Separation Logic and Concurrency (OPLSS 2016) Draft of July 22, 2016

Aleks Nanevski

IMDEA Software Institute
aleks.nanevski@imdea.org

Contents

Introduction

Separation logic, developed by O'Hearn, Reynolds and collaborators [15, 23, 28], has emerged as an effective logical framework for specifying and verifying pointer-manipulating programs. This course will cover the basic sequential separation logic, and then build on it to explain some of the main ideas behind the recent results in the verification of shared-memory concurrency.

We first introduce separation logic in its standard formulation, to give the reader a foundation for independent study of this broad and active research field. We then proceed to reformulate this standard, to set us up for introducing ideas from cutting edge research on verification of shared-memory concurrent software. Though some of these cutting-edge ideas are highly experimental and still in their infancy, learning them will be beneficial to a reader who is looking to enter the research field of verification, concurrency, or a related topic.

Through many examples, we will illustrate the specification and verification patterns that provide effective reasoning about lock-based and lock-free concurrent programs over pointer-based data structures and programs, including linearizable and non-linearizable pointer structures and programs, and pointer structures with so-called "deep sharing", such as graphs.

The presentation will be heavily bent towards showing how to carry out concrete proofs of concrete programs. The emphasis will be on analyzing the mathematical structure of the underlying problem that a program is trying to solve, and then translating the obtained insights into language and logic designs that enable effective program correctness proofs. We will not spend time on meta theory; proofs of soundness and/or completeness of the various presented formal systems are in the literature.

Along the way, we will point out the many high-level connections between separation logic and type theory. In fact, the author believes that Separation Logic is essentially a type theory (of state), even though it was not actually discovered as such. For, if you start constructing a type theory for effects of state, and then continue to concurrency, your hand will be guided in a fairly straightforward path towards a formulation of separation logic. These notes will cover precisely a path of this form, almost exactly in the way that the author has traveled it, with collaborators: for sequential separation logic, Greg Morrisett, Lars Birkedal, Amal Ahmed, and for concurrency, Ruy Ley-Wild, Ilya Sergey, Anindya Banerjee and German Delbianco.

Chapter 1

Separation logic for sequential programs

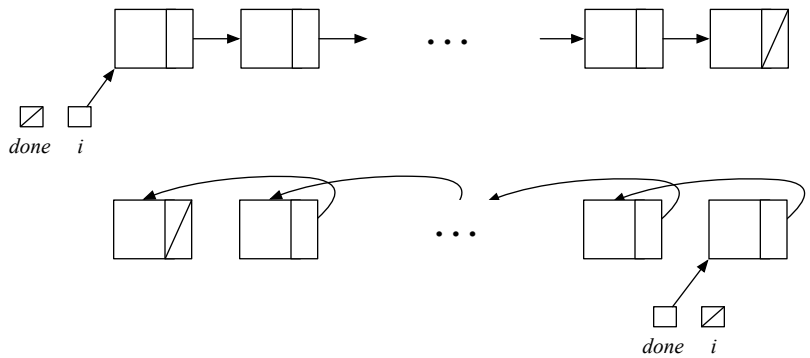
1.1 Motivating example

Separation logic is a form of Hoare logic, intended for reasoning about programs with pointers. Let us introduce it by means of an in-place list reversal example, a classic example taken from Reynolds [28].

The example begins with a singly-linked list in the heap, which it reverts in place by swinging, in order, the next pointer of each node to point not to the next, but to the previous node.

$$
\begin{array}{l}
\mathsf{reverse} = \; \mathit{done} \leftarrow \mathsf{null}; \\
\qquad\quad \mathsf{while}\ (i \neq \mathsf{null})\ \mathsf{do}\ \{ \\
\qquad\qquad k \leftarrow\ !(i+1); \\
\qquad\qquad i+1 := \mathit{done}; \\
\qquad\qquad \mathit{done} \leftarrow i; \\
\qquad\qquad i \leftarrow k; \\
\qquad\quad \}
\end{array}
$$

Pictorially, `revert` modifies the first list below, into the second one.



We want to prove that if initially i points to a list whose contents is the sequence α_0, then in the end, *done* points to $\mathsf{rev}\ \alpha_0$. Here α_0 is a mathematical list, i.e., an element of the inductive type list in ML or Haskell, defined with the constructors nil and :: (read cons):

$$\mathsf{list} = \mathsf{nil} \mid :: \mathsf{of}\ \mathsf{nat} \times \mathsf{list}$$

Thus, we use purely-functional lists to specify programs that manipulate pointer-based lists. In particular,

to specify this program, we need the function rev, defined inductively on the size of the list:

$$\begin{array}{rcl} \mathsf{rev}\ \mathsf{nil} & = & \mathsf{nil} \\ \mathsf{rev}\ (x :: xs) & = & \mathsf{rev}\ xs +\!\!+ (x :: \mathsf{nil}) \end{array}$$

where $++$ is concatenation of lists, which is also easily defined by induction on xs, as follows.

$$\begin{array}{rcl} \mathsf{nil} +\!\!+ ys & = & ys \\ (x :: xs) +\!\!+ ys & = & x :: (xs +\!\!+ ys) \end{array}$$

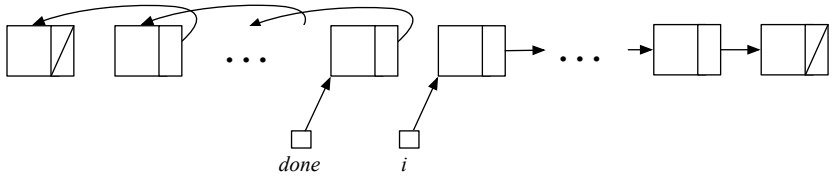
To formally write the precondition that *in the beginning* i *points to a list storing* α_0, and the postcondition that *in the end done points to a list storing* rev α_0, we need a way to describe the state (or the heap) occupied by a *singly-linked* list storing some given mathematical sequence. We use a predicate is_list with two input arguments: p and α. The first input argument, p, is the pointer heading the list, and the second input argument, α is the mathematical sequence that is being stored.

$$\mathsf{is\_list}\ p\ \alpha$$

We keep this predicate undefined for now, but will define it soon to say precisely that the *current heap* (which will remain implicit in the definition), stores a singly-linked list whose contents is α, starting from the pointer p. Then we would like to prove the following Hoare triple specification for reverse:

$$\begin{array}{c} \{\mathsf{is\_list}\ i\ \alpha_o\} \\ \mathsf{reverse} \\ \{\mathsf{is\_list}\ \mathit{done}\ (\mathsf{rev}\ \alpha_0)\} \end{array}$$

Now, i stores the initial head of the list and *done* stores the ending head of the list. But, if one looks a bit deeper into the loop of the program, rather than just the beginning and the ending state, then the variable i identifies the sub-list that remains to be reversed (i.e., pointer to the list node at which we currently are), and *done* identifies the sub-list that has already been reversed. What each iteration of the loop does, is move the next node from sub-list i to sub-list *done*.



In program verification, one formally describes the behavior of loops by so-called *loop invariants*, i.e., properties that hold throughout the loop execution. In our particular case, the loop invariant should say something like the following (which is not quite right, but is close enough for a start).

$$\exists \alpha\ \beta.\ \mathsf{is\_list}\ i\ \alpha \wedge \mathsf{is\_list}\ \mathit{done}\ \beta \wedge \mathsf{rev}\ \alpha_0 = (\mathsf{rev}\ \alpha) +\!\!+ \beta$$

Here, α and β are again purely-functional lists, describing the contents of the singly-linked lists headed at i and *done*.

When we are at the beginning of the loop, then $\alpha = \alpha_0$ and $\beta = \mathsf{nil}$, as we didn't invert anything yet. So, we can see that the equation from the loop invariant trivially holds:

$$\mathsf{rev}\ \alpha_0\ = (\mathsf{rev}\ \alpha_0) +\!\!+ \mathsf{nil}.$$

Moreover, the whole loop invariant can be trivially inferred from the precondition $\mathsf{is\_list}\ i\ \alpha_0$ and the knowledge that *done* points to an empty list (i.e., $\mathit{done} = \mathsf{null}$).

When we are at the end of the loop, then $\alpha = \mathsf{nil}$ and $\beta = \mathsf{rev}\ \alpha_0$, as we have inverted everything. So, the equation holds again:

$$\mathsf{rev}\ \alpha_0\ = (\mathsf{rev}\ \mathsf{nil}) +\!\!+ (\mathsf{rev}\ \alpha_0).$$

Moreover, from the loop invariant, and $\alpha = \mathsf{nil}$, we can derive the postcondition $\mathsf{is\_list}\ \mathit{done}\ (\mathsf{rev}\ \alpha_0)$.

We also need to show that the above proposition is indeed a loop invariant; that is, it holds from one loop iteration to the next. Specifically, in this particular case, we should prove that the loop body satisfies the following property: if we start with i storing $x::xs$, and *done* stores ys, then when the iteration finishes, we have i storing xs and *done* storing $x::ys$. That is:

$$\begin{array}{l}\{\mathsf{is\_list}\ i\ (x :: xs) \wedge \mathsf{is\_list}\ \mathtt{done}\ ys \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ (x :: xs) +\!\!+ ys\} \\ \quad k \leftarrow !(i+1); \\ \quad i+1 := done; \\ \quad done \leftarrow i; \\ \quad i \leftarrow k; \\ \{\mathsf{is\_list}\ i\ xs \wedge \mathsf{is\_list}\ done\ (x :: ys) \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ xs +\!\!+ (x :: ys)\}\end{array}$$

Obviously, the equation about $\mathsf{rev}\ \alpha_0$ in the postcondition will be easy. This equation, and the similar equation from the precondition, talk only about immutable mathematical entities α_o, x, xs and ys. Thus, the equation from the precondition remains invariant under all the mutations of the program, and eventually, proves the equation from the postcondition, since we know, just from purely-functional properties of lists that

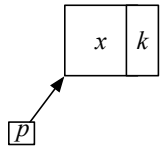
$$\mathsf{rev}\ (x :: xs) +\!\!+ ys = \mathsf{rev}\ xs +\!\!+ (x :: ys).$$

The difficulty in verifying this program is in establishing the other components of the postcondition, namely, $\mathsf{is\_list}\ i\ xs$ and $\mathsf{is\_list}\ done\ (x :: ys$, from the initial $\mathsf{is\_list}\ i\ (x :: xs)$ that we have in the precondition. In fact, the difficulty, resolved by the main insights of separation logic, is in defining $\mathsf{is\_list}\ p\ \alpha$ in such a way that we can work with these other component effectively. So how do we define $\mathsf{is\_list}\ p\ \alpha$?

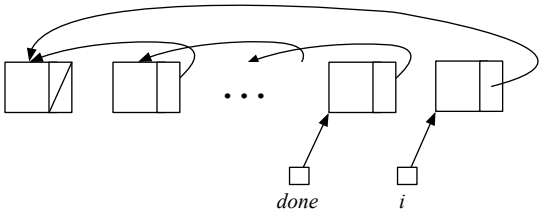
We can start with the following attempt, basically just constraining the various variables that we have in the program. We make the definition inductive on the list α.

$$\begin{array}{lcl}\mathsf{is\_list}\ p\ \mathsf{nil} & \widehat{=} & (p = \mathtt{null}) \\ \mathsf{is\_list}\ p\ (x :: xs) & \widehat{=} & \exists q.\, p \hookrightarrow (x; q) \wedge \mathsf{is\_list}\ q\ xs\end{array}$$

Here we take $p \hookrightarrow (x; q)$ to be an atomic proposition, indicating that the heap cell at address p stores a node whose value field equals x, and whose next field equals q.



