make the module

parametrized by actual implementations. Therefore, any client of this module will have to provide suitable sequential implementations of push and pop, consistent with the types of `seq_push` and `seq_pop`, correspondingly.

The module concludes with the proofs that the flat combiner, run with the codes (*i.e.*, `pushc` or `popc`), corresponding to push and pop, can be given expectable specifications.

```
Program Definition fc_push (tid : 'I_n) (e : A) : {h : hist st}, STsep [W]
  (fun i => [/\ pv_self i = Unit, fc_self i = (#nat_of_ord tid, up nown, Unit),
               [h <<= getU (fc_other i) \+ sum_up (fc_getg i)] &
               @no_reqs T tid i],
   fun (_ : unit) m => exists t ls,
      [/\ pv_self m = Unit,
          fc_self m = (#nat_of_ord tid, up nown, t \\-> (ls, e :: ls)),
          [h <<= getU (fc_other m) \+ sum_up (fc_getg m)], last_key h < t &
          @no_reqs T tid m]) :=
  Do (@flat_combine' T pv al asent alk tid pushc e).
```

Let us compare the specification of `fc_push` with the spec of Treiber stack's push on page 16. The specifications are very similar in the sense that both assume initially empty self-history (`tb_self i = Unit` *vs.* `fc_self i = (..., ..., Unit)`), and both postulate the contribution to the history in the form of the singleton entry `(t \\-> (ls, e :: ls))`. Moreover, both specifications state that the allocated timestamp `t` corresponds to the moment during the procedure's execution (`last_key h < t`). The only differences are the additional auxiliaries (`#nat_of_ord tid, up nown`) and the conjunct (`@no_reqs T tid m`), indicating that the current thread is done with its operation, in the case of FC. These differences can be abstracted away using suitable abstract predicates. Development of such predicates for stacks is the future work.

4.3 Concurrent graph manipulations (`spanning.v`)

Folder: `Examples/Graphs`

In-place spanning tree construction of a directed binary graph is the main example of the paper [9]. Its mechanized development can be informally described in several stages:

- Description of a domain-specific library, describing trees, graphs and auxiliary definitions, such as `front` and `maximal` in the file section `Definitions`.
- Definition of `graph`- and tree-shaped heaps, as well as the definitions of the `contents` function (section `GraphsHeaps`).
- We next define a number of heap graph-manipulating functions for removing edges (*e.g.*, `null_edge`) and marking nodes (*e.g.*, `mark_node`) and prove several lemmas about them (*e.g.*, `null_edgeG`, `mark_nodeG` *etc*), establishing, in particular, monotonicity results, describing the behaviour of `front` and other predicates when the graph is being reduced via removal of edges (*e.g.*, `front_mono`).
- The module `SpanResourceInvariant` defines the spanning tree concurroid state-space (`coh` predicate) and proves its properties (such as `sp_coh_zigzag`). It concludes with instantiating the standard FCSL interface `CohPredMixin` for concurroid coherence predicates.
- The following module `SpanTransitions` defines the transitions of the concurroid. In particular, the section `MarkTransition` describes the transition, corresponding to marking the node, and the section `NullifyTransition` defines the transition nullifying an edge. Both transitions along with their mandatory properties (such as `mark_zagzig`) are used to instantiate *internal* transition interface `IntGuarMixin`.[6] Finally, the coherence predicate together with the transitions is packaged into the concurroid structure `SpanTree` via FCSL's `Mod.Core` and `Mod.Make` constructors.
- Module `SpanActions` combines transitions into *atomic actions*, which also have operational meaning of commands, such as `write` or `CAS` (sections `TryMarkAction`, `NullifyAction`, and `ReadChildAction`). For example, the `trymark_step` defines the semantics of "try-to-mark" action, operationally equivalent to the `CAS` command of the spanning tree algorithm. With a number of properties proved, we can instantiate the `Action` interface for actions, which is then given to the `IntAction` constructor, along with the proof of the "erasure" property (*e.g.*, `trymark_step_erase`), stating that the result of the action doesn't depend on the auxiliary state, *i.e.*, it's operational. The last action, `read_child_act` correspond to the *idle* transition of the concurroid, since it doesn't change any state, but just examines it.

[6] Since the spanning tree concurroid doesn't obtain or give away any heap fragments, it has only *internal* transitions, which don't change the footprint of the concurroid-owned heap. *External* transitions, necessary for composing concurroids, will be shown in §4.1.1.