It turns out, however, that the above is too simplistic, and doesn't quite work. In particular, as Reynolds points out, defining $\mathsf{is\_list}$ as above, actually leads to an incorrect loop invariant, because the resulting proposition is not strong enough for our purposes. In particular, the definition of $\mathsf{is\_list}$ allows there to be *sharing* of nodes between the list at i and the list at *done*.



Referring to the above figure, one can see that it satisfies the conjunction $\mathsf{is\_list}\ i\ \alpha \wedge \mathsf{is\_list}\ done\ \beta$, for some α and β. The conjunction allows the two lists to overlap (in the case of the figure, the lists overlap in the last node).

Obviously, if there is sharing of nodes between the lists at i and the list at *done*, at the beginning of the loop, then running the loop body on a shared node, will reduce the list at i, but may potentially mangle the list *done*. For example, if the loop body or reverse is executed twice in the situation in the above figure, it will create a cycle in the ending list. Thus, the ending list starting at *done* will certainly not satisfy the predicate *is_list*.

Of course, important observation about reverse is that no sharing can actually occur in practice between the sub-lists at i and *done*. Indeed, the sub-list *done* starts empty (hence, no sharing in the base case), and when we add a node to *done*, we remove it from the sub-list i (hence, no sharing in the inductive case either). But, our first approximation of the loop invariant does not capture that fact.

1.2 Splitting the heap and definition of separation connectives

We strengthen the invariant as follows. We need to change the definition of is_list, so that it asserts the *disjointness* of the nodes in the sub-list i and sub-list *done*. Notice, this cannot be achieved by just referring to the contents of the nodes, but we have to talk about the underlying heap addresses in which the nodes reside. This also explains why we need the assertions to talk about the *current* heap (a point we already illustrated in our naive definition of is_list). In particular, our assertions can't just scope over the various program variables; we need to relate the values of these variables to the value of the heap, even though the heap is itself *not* a variable that's explicitly used by the program.

Note 1.2.1 (Stating disjointness using reachability).
Reynolds [28] considers a way of stating the disjointness property by enumerating what's reachable from the program variables. However, he shows by example that, while such an approach is theoretically possible, expressing heap disjointness this way *doesn't scale*, in the sense that one is led to writing very convoluted assertions, with a lot of tedious overhead.

Separation logic way of stating the desired disjointness involves introducing several new propositions and propositional connectives about heaps. We explain them through a new, and this time correct, definition of is_list, which is, again, inductive on the sequence of contents (the second input argument).

$$\begin{array}{lcl} \mathsf{is\_list}\ p\ \mathsf{nil} & \widehat{=} & (p = \mathtt{null}) \wedge \mathsf{emp} \\ \mathsf{is\_list}\ p\ (x :: xs) & \widehat{=} & \exists q.\, p \mapsto (x; q) * \mathsf{is\_list}\ q\ xs \end{array}$$

In the first clause of the definition, we can see that is_list p nil now uses the atomic predicate emp, to explicitly say that the heap occupied by the list headed at p, but which contains no nodes, *is actually empty*. In addition, p must be the null pointer.

In the second clause, we see the connective $*$, called "separating conjunction", and another connective $p \mapsto (x; q)$, which we again read as "heap cell at address p stores a node whose value field equals x, and next field equals q", as we did in the incorrect setting with $\hookrightarrow$ before.

The clause says that in the case our list contains nodes storing x and xs, then the heap circumscribed by the list can be divided into two *disjoint parts*. The first part satisfies $p \mapsto (x; q)$; that is, it contains a pointer p pointing to a node with a value x and next pointer q (as before), *and nothing else*. The second part satisfies is_list q xs. As the two parts are disjoint, the node identified by p (storing x and q), does not appear in the list headed at q, i.e., we have no sharing between these two parts. Thus, inductively, the whole list headed at p exhibits no sharing of nodes.

Actually, the situation is typically described a bit differently. We don't take $p \mapsto (x; q)$ as a primitive proposition, as it is really tailored to describing nodes with only two fields: one for the value, one for the pointer to the next node. Obviously, that is not suitable for more complex data structures such as doubly-linked lists, where we need two pointers, one for the next, one for the previous node. The separation logic default is to have one atomic proposition:

$$p \mapsto x$$

(read p *points to* x), and then express more complex ones as compounds with $*$. E.g., what we called $p \mapsto (x; q)$ above, may be expressed as $(p \mapsto x) * (p + 1 \mapsto q)$ in an idealized memory model where we assume that fields of a list node are stored in *consecutive* memory locations.

With this definition of is_list, the loop invariant for reverse can be re-stated as

$$\exists \alpha\ \beta.\, \mathsf{is\_list}\ i\ \alpha * \mathsf{is\_list}\ \mathit{done}\ \beta \wedge \mathsf{rev}\ \alpha_0 = (\mathsf{rev}\ \alpha) +\!\!+ \beta$$

As the invariant now uses both $*$ and $\wedge$, let us rule out any possible confusion between the separating and the ordinary conjunction, and immediately *formally* define the meaning of the new operators.

First, we need to explain what a heap is. We will follow the usual separation logic approach, and take a model where a heap is a finite list of pointers, which store some values. Mathematically, you can say that a heap is a function with a finite domain.

$$\mathsf{heap} = \mathsf{ptr} \rightharpoonup_{\mathsf{fin}} V$$

But you can also think of it more concretely as a list of pairs

$$(p_1, x_1), (p_2, x_2), \ldots, (p_n, x_n)$$

sorted by the first component. Here, p_i are the pointers that you have allocated, and x_i are their values. For uniqueness of representation, we assume that $p_1 < p_2 < \cdots < p_n$. Pointers are natural numbers. However, a heap cannot contain the natural number 0 (which we also call the null pointer) in its domain.

The domain of values, we keep unspecified. Often times in separation logic, the values are just taken to be natural numbers. This is clearly, highly idealized, because that assumes arbitrary precision nats. A more low-level memory model would take the values to be fixed sized machine integers. Although such considerations are important for verifying realistic programs, in these notes we'll avoid them, as we want to keep the focus on the more high-level verification issues. In fact, we will be even more idealized than customary in separation logic, and allow that the set of values encompasses all well-typed terms from the set universe of the Calculus of Inductive Constructions. So, all natural numbers, booleans, pairs, *monomorphic* functions, etc. This *does* exclude *higher-order heaps*, i.e., storing into a heap functions over heaps, which is what heap-manipulating programs are. Enabling higher-order heaps is a very interesting model-theoretic topic, but, while it allows more programs to be verified, it is not necessary for understanding the issues that will be discussed in these notes, nor for motivating and explaining the verification abstractions that we will introduce.

With those caveats out of the way, we can say that a separation logic assertion is a predicate over a heap. It is customary to write $\sigma \models P$ when we want to say that P holds of the heap σ. Then we have the following table giving the appropriate definitions.

σ	$\models$	emp	if $\sigma = \mathsf{empty}$
σ	$\models$	$p \mapsto v$	if $\sigma = p \mapsto v$
σ	$\models$	$P * Q$	if $\sigma = \sigma_1 \cup \sigma_2$, such that $\sigma_1 \models P$ and $\sigma_2 \models Q$
σ	$\models$	$\top$	always
σ	$\models$	$P \wedge Q$	if $\sigma \models P$ and $\sigma \models Q$
σ	$\models$	$P \implies Q$	if $\sigma \models P$ implies $\sigma \models Q$
	$\vdots$		

In English, the heap σ satisfies emp if it equals the empty finite map, which we call empty. The heap σ satisfies $p \mapsto v$, if it is a singleton map consisting of a single pointer p, storing v, and nothing else. When defining $*$ we split the heap σ into a disjoint union of σ_1 and σ_2, and request that P and Q hold of the two pieces. In contrast, when defining $\wedge$, we don't split the heap at all, but require that P and Q hold of the same heap σ. Similarly for implication. Of course, we will use the other standard propositional connectives (disjunction, negation, equality, inequalities, quantifiers, etc), but we omit their formal definitions, as they are easy to fill in. In particular, we don't split the heap for them.

To familiarize ourselves with these connectives, let us consider a few implications between them. First, as we already said, `null` is not a pointer that can store anything, and thus:

$$\mathsf{null} \mapsto x \implies \bot$$

Disjoint heaps cannot contain a pointer in common, hence:

$$p \mapsto x * p \mapsto y \implies \bot$$

Or alternatively:

$$p \mapsto x * q \mapsto y \implies p \neq q$$

Conjoined predicates talk about equal heaps. For example:

$$p \mapsto x \wedge p \mapsto y \implies x = y$$

Also,

$$p \mapsto x \wedge \mathsf{emp} \implies \bot$$

Propositions that don't talk about heaps (aka. pure propositions) hold of any heap, and can thus be re-associated arbitrarily. E.g.:

$$p \mapsto x * (q \mapsto y \wedge x > y) \iff (p \mapsto x \wedge x > y) * q \mapsto y.$$

Another important property is that emp is a unit for $*$, that is:

$$P * \mathsf{emp} \iff \mathsf{emp} * P \iff P$$

Also, $*$ is commutative and associative:

$$\begin{array}{rcl} P * Q & \iff & Q * P \\ P * (Q * R) & \iff & (P * Q) * R \end{array}$$

An important combination that we will use is a conjunction between a proposition describing a heap, and a pure proposition, e.g.:

$$p \mapsto x \wedge x > 3$$

is equivalent to

$$p \mapsto x * (x > 3 \wedge \mathsf{emp})$$

but *is not* equivalent to

$$p \mapsto x * x > 3$$

because the latter hold of a heap $p \Mapsto 4 \cup q \Mapsto 5$, while the former does not.

Also:

$$\mathsf{is\_list}\ i\ \alpha \wedge i \neq \mathsf{null} \implies \exists x\ xs.\ \alpha = x :: xs$$

which can be proved by case-analysis on α.

And the dual property:

$$\mathsf{is\_list}\ \mathsf{null}\ \alpha \implies \alpha = \mathsf{nil} \wedge \mathsf{emp}$$

Exercise 1.2.1.

- Draw a diagram describing the formula $x \mapsto (3, y) * y \mapsto (3, x)$.
- Draw a diagram describing $x \mapsto (3, y) \wedge y \mapsto (3, x)$.
- Draw the diagrams describing the conjuncts separately. Make sure to draw the dangling pointers.
- Write the inductive predicate $\mathsf{is\_tree}\ p\ \alpha$, that describes a heap layout of a singly-linked binary tree rooted at p, storing the contents described by the purely-functional tree α.

Now we can get back to our reverse example, and write out the proof outline for it, which lists the assertions that hold at each step of the program.

$\{\mathsf{is\_list}\ i\ \alpha_0\}$
$done \leftarrow \mathsf{null};$
$\{\mathsf{is\_list}\ i\ \alpha_0 \wedge done = \mathsf{null}\}$
$\{\mathsf{is\_list}\ i\ \alpha_0 * (done = \mathsf{null} \wedge \mathsf{emp})\}$
$\{\mathsf{is\_list}\ i\ \alpha 0 * \mathsf{is\_list}\ done\ \mathsf{nil}\}$
// the loop invariant holds
$\{\exists \alpha (= \alpha_0)\ \beta (= \mathsf{nil})\ q.\ \mathsf{is\_list}\ i\ \alpha * \mathsf{is\_list}\ done\ \beta \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ \alpha +\!\!+ \beta\}$
while $(i \neq \mathsf{null})$ do {
  $\{\exists \alpha\ \beta.\ \mathsf{is\_list}\ i\ \alpha * \mathsf{is\_list}\ done\ \beta \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ \alpha +\!\!+ \beta \wedge i \neq \mathsf{null}\}$
  // we know from $\mathsf{is\_list}\ i\ \alpha \wedge i \neq \mathsf{null}$ that $\alpha = x :: xs$
  $\{\exists x\ xs\ \beta.\ \mathsf{is\_list}\ i\ (x :: xs) * \mathsf{is\_list}\ done\ \beta \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ (x :: xs) +\!\!+ \beta\}$
  // unfolding $\mathsf{is\_list}\ i\ (x :: xs)$ and $\mathsf{rev}\ (x :: xs)$
  $\{\exists x\ xs\ k\ \beta.\ i \mapsto x; k * \mathsf{is\_list}\ k\ xs * \mathsf{is\_list}\ done\ \beta \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ xs +\!\!+ (x :: \beta)\}$
  $k \leftarrow !(i + 1);$
  // we now have a free variable for k, so remove the quantifier over k
  $\{\exists x\ xs\ \beta.\ i \mapsto x; k * \mathsf{is\_list}\ k\ xs * \mathsf{is\_list}\ done\ \beta \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ xs +\!\!+ (x :: \beta)\}$
  $i + 1 := done;$
  // write $done$ into $i + 1$
  $\{\exists x\ xs\ \beta.\ i \mapsto x; done * \mathsf{is\_list}\ k\ xs * \mathsf{is\_list}\ done\ \beta \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ xs +\!\!+ (x :: \beta)\}$
  // commute and reassociate $i \mapsto x; done$ with $\mathsf{is\_list}\ done\ \beta$
  $\{\exists x\ xs\ \beta.\ \mathsf{is\_list}\ k\ xs * \mathsf{is\_list}\ i\ (x :: \beta) \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ xs +\!\!+ (x :: beta)\}$
  $done \leftarrow i;$
  // since $done$ and i now equal, replace i by $done$
  $\{\exists x\ xs\ \beta.\ \mathsf{is\_list}\ k\ xs * \mathsf{is\_list}\ done\ (x :: \beta) \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ xs +\!\!+ (x :: \beta)\}$
  $i \leftarrow k;$
  // and replace k by i
  $\{\exists x\ xs\ \beta.\ \mathsf{is\_list}\ i\ xs * \mathsf{is\_list}\ done\ (x :: beta) \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ xs +\!\!+ (x :: \beta)\}$
  $\{\exists \alpha = (xs)\ \beta_1 = (x :: \beta).\ \mathsf{is\_list}\ i\ \alpha * \mathsf{is\_list}\ done\ \beta_1 \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ xs +\!\!+ \beta_1\}$
  // the loop invariant is reestablished
}
// upon exit, the loop invariant holds, but loop condition doesn't
$\{\exists \alpha\ \beta.\ \mathsf{is\_list}\ i\ \alpha * \mathsf{is\_list}\ done\ \beta \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ \alpha +\!\!+ \beta \wedge i = \mathsf{null}\}$
// from $i = \mathsf{null}$ it follows $\mathsf{is\_list}\ i\ \alpha \iff \mathsf{emp} \wedge \alpha = \mathsf{nil}$
$\{\exists \alpha\ \beta.\ \mathsf{emp} * \mathsf{is\_list}\ done\ \beta \wedge \mathsf{rev}\ \alpha_0 = \mathsf{rev}\ \alpha +\!\!+ \beta \wedge \alpha = \mathsf{nil}\}$
$\{\mathsf{is\_list}\ done\ (\mathsf{rev}\ \alpha_0)\}$

1.3 Separation Hoare Logic: properties and rules

The outline looks natural: it just involves tracking a few points-to predicates, and re-associating them as needed to derive the two instances of is_list, one for the sub-list at i, another for the sub-list at *done*. But there are some additional points that one should observe about this outline, so let us write them out.

First, notice that we could change this proof outline by attaching, by means of $*$, an arbitrary heap predicate R in every line. Because of commutativity and associativity of $*$, that predicate will just be commuted out. That is, once we established the above proof outline, we see intuitively that the following holds.

$$\{\mathsf{is\_list}\ i\ \alpha_0 * R\}$$
$$\texttt{reverse}$$
$$\{\mathsf{is\_list}\ done\ (\mathsf{rev}\ \alpha_0) * R\}$$

This is a property that can be proved in general, not just for `reverse` but for any program c, and not just for is_list, but any precondition P and postcondition Q. I.e., we have:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

(with some side-conditions on R, to be discussed soon), which is known as the *frame rule* in separation logic.

It is also important to observe that the frame rule holds because the precondition of every program *always* describes (a superset of) the pointers that the program touches (reads from, writes to, deletes, allocates). We will see promptly how this property is captured in the inference rules for reading, writing, deletion and allocation, but for now, we just notice that it is a key property that makes separation logic different from other styles of Hoare logic, and that it enables the frame rule.

For example, if reverse was to read or modify some additional pointer y, which is not encompassed by is_list i α_0 (i.e. y is not reachable from i), then the frame property wouldn't hold. For example, we can just pick for R the proposition

$$R = (y \mapsto 1).$$

If reverse modified y into 2, then clearly we wouldn't have $y \mapsto 1$ in the end (we will have $y \mapsto 2$), thus invalidating the frame rule.

On the flip side, if we frame the precondition by R, then we can't, in general, "erase" R from the postcondition. R will propagate through the derivation, and will end up appearing in the postcondition. It is not possible to "ignore" a piece of state in separation logic; every pointer has to be accounted for. Thus, if a program has a memory leak (i.e., it is not deallocating some state that it should have deallocated), that information will be visible in the specification. Thus, separation logic can be used to reason about lack of memory leaks.

The set of pointers that the program touches is known as the program's *footprint*. Also, it is frequently said that the pointers from the footprint are *owned* by the program. The ownership metaphor will become more apparent when we move to concurrency, where it will be the case that parallel threads t_1 and t_2 must have disjoint footprints; thus t_2 cannot read, modify, or deallocate the pointers that are *owned* by t_1.

Exercise 1.3.1. If we introduced a memory leak into reverse, e.g., by making it allocate a pointer in every iteration of the loop, without ever deallocating the pointer, what spec could we ascribe to reverse?

Again on the flip side, this means that Separation Logic, in its purest variant, is not all that well-suited for reasoning about garbage-collected languages. This point is often referred to as "Reynolds' conundrum", but we won't talk about it here. Papers by Calcagno, O'Hearn and Bornat [3] and Hur, Dreyer and Vafeiadis [14] are a good starting point for investigating this issue further.

Semantically, the frame rule is a consequence of definition of the Hoare triple

$$\{P\}\ c\ \{Q\}$$

which says that a program c *does not crash* in a heap satisfying P, and if it terminates, it returns a heap satisfying Q. The definition extends most previous formulations of Hoare logic, which only require that the program satisfies Q upon termination, but do not insist on not crashing. The not-crashing part is usually referred to as *fault avoidance*. In particular, it ensures that if we read from some pointer p, then p must be declared as allocated by P (and, in our case of typed heaps, p will also store a value of expected type). Similarly, if we're writing into p, or deallocating it, then p also must be declared as allocated by P. Thus, a footprint of the program must be described by the precondition, which in turn leads to the frame rule, as discussed above.

Note 1.3.1 (The first connection with type theory).

This fault avoidance property is one property in which separation logic is similar to type theory. As observed early on by Ishtiaq and O'Hearn [15]: a program which is *well-proved* in separation logic, doesn't go wrong. Obviously, this is a variation of the famous motto of Milner regarding type system. This property will motivate our formulation of separation logic as a dependent type system. As a historical remark, O'Hearn [25] attributes the idea of fault avoidance to Hoare and Wirth's axiomatic definition of Pascal [13].

Note 1.3.2 (On fault avoidance and framing).

One may think of satisfying fault avoidance by changing the program semantics to artificially never cause faults. For example, we can have the command $x := 3$ first check if x is allocated, and if so write 3 into it, and if not, behave like skip, i.e., return without doing anything. In this trivial way, one can make Milner's motto also hold of every programming language, by having the programs perform run-time type checks during execution. The whole point of the motto is that such checks do not need to be performed in a strongly-typed language, and similarly, no checks for allocation have to be performed in separation logic. More commentary on this issue, including the semantic property of programs called "framing property" and "safety monotonicity" which are prerequisite for the soundness of the frame rule, can be found in Reynolds' mini-course on separation logic (chapter 3, page 19).

We next present the inference rules of separation logic. We first present them in the classical form, as given, for example, by Ishtiaq and O'Hearn [15] and Reynolds [28], so that the reader can get acquainted with the standard. We then immediately revert to a presentation more in line with functional programming and type theory, which simplifies some issues, notably, it removes stack variables from the formalism. The functional presentation is not standard in the wider separation logic community, but it is widely adopted among researchers working on concurrency, and also higher-order variants of separation logic.

Separation logic, just like any other Hoare-style logic, makes a distinction between two categories of terms. First category is *commands* (ranged over by c), which are:

- variable assignment $x \leftarrow e$
- idle program skip
- pointer dereference (or lookup) $x \leftarrow !p$
- pointer mutation $p := e$
- allocation $x \leftarrow \mathsf{alloc}\ e$
- deallocation $\mathsf{dealloc}\ p$
- sequential composition $c_1; c_2$

Second category is *expressions*, which are purely-functional terms, such as integer expressions $n+1$, booleans $\mathsf{done}\,! = \mathsf{null}$, etc.

We use e and v to range over expressions (v to indicate that the expression can be evaluated to a value without any interaction with state, because it's purely-functional). We will often use distinct letters to differentiate between differently-typed expressions; e.g., p for pointers, n for nats, b for booleans, etc.

The Hoare triples provide a precondition and a postcondition for a *command*, but the commands may use expressions, e.g., $p := e$ contains two expressions p and e.

Let's start with the rule for sequential composition:

$$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1; c_2\ \{R\}}$$

This rule justifies the proof outline we wrote. You can see that the proof outline is, intuitively, a repeated application of this rule, where we essentially write the rule in a, somewhat abbreviated and vertical fashion, e.g.:

$$\begin{array}{c}\{P\}\\ c_1\\ \{Q\}\\ c_2\\ \{R\}\end{array}$$

Then we have a rule of consequence:

$$\frac{P \implies P' \quad \{P'\}\ c\ \{Q'\} \qquad Q' \implies Q}{\{P\}\ c\ \{Q\}}$$

We have a variable assignment rule, which is attributed to Tony Hoare.

$$\{[e/x]P\}\ x \leftarrow e\ \{P\}$$

And a rule for the idle computation

$$\{\mathsf{emp}\}\ \mathsf{skip}\ \{\mathsf{emp}\}$$

Then we have the rule for pointer update. Notice that the precondition insists that the pointer already exists, which, as mentioned before, is one of the essential properties to satisfy fault avoidance.

$$\{p \mapsto -\}\ p := v\ \{p \mapsto v\}$$

We haven't used allocation and deallocation in the reverse example but here are the rules:

$$\{p \mapsto -\}\ \mathsf{dealloc}\ p\ \{\mathsf{emp}\}$$
$$\{\mathsf{emp}\}\ p \leftarrow \mathsf{alloc}\ e\ \{p \mapsto e\}$$

In fact, when it comes to allocation, we will often need a command which allocates a block of several consecutive memory locations. Thus, we actually make alloc polymorphic in the number of arguments (i.e., we make it take a list), and the above version corresponds to passing a singleton list.

$$\{\mathsf{emp}\}\ p \leftarrow \mathsf{alloc}\ e_0\ \cdots\ e_n\ \{p \mapsto e_0 * \cdots * p + n \mapsto e_n\}$$

In the rule for allocation, we get the first complication regarding the treatment of variables. The rule has to insist that $p \notin \mathsf{FV}(e_i)$.