- Section SpanStabilityMain provides *stable* specifications to the defined above actions. To do so, it proves a number of lemmas, describing the behaviour of the subgraph predicate, packaging together all relevant stability-related properties, while more nodes are being marked and more edges are removed. Next, the Hoare-style specifications to the atomic actions are given. In particular, one can observe that the following spec is stable.

```
Program Definition trymark x :
  {s1 (g1 : graph (joint s1))}, STsep [W]
    (fun i => s1 = i /\ x \in dom (joint i),
     fun b s2 => exists (g2 : graph (joint s2)),
       [/\ subgraphT g1 g2,
           self s2 = if b then #x \+ self s1 else self s1,
           mark g2 x = true &
           b -> contents g1 x = (false, edgl g2 x, edgr g2 x)]) :=
  Do (act (trymark_act sp x)).
```

  However, the stability of the pre- and postconditions given to trymark should be proved formally out of the atomic specification of the trymark_act action, via the defined above lemmas, such as subgr_steps. This proof follows the Next Obligation command and completes with Qed. The same obligations are proven for the specifications of read_child and nullify that follow.

- Next, there comes the specification of the program span itself. For convenience, the postcondition is extracted as a separate predicate span_post, which is then used in the spec. The proof script is annotated at the points of main stages, which should help the reader to notice the applications of Hoare-style structural lemmas, such as val_ret, step, par_do *etc*, and follow the general flow of the proof.
- Module SpanDecorate defines the *decorator* predicate, necessary for allocation of the spanning tree concurroid from the thread-local state. A number of properties (*e.g.*, dec_onto and dec_env) have to be proven, so we could eventually instantiate the Decorate interface for the spanning tree concurroid.
- Finally, we can prove the specification span_root_tp for the topmost call of the span procedure, which is enclosed into the hiding constructor using the above defined decorator as follows:

```
Program Definition span_root x : span_root_tp x :=
  Do (@priv_hide pv empty_mod (SpanTree sp) bool _ (SpanDecorate sp) (Unit, h)
     (extend _ (span sp x))).
```

  Two obligations are emitted. The first obligation, which is trivially discharged, requires us to establish the validity of injection of the procedure span operating on the concurroid (SpanTree sp) into a larger concurroid (entangle (Priv pv) (SpanTree sp)). The second obligation requires proving weakening of the precondition and strengthening of the postcondition in the presence of hiding. The key step of the following proof is the script statement apply: val_privhide', which applies the logical rule for hiding, implemented as a lemma in lemmas.v.

4.4 Non-linearizable data structures and their clients

Folder: Examples/NonLinearizable

For the details, see the descriptions in the README.md file of the project and the accompanying manuscript [11].

4.4.1 Concurrent Data Structures Linked in Time

Folder: Examples/Relink

This folder contains the main case study of the *Linking in Time* technique for verifying concurrent data structures with non-fixed linearization points introduced in [1]. It consists of several files implementing the verification in FCSL of Jayanti's single writer/single scanner snapshot construction [3]. In the sequel, we relate the different parts of the development with their corresponding presentation in the paper.

The invariants described in Section 5, together with the implementation of the concurrent resource for the snapshot object are defined in the jayanti_ghoststate.v and jayanti_concurroid.v files. In the later file, there is also the implementation of the auxiliary code functions (a.k.a transitions in FCSL jargon) from Section 6.

The proof of the 2-state invariants from Invariant 1, and those from Invariant 2, together with the definitions of the main assertions (e.g. the *stable order* Ω) are carried out in jayanti_stability.v. The verification of the write and scan files, as presented in Section 7, are carried our in the file jayanti_library.v.

Finally, the clients in Section 4 are implemented and verified in `jayanti_clients.v`. For further details, we refer the reader to the descriptions in the `README.md` file of the project and the accompanying manuscript [1].

References

[1] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Concurrent data structures linked in time. In Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP 2017), to appear. Pre-print available from http://arxiv.org/abs/1604.08080.

[2] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of the 24th European Conference Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

[3] Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *STOC*, pages 723–732. ACM, 2005.

[4] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2013)*, pages 561–574. ACM, 2013.

[5] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems (ESOP 2014)*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014. Extended version is available from http://software.imdea.org/fcsl/papers/concurroids-extended.pdf.

[6] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2010)*, pages 261–274. ACM, 2010.

[7] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Th. Comp. Sci.*, 375(1-3):271–307, 2007.

[8] Ilya Sergey. *Programs and Proofs: Mechanizing Mathematics with Dependent Types*. Lecture notes with exercises. Available at http://ilyasergey.net/pnp, 2014.

[9] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, pages 77–87. ACM, 2015.

[10] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP 2015)*, volume 9032 of *LNCS*. Springer, 2015. Extended version is available from http://software.imdea.org/fcsl/papers/histories-extended.pdf.

[11] Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*, pages 92–110. ACM, 2016.