Why? Because if $p \in \mathsf{FV}(e)$, say, then the occurrence of e in the postcondition will accidentally use the new value of p, when the old one is expected.

Notice that this condition has to be given for frames as well. E.g., if we wanted to give a *large footprint* specification for allocation, it would look like:

$$\{R\}\ p \leftarrow \mathsf{alloc}\ e\ \{p \mapsto e * R\}$$

where $p \notin \mathsf{FV}(e, R)$.

Exercise 1.3.2. The above is a rule for so-called "non-overwriting" version of allocation. What should we do when e *does* contain p? As Reynolds explains in his mini-course on separation logic [29], Chapter 3, page 87, we require a new rule of the form:

$$\{p = v' \wedge \mathsf{emp}\}\ p \leftarrow \mathsf{alloc}\ e\ \{p \mapsto [v'/p]e\}$$

where v' is a variable distinct from p. Reynolds discusses other forms of the allocation rule, and when and how these forms can be interderived. In these lectures, we will work with the simple non-overwriting rule, as it will suffice once we remove the mutable variables from the formulation.

Similar side condition exists for the lookup rule:

$$\{e \mapsto v\}\ x \leftarrow !e\ \{x = v \wedge e \mapsto v\}$$

where $x \notin \mathsf{FV}(e)$.

Notice an interesting point here, which is that we introduced a variable v, which scopes over the precondition and postcondition, but not over the command. It is used to relate some aspect from the pre-state (in this case, the contents of the pointer p), to the post-state, in this case, the value of the variable x, and again, to say that the contents of p remained unchanged.

Such variables are called *logical variables* in Hoare logic.

Finally, for the frame rule, we have:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

where no variable from $\mathsf{FV}(R)$ is modified by c, which essentially collects the side conditions on variable occurrences that we would have required if we gave the large footprint rules for each command (Reynolds' notes [29], page 75).

These side-conditions on variable occurrences are fairly benign and easy to check in the setting considered so far. However, in the extensions of separation logic with more involved programming constructs, and especially in the presence of concurrency, they are much more unpleasant. Thus, we forego the presentation of the rest of the separation logic rules, and move on to a presentation in the style of functional programming, where the complication related to stack variables are avoided. We omit a number of separation logic structural rules (i.e., rules that change the specification, but have the same program c in the premiss and the conclusion), and refer to Reynolds' notes [29] for a full set. The rules will have their equivalents in the functional presentation.

1.4 Functional programming formulation

The elimination of stack variables is not too complicated. The rules don't change very much (at least on the surface), but they do lose most of the side conditions on the appearance of variables. Or rather, the side-conditions are modified, and now appear when a (functional) variable is introduced, rather than when that variable is used in an inference rule. As usual in functional languages, such side conditions can always be easily satisfied by α-renaming. The presentation is new, but derived from the previous work in [20, 22]. That work is formulated in a way suitable for use in an interactive theorem prover, but is not so convenient for paper-and-pencil presentations.

The idea on how to proceed with the reformulation comes from yet another connection between separation logic (or more generally, Hoare logic) and types. Hoare logic, in its separation between purely-functional expressions e, and side-effectful commands c, is very similar to another concept from the world of types: *monads*.

A monadic type $ST\ A$ stands for a (suspended) side-effectful computation, which, if executed, returns a value of type A, or diverges. Thus, monadic languages make a semantic distinction between expression and commands. The distinction can often be somewhat blurred, because commands, when suspended, are embedded back into the expressions, so there is not always an explicit syntactic distinction between the two categories. Nevertheless, there are formulations of monadic calculi where the semantic distinction between expressions and commands *is* reflected in the syntax. For example, one such is given by Pfenning and Davies [27].

Hence, we take the separation logic specification, in the form of Hoare triples, and make monads out of them. Instead of $\{P\}\ c\ \{Q\}$, we will write:

$$c : \{P\}\ \{Q\}$$

in cases when c is not supposed to return a value upon termination (equivalently, when it returns a value of unit type). More generally, if c returns a value of type A, we slightly generalize, and write:

$$c : \{P\}\ A\ \{Q\}$$

But now Q is a predicate not only over the ending heap as before in separation logic, but also scopes over the return value of type A.

For example, the inference rule for for return (aka., monadic unit) is as follows.

$$\frac{\vdash e : A}{\vdash \mathsf{return}\ e : \{\mathsf{emp}\}\ A\ \{\lambda r.\, \mathsf{emp} \wedge r = e\}}$$

We can see that the rule for return essentially binds a value to the return variable r. In that sense, it can be seen as subsuming the "variable assignment" rule of the classical system. Also, the special case of the rule, when $A = unit$, derives the inference rule for skip. Hence, we can say that monadic unit combines variable assignment and skip rules from the classical system.

At this point, we also introduce the following notational convention. To simplify matters, and make the notation look more like classical separation logic, we frequently drop "$\lambda r.$" from the postcondition Q in the Hoare triples, implicitly assuming that the return result is called r, and that r is not used in any other way in Q. When we want to use a different name for the return result, say x, we explicitly prefix the postcondition with "$\lambda x.$".

Thus, the rule for return looks like the following:

$$\frac{\vdash e : A}{\vdash \mathsf{return}\ e : \{\mathsf{emp}\}\ A\ \{\mathsf{emp} \wedge r = e\}}$$

It turns out that we actually need a bit more general syntax for the Hoare triple types. In order to deal with logical variables, we associate an extra context Γ, to the Hoare triple type, and write

$$c : [\Gamma].\ \{P\}\ A\ \{Q\}$$

where now P and Q (but not A) can contain variables from the context Γ. These will be the logical variables from the classical formulation of SL, but here we make it explicit that their scope extends through P and Q. The context Γ serves as a form of universal quantification, as we shall see from the forthcoming inference rules. If Γ is empty, we omit the prefix [ ].

Next we show the rule for sequential composition, i.e., monadic bind.

$$\frac{\vdash c_1 : [\Gamma].\ \{P\}\ A\ \{Q\} \qquad x : A \vdash c_2 : [\Gamma].\ \{Q\ x\}\ B\ \{R\} \quad x \notin \mathsf{FV}(B, R)}{\vdash x \leftarrow c_1; c_2 : [\Gamma].\ \{P\}\ B\ \{R\}}$$

The rule introduces the variable x into the scope of c_2, to name the return value of c_1. Since the postcondition Q of c_1 is parametrized over the return result of c_1, we use $Q\ x$ as a precondition for c_2. Alternatively, applying the convention on r that we introduced above, we can write the bind rule as:

$$\frac{\vdash c_1 : [\Gamma].\ \{P\}\ A\ \{Q\} \qquad x : A \vdash c_2 : [\Gamma].\ \{[x/r]Q\}\ B\ \{R\} \quad x \notin \mathsf{FV}(B, R)}{\vdash x \leftarrow c_1; c_2 : [\Gamma].\ \{P\}\ B\ \{R\}}$$

The use in proof outlines also needs to perform the renaming $[x/r]$. In particular, if we want to write this rule in a vertical format of the proof outline, we have:

$$\begin{array}{ll} [\Gamma]. & \{P\} \\ & x \leftarrow c_1; \\ [\Gamma]. & \{[x/r]Q\} \\ & c_2 \\ & //\ \text{ensure that } x \notin FV(B, R) \\ [\Gamma]. & \{R\} \end{array}$$

The rules for mutation, allocation and deallocation are straightforward, so we just show them expediently, using dependently typed notation. In particular, the type $\Pi x : A\ B(x)$ is the *dependent function type*; it classifies functions which take an argument x of type A, and produce a result of type $B(x)$. The conventional function type $A \to B$ is the special case, when the result type does not depend on x.

$$\begin{array}{rcl} := & : & \Pi x : \mathsf{ptr}.\, \Pi v : A.\, \{x \mapsto -\}\ \{x \mapsto v\} \\ \mathsf{alloc} & : & \Pi v : A.\, \{\mathsf{emp}\}\mathsf{ptr}\{r \mapsto v\} \\ \mathsf{dealloc} & : & \Pi x : \mathsf{ptr}.\, \{x \mapsto -\}\ \{\mathsf{emp}\} \end{array}$$

The type of dereference function is the first case where we use a non-trivial context of local variables. As in the classical formulation, we need a variable $v : A$ to scope over the precondition and postcondition in order to capture that the return result r equals what's written in the pointer x being read.

$$! : \Pi x : \mathsf{ptr}.\, [v : A].\, \{x \mapsto v\} A \{x \mapsto v \wedge r = v\}$$

We next present rules for manipulating the context Γ, which will illustrate that $[\Gamma]$ is as form of universal quantification scoping over P and Q. For example, we can substitute an arbitrary expression $e : A$ for a variable $v : A$ in Γ, as one would expect in a universal quantifier.

Logical variable elimination rule:

$$\frac{\vdash c : [\Gamma, x : A].\ \{P\ x\}\ B\ \{Q\ x\} \qquad \Gamma \vdash e : A}{\vdash c : [\Gamma].\ \{P\ e\}\ B\ \{Q\ e\}}$$

We also have the usual structural rules for manipulating contexts, e.g., context weakening.

$$\frac{\vdash c : [\Gamma].\, \{P\}\ B\ \{Q\}}{\vdash c : [\Gamma, x : A].\, \{P\}\ B\ \{Q\}}$$

The following rule for existential quantification over logical variables is very important, and says that a logical variable can be removed from the context, if existentially abstracted in the precondition.

Existential rule

$$\frac{\vdash c : [\Gamma, x : A].\, \{P\ x\} B \{Q\} \qquad x \notin \mathsf{FV}(B, Q)}{\vdash c : [\Gamma].\, \{\exists x : A.\, P\ x\} B \{Q\}}$$

If we invert this rule and write it in a vertical fashion, as in proof outlines, it looks as follows (where we make explicit the context of logical variables at each point).

$$\begin{array}{ll} [\Gamma]. & \{\exists x : A.\, P\ x\} \\ & // \text{ introduce } x \text{ into scope} \\ [\Gamma, x : A]. & \{P\ x\} \\ & c \\ & // \ x \notin \mathsf{FV}(B, Q) \\ [\Gamma]. & \{Q\} \end{array}$$

The rule of consequence now accounts for Γ.

$$\frac{c : [\Gamma].\, \{P'\} A \{Q'\} \qquad \forall \Gamma.\, P \implies P' \qquad \forall \Gamma.\, Q' \implies Q}{c : [\Gamma].\, \{P\} A \{Q\}}$$

We need three more rules.

The rule for conditionals:

$$\frac{\vdash b : \mathsf{bool} \qquad \vdash c_1 : [\Gamma].\, \{P \wedge b\} A \{Q\} \qquad \vdash c_2 : [\Gamma].\, \{P \wedge \neg b\} A \{Q\}}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : [\Gamma].\, \{P\} A \{Q\}}$$

The rule for fixed points, which has the usual typing of the fixed point combinator for functions with *one* argument $x : A$.

$$\mathsf{fix} : (\Pi x : A.\, [\Gamma].\, \{P\} B \{Q\} \to \Pi x : A.\, [\Gamma].\, \{P\} B \{Q\}) \to \Pi x : A.\, [\Gamma].\, \{P\} B \{Q\}$$

When we want functions with more than one argument, we implicitly curry them and take the type A to be a product. Notice that, as often the case in type systems, the type $\{P\}B\{Q\}$ of the recursive function cannot be inferred, and will have to be supplied by hand. In this sense, the type is analogous to a loop invariant, except it consists of two components P and Q. We will see in an example that the two-component type can somewhat simplify the way we write loop invariants (e.g., the loop invariant for reverse).

And finally, the rule of frame, where the frame R is now allowed to use variables from Γ.

$$\frac{\vdash c : [\Gamma].\, \{P\} A \{Q\}}{\vdash c : [\Gamma].\, \{P * R\} A \{Q * R\}}$$

Note 1.4.1 (Caveat).

The structural rules of Hoare logic are the rules where the same command c appears in the premiss and the conclusion of the rule. Such are the rules for logical variable elimination, the existential rule, the rule of consequence, and frame.

Formulating the rules like this is a bit unsatisfactory, because the typechecker loses information. For example, if the type for the program c can be derived by applying the rule of consequence as the last step, there is nothing in c that guides the type checker into taking that step. Moreover, even if the typechecker knew that it should apply the rule of consequence, it still has to figure out how to prove the implications $P \to P'$ and $Q' \to Q$. There is nothing in c that guides it in this process. In the terminology of Martin-Löf, the above system is not *analytic*, because the program c does not contain the necessary information to reconstruct the type derivation.

Thus, the formulation of the above Hoare-style type system that is implemented in Coq [21,22], *does not* apply the structural rules silently. It features an explicit program constructor, called do, which implements a *generalized* form of the rule of consequence, as follows.

When we want to change the type of the program $c : [\Gamma'].\{P'\}\{Q'\}$ into $[\Gamma].\{P\}\{Q\}$, we write do $c\ pf$, where pf is a proof of the conjunction of the following two properties, which generalize the customary rule of consequence to account for the two different logical variable contexts Γ and Γ'.

- $\forall \Gamma\ i.\, P\ i \implies \exists \Gamma'.\, P'\ i.$
- $\forall \Gamma\ i\ m.\, P\ i \implies (\forall \Gamma'\ i\ m.\, P'\ i \implies Q'\ m) \implies Q\ m.$

In the above, we make it explicit that P and Q are predicates over heaps, and explicitly quantify over such heaps: i for the pre-heap, and m for the post-heap.

It turns out that with such a coercion, all the other structural rules become derivable, and can thus be omitted from the calculus. We can also organize the calculus a bit differently, and give an inference rule for sequential composition which *infers* the precondition, the postcondition, and the context of logical variables of the sequential composition, out of those of the components. This requires using postconditions that are *binary*, i.e., scope over the initial and ending state of the program, but makes it unnecessary to apply the spec-changing coercion at every step of the sequential composition [22] thus reducing the proof sizes.

Furthermore, it is possible to formulate the calculus using two separate typing judgments: one for pure expressions, and another one for commands [20]. In that formulation, do is a coercion between the judgments, operationally suspending an effectful command and making it a pure expression, analogous to a similar coercion in the judgmental reconstruction of monadic calculi by Pfenning and Davies [27]. One can thus see that that separation logic is in a correspondence with a monadic calculus, in the style of Curry-Howard isomorphism: the rule of sequential composition corresponds to monadic bind, the rules of variable assignment and skip, taken together, correspond to monadic unit, and the structural rules all correspond to monadic do.

Exercise 1.4.1. We will frequently use the inference rules in situations when the precondition not only contains extra heap, but also extra conditions. For example, we may know that $x \mapsto v \wedge v > 3$, and we want to infer that the command $t \leftarrow !x$ returns a value that is bigger than 3. E.g.:

$$\begin{array}{ll} [v]. & \{x \mapsto v \wedge v > 3\} \\ & t \leftarrow !x; \\ & \mathsf{return}\ t \\ [v]. & \{x \mapsto v \wedge v > r \wedge r = v\} \end{array}$$

But we cannot directly apply the rule for sequential composition because the type of $!x$ is:

$$!x : [v : \mathsf{nat}].\{x \mapsto v\}\ \mathsf{nat}\ \{x \mapsto v \wedge r = v\}$$

I.e., its precondition is $x \mapsto v$, but we need $x \mapsto v \wedge v > 3$.

To enable this step, we frame the type of $!x$ with $R = (\mathsf{emp} \wedge v > 3)$. E.g., we turn $!x$ into

$$!x : [v : \mathsf{nat}].\{x \mapsto v * (\mathsf{emp} \wedge v > 3)\}\ \mathsf{nat}\ \{(x \mapsto v \wedge r = v) * (\mathsf{emp}\ \ \wedge v > 3)\}$$

We also need to weaken the logical variable context of return t with v, and then frame the rule with $R = (x \mapsto v \wedge v > 3 \wedge t = v)$:

$$\begin{array}{l} \mathsf{return}\ t : [v].\{\mathsf{emp} * (x \mapsto v \wedge v > 3 \wedge t = v)\}\ \mathsf{nat} \\ \qquad\qquad \{(\mathsf{emp} \wedge r = t) * (x \mapsto v \wedge v > 3 \wedge t = v)\} \end{array}$$

With these two typings, we can write a valid proof outline as follows.

$[v]$. $\{x \mapsto v \wedge v > 3\}$
// by rule of consequence
$[v]$. $\{x \mapsto v * (\mathsf{emp} \wedge v > 3)\}$
$t \leftarrow !x$;
// notice renaming $[t/r]$
$[v]$. $\{(x \mapsto v \wedge t = v) * (\mathsf{emp} \wedge v > 3)\}$
// by rule of consequence
$[v]$. $\{\mathsf{emp} * (x \mapsto v \wedge v > 3 \wedge t = v)\}$
return t
$[v]$. $\{(\mathsf{emp} \wedge r = t) * (x \mapsto v \wedge v > 3 \wedge t = v)\}$
// by rule of consequence, to remove free occurrence of t
$[v]$. $\{x \mapsto v \wedge v > 3 \wedge r = v\}$

In general, the strategy of writing the proof outlines will be as shown above. We use the context of the logical variables from the target specification. We treat it as part of our context. Then for each command that we encounter in the sequential composition, we change its type, to match the context of logical variables, and to match the precondition with the assertion that we have, which can be done by the structural rules.

Exercise 1.4.2. Consider a stack data structure, laid out in the heap as a singly-linked list, with a dedicated sentinel pointer (call it snt), pointing to the top of the stack. We implement, specify, and verify the push method of the stack, leaving pop as an exercise for the reader. First, we need a separation logic predicate to describe the shape of the stack, just like we had one for presenting the list in the case of reverse. Similarly to reverse, we parametrize the predicate over a purely-functional list $\alpha : \mathsf{list}\ A$, which describes the contents of the stack. In fact, when defining this predicate, we can reuse our previous definition of is_list.

$$\mathsf{stack}(\alpha) = \exists p.\, snt \mapsto p * \mathsf{is\_list}\ p\ \alpha$$

Now the specification and the implementation we want to give to $\mathsf{push}(v)$ is as follows. It merely says that the contents α is increased by putting v on top.

$$\begin{array}{rcl} \mathsf{push}(v : A) & : & [\alpha].\{\mathsf{stack}(\alpha)\}\ \{\mathsf{stack}(v :: \alpha)\} \\ & = & t \leftarrow !snt; \\ & & q \leftarrow \mathsf{alloc}\ v\ t; \\ & & snt := q \end{array}$$

The proof outline is below, where most of the time we silently apply the structural rules as needed.

$[\alpha]$. $\{\mathsf{stack}(\alpha)\}$
$[\alpha]$. $\{\exists p.\, snt \mapsto p * \mathsf{is\_list}\ p\ \alpha\}$
// rule of existentials, put p into scope
$[\alpha, p]$. $\{snt \mapsto p * \mathsf{is\_list}\ p\ \alpha\}$
$t \leftarrow !snt$;
// t shouldn't appear in the ending postcondition
$[\alpha, p]$. $\{snt \mapsto t * \mathsf{is\_list}\ t\ \alpha\}$
$q \leftarrow \mathsf{alloc}\ v\ t$;
// q shouldn't appear in the ending postcondition
$[\alpha, p]$. $\{snt \mapsto t * \mathsf{is\_list}\ t\ \alpha * q \mapsto v; t\}$
$snt := q$;
$[\alpha, p]$. $\{snt \mapsto q * \mathsf{is\_list}\ t\ \alpha * q \mapsto v; t\}$
$[\alpha, p]$. $\{snt \mapsto q * \exists t.\, q \mapsto v; t * \mathsf{is\_list}\ t\ \alpha\}$
$[\alpha, p]$. $\{snt \mapsto q * \mathsf{is\_list}\ q\ (v :: \alpha)\}$
$[\alpha, p]$. $\{\mathsf{stack}(v :: \alpha)\}$
// remove p from scope
$[\alpha]$. $\{\mathsf{stack}(v :: \alpha)\}$

Exercise 1.4.3.

As a more involved example of type-based notation, consider the functional implementation of reverse, as follows.

```
reverse (i : ptr) : [α0]. {is_list i α0} ptr {λdone. is_list done (rev α0)} :=
  let recfun f (i done : ptr) : revT =
    if i = null then return done
    else k ← !(i + 1);
         i + 1 := done;
         f k i)
  in
    f i null
  end
```

The head pointer i is now an input to reverse, but *done* is not an input – rather, *done* is the return result of reverse (we explicitly name it, instead of calling it r). The loop is encoded as a recursive function over two pointers i and *done*, which is called with the initial value null for *done*.

The type revT depends on f's arguments i and *done*, and is defined as:

$$\mathsf{revT} = [\alpha, \beta].\, \{\mathsf{is\_list}\ i\ \alpha * \mathsf{is\_list}\ done\ \beta\}\ \mathsf{ptr}\ \{\lambda r.\, \mathsf{is\_list}\ r\ (\mathsf{rev}\ \alpha\ +\!+\ \beta)\}$$

and it serves as the loop invariant. However, unlike the loop invariant in the classic formulation, we don't use α_0 in revT to name the initial list (in fact, α_o is not even in scope when we define revT). The fact that we're reversing α and pre-pending it to β is expressed directly via logical variables, rather than by equations to α_0.

As an exercise, adapt the proof outline of reverse from the classical setting to the functional one.

1.5 Pointed assertions and Partial Commutative Monoids (PCMs)

We close this section with one more change to the classical separation logic, that will serve us well once we move into concurrency.

In the previous sections, we already illustrated the implicit nature of heaps in separation logic. Indeed, we write emp to say that the current heap is empty. We write $P * Q$ to split the current heap into two. But, we never write out the heap explicitly as a value. A technical way to describe this property, is that separation logic assertions are *point free* (i.e., they don't mention the heap in point).

The point-free approach is most of the times very effective *in the sequential case*, but it does not scale all that well. Its effectiveness arises from the fact that in the sequential case, our state consists of only one component, namely, the heap. When we have only one such "point" of concern, we might as well keep it implicit, as it is always clear which heap the assertions are refering to. However, in the forthcoming extensions, we will have notions of state with many more components (i.e., "points"), which evolve in interdependent ways. In such a setting, it becomes very unwieldy if the components cannot be referred to by explicit names. Thus, multiple components immediately force a *point-full* style of presentation.

Formulating separation logic in a point-full style is not very common in the literature, but it is not very difficult, so let us quickly describe it. We work with a model of heaps instead of describing heaps in the roundabout way via predicates.

In particular, we build heap values out of the four primitives:

- empty heap, a finite map with no elements in the domain
- $x \Mapsto v$ – a singleton heap, and
- a distinguished value called undef standing for "undefined" heap
- $h_1 \bullet h_2$ – a disjoint union of two heaps, undefined if h_1 and h_2 have a pointer in common

Obviously, the first three constructors directly correspond to the propositional constructors we already had, e.g., emp, $x \mapsto v$, and $*$. In the point-full setting, the propositional versions just become the syntactic sugar as follows.

We first introduce a common label σ to refer to the current heap. We also introduce a predicate valid which tests if a heap is defined or not. Then we can define the classical propositions as:

$$\begin{array}{rcl} \mathsf{emp} & \widehat{=} & \sigma = \mathsf{empty} \\ x \mapsto v & \widehat{=} & \sigma = x \Mapsto v \\ P * Q & \widehat{=} & \exists h_1\ h_2.\, \sigma = h_1 \bullet h_2 \wedge \mathsf{valid}\ h \wedge [h_1/\sigma]P \wedge [h_2/\sigma]Q \end{array}$$

We can also introduce similar abbreviations as with the point-free notation, and write:

$$\sigma = x \Mapsto (v_0; v_1; \ldots; v_n) \ \widehat{=}\ \sigma = (x \Mapsto v_0) \cup (x+1 \Mapsto v_1) \cup \cdots \cup (x+n \Mapsto v_n)$$

All this notation will give us a unifying mechanism for extending our reasoning, as the notation applies to more than just heaps. It applies to any algebraic structure satisfying the properties of an abstract algebraic structure of *Partial Commutative Monoid* (PCM), described below, of which heaps are just an instance. As has been recognized by Calcagno, O'Hearn and Yang in abstract separation logic [4], many important separation logic rules, such as the rule of frame, are not specific to heap, and only require that the underlying notion of state satisfies the PCM properties. Thus, by moving to PCMs, we will gain a lot of mileage in the case of concurrency, where we will have to use non-heap state in an essential way.

What is a PCM? A PCM is an algebraic structure $(U, \bullet, \mathbb{1}, \mathsf{valid})$, which, in these notes, we take to have the following properties.

- U is a carrier set.
- $\bullet$ (called "join") is a binary operation. It is partial, that is, it returns undefined elements on some subset of its domain. It is commutative and associative.
- valid determines if an element is undefined (we can have more than one undefined element).
- $\mathbb{1}$ is a distinguished element, which is the unit element for $\bullet$.
- if $\mathsf{valid}(x \bullet y)$, then $\mathsf{valid}\ x$ and $\mathsf{valid}\ y$. In other words, joining undefined elements can't produce a defined one.
- $\mathbb{1}$ is a valid element.

Other works define PCMs differently. Most do not consider valid, and instead treat undefinedness implicitly. Also, often $\bullet$ is taken to be cancellative, in which case the PCM is called a *separation algebra*, which we do not require here.

Chapter 2

Concurrency and the Resource Invariant Method

2.1 Parallel composition and Owicki-Gries-O'Hearn resource invariants

Concurrency poses several new challenges that will require extending our reasoning system with several new components. To illustrate the challenges and the principles that we want to follow when addressing them, we start with a classic example, due to Owicki and Gries [26].

We want to verify the following program, which has two threads that concurrently increment the shared pointer x. We write out the program in a monadic fashion, to be consistent with the notation from Chapter 1.

$$\begin{array}{lll} \mathsf{incr}_2 = & \mathsf{lock}\ l; & \mathsf{lock}\ l; \\ & t \leftarrow !x; & t \leftarrow !x; \\ & x := t + 1; & x := t + 1; \\ & \mathsf{unlock}\ l; & \mathsf{unlock}\ l; \end{array}$$

The lock l protects the variable x, so that only one of the threads can operate on x at any given time. In this particular case, when the *left* thread is reading from x, or writing $t + 1$ into x, or is in-between these operations, the *right* thread is not able to access x, and vice versa. The property we want to prove in our verification is that if we start with $x = 0$, then in the end $x = 2$ (or more generally, if we start with $x = n$, then in the end, $x = n + 2$).

Note 2.1.1 (Some terminology, notation, and related work context).

- The property that only one thread can access a shared state at any given time is called *mutual exclusion*.
- When a thread holds l, it is common to also say that the thread is in a *critical section*.
- Alternatively, it is also common to say that the code in the critical section is executed *atomically*. This does not mean that all the operations execute at one moment in time (*physical atomicity*). Rather, no other threads can interfere with the execution, hence, no programs that invoke these operations, can distinguish their execution times (*logical atomicity*).
- We will use all these expressions (locking, critical sections, atomic execution), interchangeably.
- Some logics, including the original one of Owicki and Gries, streamline the presentation, and have a command such as

  $$\langle c \rangle$$

  which, in our terminology, stands for $\mathsf{lock}\ l; c; \mathsf{unlock}\ l$, where l is a global lock. In other words, locking and unlocking are done in a scoped (i.e., LIFO) manner.
- Some logics, including Concurrent Separation Logic (CSL) of O'Hearn [24], allow more than one lock. Hence locking can be done in a more controlled fashion. CSL also insist that locking and unlocking

are done in a scoped manner, but the notation is more involved than $\langle - \rangle$. In particular, in CSL one can write:

$$\mathsf{with}\ l\ \mathsf{when}\ b\ \mathsf{do}\ c$$

This command identifies the lock that we care about (here l), and then, if the condition b is true, locks l, executes c, and then unlocks l. The condition b often expresses a property of the state protected by l. For example, if that state is a buffer, b may require that the buffer is full, or empty, etc., at the moment the lock is acquired. This is operationally typically achieved by by holding a lock for the duration testing b.

- We use explicit lock and unlock commands, instead of $\langle - \rangle$, as this will set us up for the next sections on *fine-grained* concurrent programs, where one can *implicitly* lock only parts of a data structure. In contrast, concurrent programs with explicit locking of the shared structure in full, are called *coarse-grained*. Nevertheless, if we are working with an example in which the locking is well-bracketed, and we are working with only one lock, or the lock is otherwise clear from the context, we use $\langle c \rangle$ to abbreviate the construction above.
- Many logics, including the variant of CSL from [24], assume that the program is a large top-level parallel composition of the form

  $$c_1 \parallel \cdots \parallel c_n$$

  where neither c_i itself contains parallel composition, i.e., each c_i is *sequential*. In the common terminology, a *thread* is one such sequential component. In these lectures, we will assume that nesting of parallel compositions *is* allowed, and thus, for us, the word *thread* will refer to any program that appears as an operand of $\parallel$, irrespective of that program itself being sequential or not.

One way to verify a concurrent program is to enumerate all the interleavings of its component threads. This is not particularly difficult to do in the case of incr_2, as there are only two possibilities: either the left thread acquires the lock first and proceeds to execute, followed by the right thread, or the right thread locks first. But in principle, enumerating all the interleavings does not scale. If we had even slightly longer programs, each with several critical sections in it, then the number of interleavings grows exponentially in the number of critical sections.

Instead, we want a method that arranges the reasoning so that we can verify the left thread separately from the right one, and then just put the two verifications together into a verification of the parallel composition. The conclusion about the combination should be derived merely from the *specifications* of the two sub-threads, without going through their individual code, or their individual proofs. This property, unsurprisingly, goes under the name of *compositional reasoning*.

How should we do that?

As a first step towards the answer, let us consider a very special case. Had the two threads of incr_2 worked with *disjoint* state, instead of both invoking the shared variable x, then it would have been easy to decompose the reasoning. For example, if the left thread operated on the pointer x_1, and the right thread operated on x_2, then it doesn't matter how the two threads interleave, as all interleavings produce the same result. We don't even have to use any locking.

For example, if we had already established the following two Hoare triples:

$$t \leftarrow !x_1; x_1 := t + 1 : \{x_1 \mapsto 0\}\{x_1 \mapsto 1\}$$

and

$$t \leftarrow !x_2; x_2 := t + 1 : \{x_2 \mapsto 0\}\{x_2 \mapsto 1\}$$

Then, clearly, for the parallel composition we should be able to infer:

$$\{x_1 \mapsto 0 * x_2 \mapsto 0\}\{x_1 \mapsto 1 * x_2 \mapsto 1\}$$

To perform this kind of inferences, when the state manipulated by the concurrent threads is disjoint, O'Hearn in CSL [24] introduced the following rule.

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\{Q_2\}}{\{P_1 * P_2\}\ c_1 \parallel c_2\{Q_1 * Q_2\}}$$

with condition that no variable in $\mathsf{FV}(P_i, Q_i)$ is modified by c_j for $j \neq i$.

As usual, when we switch to the monadic style, we have to take care of the return results, so for us, the rule looks like following (we omit the context Γ for simplicity, as they don't change across the rule).

$$\frac{c_1 : \{P_1\}A_1\{Q_1\} \qquad c_2 : \{P_2\}A_2\{Q_2\}}{c_1 \parallel c_2 : \{P_1 * P_2\}\ A_1 \times A_2\{[\mathsf{fst}\ r/r]Q_1 * [\mathsf{snd}\ r/r]Q_2\}}$$

This is a very important rule, and we will come back to it, to modify it in important ways. For now, just notice that the rule essentially generalizes the frame rule from the sequential case. More specifically, the frame rule is an instance of this rule, where c_2 is chosen to be the idle thread (i.e., skip, or, in the monadic formulation, $\mathsf{return}\ ()$).

OK, but the above alone doesn't suffice for our example, since we have concurrent threads accessing a shared pointer. Here's where the idea of *resource invariants* comes in. This is the first of the two related ideas of Owicki and Gries that we will consider.

> We name by *resource*, the shared state that threads manipulate; that is, the state protected by the lock. In our case, the lock l is part of the shared state, since all threads can access it, but for us in this section, the resource includes only the pointer x that the lock protects (we will generalize this slightly in the next chapter, where we will also admit l as part of the resource). With each resource we associate a predicate I, called *resource invariant*, which describes what holds of the resource state when it is *not accessed* by any thread, i.e., when the corresponding lock is free. In our case, I will be a predicate over the current heap (which we leave implicit when working in the point-free style, and explicitly name σ, when working in the point-full style). Thus, I *abstracts* the behavior of threads on the resource.
>
> In the proofs, we expect the threads to follow the following protocol. As I describes the resource when it is unlocked, each thread, upon locking, may assume that I holds of the resource at the beginning of the locking thread's critical section. While the thread holds the lock, i.e., for the duration of the critical section, the thread can make inferences out of the property I. It can also modify the state to violate I. However, before releasing the lock, the thread has to restore I, so that I will hold, as expected, once the lock is released.

Owicki and Gries formulated the idea of resources in the absence of dynamic memory with pointers. To cover those, we need one more idea, this one due to O'Hearn and CSL [24].

> Upon locking a resource with a resource invariant I, the locking thread *acquires ownership* of the heap circumscribed by I. This heap becomes part of the footprint of the acquiring thread. Upon unlocking, the heap circumscribed by I, which may now be different from the one upon entrance, is removed from the footprint. We say that the ownership of the resource's heap is exchanged between the program and the resource upon locking/unlocking.

These two ideas together will give us a way to decompose the verification of threads that operate over shared resources, as follows. As a matter of principle, we will think of concurrent threads as performing independent computations *on disjoint state*. Inside each thread, we can have locking and unlocking of a shared resource, but since the threads respect the Owicki-Gries-O'Hearn discipline on resource invariant, *the fact that they manipulate shared state can be ignored*, as long as the inferences about the threads' behaviors are made out of knowing that I (but nothing stronger than I) holds upon resource acquisition. Then the threads can be verified separately and their proofs combined using the rule of parallel composition, as if they operated on disjoint state.

These ideas are codified in the logic as follows. First, since we have resource invariants around, we have to make them part of the specification. Thus, we generalize our specifications a bit, and make them contain a context of resource invariants, in addition to preconditions and postconditions.

$$\{P\}A\{Q\}@(l_1 \mapsto I_1, \ldots, l_n \mapsto I_n)$$

There is an important requirement that the predicates I_k are *precise*. We will define this notion subsequently and have a brief discussion about it. The specification is ascribed to programs that, if executed

in a state satisfying $I_1 * \cdots * I_n * P$ either diverge, or terminate in a state satisfying $I_1 * \cdots * I_n * Q$, and throughout the execution, respect the resource invariant protocol on each I_i, as outlined above.

As a side observation, notice that in any useful spec, $I_1, \cdots, I_n$ will circumscribe disjoint pieces of state. If the invariants I_k overlap, such a spec will trivially hold of any program, in the same way that a spec with a false precondition can be ascribed to any program. But, of course, we won't be able to find a heap in which to safely run such a program.

Second, we need inference rules for locking and unlocking.

$$\frac{}{\mathsf{lock}\ l : \{\mathsf{emp}\}\ \{I\}@(\Gamma, l \mapsto I)}$$

$$\frac{}{\mathsf{unlock}\ l : \{I\}\ \{\mathsf{emp}\}@(\Gamma, l \mapsto I)}$$

As can be seen, locking l acquires ownership of the state described by I, and dually for unlock. These precisely capture the ideas of Owicki and Gries about the protocol on resource invariants, and of O'Hearn about resource ownership.

Note 2.1.2 (On Resource Context).

We note that, at this stage, the addition of the context of resource invariants is a bit of a moot point. In the current chapter, we will only use one resource (i.e., one global lock l), so we can just as well ignore the context of resource invariants. In the next chapter, we will generalize the idea of resources significantly, anyway, so that locks will be treated as ordinary pointers, and will not have any specific logical infrastructure attached to them.

Nevertheless, we bring up the resource contexts now, for completeness. In particular, we note that in a variant of CSL due to Brookes [2], it is possible to dynamically allocate resources using a rule that looks *roughly like the following*:

$$\frac{c : \{P\}\ A\ \{Q\}@(\Gamma, l \mapsto I) \quad l \text{ is fresh} \quad I \text{ is precise}}{\mathsf{resource}\ l\ \mathsf{in}\ c : \{P * I\}A\{Q * I\}@\Gamma}$$

Looking at the conclusion, if we can identify a state satisfying I in our initial state, then we can promote I into a resource under a fresh lock name l. If we execute c in such a context, upon termination, we can collect the state back. Because c follows the Owicki-Gries-O'Hearn protocol, we know that I holds of it in the end.

Note 2.1.3 (On precision).

As one can see from the inference rules, as threads lock and unlock, they acquire and release state of the shared resource. It will thus be important to determine how much of the state should be transferred upon locking and unlocking.

In the incr_2 example, the shared state consists just of the integer pointer x. So, we can choose a resource invariant $I = \exists v{:}\mathsf{nat}.\, x \mapsto v$, which describes this heap precisely.

With this resource invariant, upon acquisition, a thread will get x into the ownership, and upon release, x goes away. In both cases, the chunk of transferred state is *uniquely determined*. In general, this requirement of the state being uniquely determined is codified by the following definition.

Definition 2.1.1 (Precision)**.** *A predicate P over heaps (but we can define this for any other PCM U) is* precise*, if for every heap, a subheap for which P holds, if any exists, is uniquely determined.*

$$\forall h_1\ h_2\ k_1\ k_2 : \mathsf{heap}.\, \mathsf{valid}(h_1 \bullet k_1) \implies P\ h_1 \implies P\ h_2 \implies h_1 \bullet k_1 = h_2 \bullet k_2 \implies h_1 = h_2.$$

According to this definition, I above is precise, because for every heap, there is at most one subheap containing exactly the pointer x, and nothing else. An example of an imprecise predicate is, for example, $\top$. If we had a resource invariant $I = \top$, then it would not be possible to determine uniquely upon locking which

heap we obtain, and upon unlocking, which heap we give away. Another example of a precise predicate is $\mathsf{is\_list}\ p\ \alpha$, for any given α, or even $\exists\alpha.\,\mathsf{is\_list}\ p\ \alpha$. Indeed, in both cases, the heap the predicate describes is uniquely determined: it contains exactly the state that is reachable from p.

Precision may be seen as yet another connection with type theory, which is why we emphasize it here. When a resource with a precise resource invariant is locked or unlocked, the next symbolic state in the evaluation of the program is uniquely determined, because the amount of heap to be added to the current state upon locking, or subtracted from the current state upon unlocking, is uniquely determined. Thus, when verifying a program that starts with locking or unlocking, the typechecker can proceed to verify the continuation without any guesswork as to how the heap changes. This is similar to type theory, where the proofs are always designed to avoid guesswork too: a proof of disjunction explicitly states which side of the disjunction holds, a proof of an existential explicitly provides the witness. Requiring precision of resource invariants in CSL is a design of exactly the same nature.

Let us now illustrate the rules on an example *related* to incr_2, and observe their shortcomings. It will turn out that we can use the Owicki-Gries-O'Hearn method of resource invariants to prove some non-trivial properties, though not necessarily *yet* the one property that we set out to prove: namely that incr_2 increments x by 2. For that property, we will need an interesting generalization. But, for now, we can prove the following program and spec, which we consider below using the pointed notation for the specs:

$$\mathsf{lock}\ l; t \leftarrow\ !x; x := t+2; \mathsf{unlock}\ l; \mathsf{return}\ (t+1) : \{\sigma = \mathsf{empty}\}\ \mathsf{nat}\ \{\sigma = \mathsf{empty} \wedge \mathsf{odd}\ r\}@(l \mapsto I)$$

under the resource invariant $I = (\exists n : \mathsf{nat}.\,\sigma = x \Mapsto n \wedge \mathsf{even}\ n)$. We also start omitting the contexts of logical variables from proof outlines, for simplicity.

$\{\sigma = \mathsf{empty}\}$
$\mathsf{lock}\ l;$
$\{\exists n.\,\sigma = x \Mapsto n \wedge \mathsf{even}\ n\}$
// enter n into scope; watch that n is removed by the end
$\{\sigma = x \Mapsto n \wedge \mathsf{even}\ n\}$
$t \leftarrow !x;$
// by framing lookup, as we've seen; also rename $[t/r]$ in continuation
$\{\sigma = x \Mapsto t \wedge \mathsf{even}\ t\}$
$x := t + 2;$
$\{\sigma = x \Mapsto t+2 \wedge \mathsf{even}\ t\}$
// by rule of consequence
$\{(\exists n (= t+2).\,\sigma = x \Mapsto n \wedge \mathsf{even}\ n) * (\sigma = \mathsf{empty} \wedge \mathsf{even}\ t)\}$
$\mathsf{unlock}\ l;$
// by framing the unlock rule
$\{\sigma = \mathsf{empty} \wedge \mathsf{even}\ t\}$
$\mathsf{return}\ (t+1)$
$\{\sigma = \mathsf{empty} \wedge \mathsf{even}\ t \wedge r = t+1\}$
// eliminate t by rule of consequence to satisfy side condition
$\{\sigma = \mathsf{empty} \wedge \mathsf{odd}\ r\}$

The proof outline illustrates our intuition, namely, that upon acquiring the lock a piece of heap satisfying I materializes into the ownership of the verified thread, and dually, it is lost upon exiting. However, between acquisition and release, we can make inferences out of I that survive the critical section; e.g., in this case, we inferred that t must be even, which we ultimately used to show that the return result of the whole program is odd.

2.2 Auxiliary state

In the above example, we could prove that the result r is odd, but, it turns out, we cannot *directly* prove something more precise, e.g., that the program increments x by 2. The same exact problem appears for

the original incr_2 example. While one thread may increment x by a given amount, x may be independently incremented by many other threads running in parallel. Thus, the amount by which x is incremented is a property of each individual thread; it is not an invariant property of the resource, and hence cannot be captured using *only* a resource invariant.

We need something more, which brings us to the second idea of Owicki and Gries. They proposed introducing additional mutable variables which keep track of how each thread modifies the shared resource. In the case of incr_2, we will have two variables, a and b, that capture how much the left and the right thread, respectively, have added to x. For obvious reasons, these are called *auxiliary variables*, or more generally, *auxiliary state*, as they are introduced solely for the purpose of the proof, rather than for actual execution (also used names are *ghost variables* and *ghost state*). We assume that the auxiliary variables are in the scope of the resource invariant, and the two threads.

To illustrate the idea, let us revert back to using stack variables for a second, as in the classical formulation of separation logic. We will see that then reformulating the idea back into the functional formulation will provide some new insights. In order to deal with incr_2, we choose the resource invariant:

$$I = (\sigma \;=\; x \Mapsto a + b)$$

We also change the program as follows. We annotate each thread with the code, called *auxiliary code*, that mutates the auxiliary variables. The annotated version of incr_2 is presented below, and we immediately write out how *we would like the proof outline to look like*. Clearly, this is not a real proof outline yet, since we do not have rules for dealing with auxiliary variables, but it will serve as an inspiration for modifying our inference rules once more.

$\{a = 0 \wedge b = 0 \wedge \sigma = \mathsf{empty}\}$
// thus, by resource invariant, $x \mapsto 0$

// value of b unknown on left	// value of a unknown on right
$\{a = 0 \wedge \exists n.\, b = n \wedge \sigma = \mathsf{empty}\}$	$\{\exists m.\, a = m \wedge b = 0 \wedge \sigma = \mathsf{empty}\}$
lock l;	lock l;
$\{a = 0 \wedge \exists n.\, b = n \wedge \sigma = x \Mapsto a + b\ (= n)\}$	$\{\exists m.\, a = m \wedge b = 0 \wedge \sigma = x \Mapsto a + b\ (= m)\}$
// introduce n into scope	// introduce m into scope
$\{a = 0 \wedge b = n \wedge \sigma = x \Mapsto n\}$	$\{a = m \wedge b = 0 \wedge \sigma = x \Mapsto m\}$
$t \leftarrow\, !x$;	$t \leftarrow\, !x$;
$\{a = 0 \wedge b = n \wedge \sigma = x \Mapsto n \wedge t = n\}$	$\{a = m \wedge b = 0 \wedge \sigma = x \Mapsto m \wedge t = m\}$
$x := t + 1$;	$x := t + 1$;
$\{a = 0 \wedge b = n \wedge \sigma = x \Mapsto n + 1\}$	$\{a = m \wedge b = 0 \wedge \sigma = x \Mapsto m + 1\}$
$a \leftarrow a + 1$;	$b \leftarrow b + 1$;
$\{a = 1 \wedge b = n \wedge \sigma = x \Mapsto a + b\}$	$\{a = m \wedge b = 1 \wedge \sigma = x \Mapsto a + b\}$
// remove n	// remove m
$\{a = 1 \wedge \sigma = x \Mapsto a + b\}$	$\{b = 1 \wedge \sigma = x \Mapsto a + b\}$
unlock l;	unlock l;
$\{a = 1 \wedge \sigma = \mathsf{empty}\}$	$\{b = 1 \wedge \sigma = \mathsf{empty}\}$

$\{a = 1 \wedge b = 1 \wedge \sigma = \mathsf{empty}\}$
// thus, by resource invariant, $x \mapsto 2$

We want to engineer a proof system that can enable this kind of reasoning. What should we pay attention to?

- It would be nice to *not* use stack variables. This is not a goal in itself, of course: if the stack variables worked well, we should continue using them. The problem is that they don't work all that well. In particular, their use makes the above proof specific to the number of threads that appear in the program.
  For example, if we wanted to add a third thread, we would need an additional variable c, and a new resource invariant $I \;\widehat{=}\; (\sigma = x \Mapsto a + b + c)$. Thus, if we did the proof once for two threads, and then decide to generalize to three, i.e., write the program

$$\mathsf{incr}_3 = \mathsf{incr}_2 \parallel \mathsf{incr}_1$$

the sub-proof for incr_2 that we may have already finished, will be formally *invalidated*. It will have to be redone from scratch, using the new resource invariant. The situation is even worse if we don't know statically how many threads we will have at run time (e.g., if we fork threads at run-time while iterating over a list, where each thread is supposed to process one element of a list).
We seek a solution that will enable the proof of incr_2 to be reusable. But then, what should a and b be? Pointers in the heap?

- The variables a and b are used in both sides of the proof. Thus, if we want to turn them into pointers, then neither of these pointers can be owned *exclusively* by only one of the threads. Clearly, this conflicts with the rule for parallel composition, which wants concurrent threads to work over disjoint state. We may decide to make the pointers a and b be part of the shared resource. In that case, however, we will have to relax the current setup where the resource state can only be looked up within critical sections. Indeed, the above proof reads a and b outside critical sections, e.g., in the precondition and postcondition of the whole program.
- It is important to guarantee that in the critical section on the left, the variable (or pointer, if we somehow manage to make it a pointer) b remains unchanged (i.e., equal to n). Dually, the variable a remains unchanged (equal to m) in the critical section on the right. Otherwise, in the last line in the proof of the critical section, just before unlocking, *we won't be able to prove that we have restored the resource invariant*. Note that this is not merely a requirement that the left thread doesn't change b; it's that the right thread doesn't change b when the left thread owns the lock. In other words, the manipulation of a and b has to somehow encode the mutual exclusion property.
- The proofs are symmetric, modulo the names of auxiliary variables used.

2.3 Subjective auxiliary state

We consider now how to modify a and b, so that they are not stack variables. What should they be? Pointers in the heap, or something completely different?

Note 2.3.1 (Possible approaches that we won't follow).
There have been several different ideas on how to redesign the Owicki-Gries-O'Hearn method to reconcile it with pointer-based auxiliary state. Some of them have been designed with motivation that is tangential to auxiliary state, but can nevertheless be applied to the question.

The first idea is to have a and b be ordinary pointers in the heap, but, as we outlined before, they cannot be owned exclusively by either thread, nor by the resource. One way we can express how the ownership of a and b should work, is due to Bornat et al. [1], and goes by the name of *fractional permissions*.

In separation logic, a predicate such as $a \mapsto m$ (alternatively, $\sigma = a \Rightarrow m$), acts as a permission that lets a thread that has it in the precondition access a for reading or mutation. Bornat et al. [1] propose re-engineering the logic to work with predicates of the form $a \mapsto^f m$, where f is a fraction $0 \leqslant f \leqslant 1$. For example, if we have a with permission $f = \frac{1}{2}$, that means that some other thread, or a combination of threads, has the other $\frac{1}{2}$-permission. In case $f = 1$, we have a *full permission*, i.e., nobody else has any ownership of a. If we have a permission $f < 1$, then we can read from the pointer a, but cannot mutate it; for mutation, we need full permission, in order to prevent others from reading an inconsistent value of a, obtained while we're writing into a.

In our example, we can have the left thread own a with $\frac{1}{2}$-permission, and the right thread own b with $\frac{1}{2}$-permission. On the other hand, the resource owns x fully, but both a and b with the remaining $\frac{1}{2}$-permission. When the left thread acquires the resource upon locking, it ends up owning b with $\frac{1}{2}$-permission (hence, can read b but not modify it, and also, it knows that nobody else can modify b), and a and x with full permission. In the case of a, the left thread already had $\frac{1}{2}$ of a of its own, but it obtained the resource's $\frac{1}{2}$ with the acquisition. The thread also obtained the resource's full permission x. Hence, the thread can modify both x and a, just as needed. Dually with the right thread.

Another idea is that of *ghost objects*, used for example, by Jacobs and Piessens [16], and in tools by VCC of Cohen et al. [5]. Roughly, one can think of the idea as follows. We can keep all of a and b together as part of one object, a ghost object, which can be part of the resource (thus packaged with x as well), or kept

as a separate shared state (often owned by the clients of the resource). As a and b are shared, we have to allow programs in critical sections to mutate shared state. This is a bit different from the CSL approach, where we first acquire a resource, and can only mutate the state when the state is in private possession. With incr_2, one would still have an obligation to prove that the left thread does not mutate b, and the right thread does not mutate a, but this is doable. The symmetry of the proofs for the left and the right thread is achieved by abstracting a and b from the left and the right thread (i.e., making these threads take the appropriate auxiliary pointer as an argument, a for the left, b for the right thread). In many examples, one also needs to supply *auxiliary code* as an argument to the proofs, leading to *higher-order* auxiliary code. The idea of higher-order auxiliary code has been proposed by Jacobs and Piessens [16], and later generalized by Svendsen and Birkedal [34], da Rocha Pinto et al. [6], and Jung et al. [17]. This is presently a very active research area, and the work is currently focused on exploring the limits of what kind of programs can be specified and verified using such higher-order abstractions.

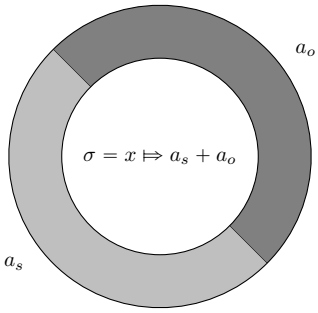
A shared characteristic of both approaches above is that we still have to account for the possibility that we may have more than two threads, or not even know statically how many threads we will have at run-time. So, in general, we need to organize the auxiliary state as an array, in the case of a fixed number of threads, or as some kind of linked structure, if we don't know the number of threads statically. Notice that the linked structure will have to somehow reflect the order in which the threads are forked and joined. For example, if we are iterating over a list, forking one thread to process one list element, we'll probably organize our auxiliaries in a list. If we're iterating over a tree, we'll need a tree of auxiliary values, etc. The definition of the resource invariant and the thread proofs will also need to abstract over the state that stores this linked structure, leading to even more higher-orderness and impredicativity. The clients will have to formalize their own desired linking behavior (depending on how they fork threads over the structure), and then implement and verify the discipline of allocating and deallocating the state holding auxiliary values upon thread forking and joining, respectively.

We will instead develop a different, and very simple approach of reformulating auxiliary state which enables proof reuse, and works in the functional setting. Moreover, the approach very naturally generalizes separation logic with *non-local* reasoning, i.e., reasoning about state *not owned* by the thread in question. This may be a bit surprising, because separation logic is usually presented as a logic for local reasoning. But we will see that we can reason equally well about non-local state, as long as that state is represented as a *local* variable in the specifications. A nice characteristic of the approach is that it does not require any sophisticated higher-order parametrization (though it does not preclude it either).

The idea may already be gleaned from the previous informal proof outline, where we treated a, b and σ very similarly, in the sense that the value of each of them is provided as an equality in the assertions. We build on this observation as follows. Instead of considering a and b as pointers in σ, we consider them as separate entities, which have the same first-class status as σ.

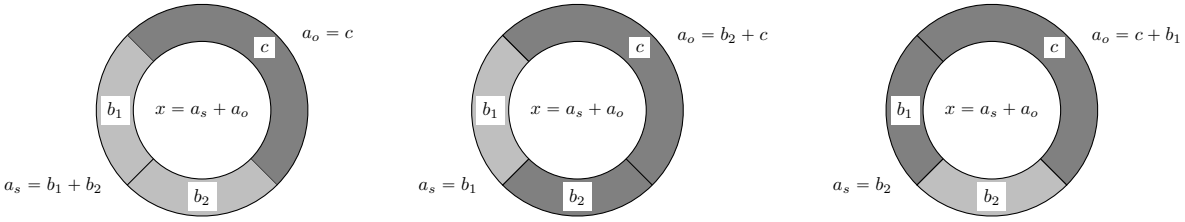
Here's the first approximation of the development. In addition to σ, we make *every* thread have *two* auxiliary variables, which we call a_s and a_o. These variables are *local*, in the same way that σ is local in the proof of each thread, but in contrast to the *global* variables a and b in the classical Owicki-Gries-O'Hearn setting. The variable a_s keeps track of how much the specified thread has added to x, and dually, a_o keeps track of how much *every other thread* has added to x. We also call a_s the *self* variable, and a_o the *other* variable, and similarly, speak of the *self* and *other* threads. E.g., when we are verifying the left thread, it is the left thread that is the *self* thread, and the right one is the *other*, and vice versa. Because of this dichotomy between *self* and *other*, we use the expression *subjective auxiliary state* to refer to these variables in tandem. Each thread can modify only its own self variable, but not its other variable; however, it can use the other variable in assertions, and relate its value to the manipulated state.

As mentioned, the variables a_s and a_o are not pointers in the heap, but are separate fields in the specification of programs, just like the heap σ. As for the resource invariant, we make it be a predicate I, which is now a predicate over a heap, and a variable α, which in each thread we will instantiate with $a_s + a_o$. More concretely, in the case of incr_2, we take $I \mathrel{\widehat{=}} (\sigma = x \mapsto \alpha)$, and the whole resource can be represented by the picture below.



Because a_s, a_o are now local "labels" to the specification of each thread, they stand for different values in different threads. In the incr example, in the the left thread, (a_s, a_o), will name what we used to name (a, b). But for the right thread, (a_s, a_o) will stand for (b, a), thus, inverting the roles and giving us the symmetry we wanted. Similarly, if in the classical setting we had three threads, we would have needed variables (a, b, c) to name the contributions of each of the three threads. In the subjective setting, we have (a_s, a_o) encoding $(a, b + c)$ in the left thread, $(b, c + a)$ in the middle thread, and $(c, a + b)$ in the right thread. But notice that in each of the three threads, the resource invariant holds, no matter the distribution of values between a_s and a_o, because the sum $a_s + a_o$ remains constant.

Note that a_s's and a_o's of various threads cannot be independent. After all, in the classical setting, when we have n threads, we have n global variables. But here, when we have n threads, we have $2n$ variables (two per thread). In particular, since a_s names *self* contribution, and a_o names *other* contribution, it should be apparent that we have to enforce the dependence between a_s's and a_o's, that *when one thread changes its own a_s, that should induce a change in a_o of every other thread*. Obviously, this dependence is critical in the rule for parallel composition, as it is the operation of parallel composition that changes the number of threads n. The following picture describes how a_s and a_o change upon forking and joining.



(1) in parent thread $e_1 \parallel e_2$ (2) in left thread e_1 (3) in right thread e_2

When a thread forks its children, then it gives a part of its *self*-contribution to the *self*-contribution of the left child, and the remaining part goes to the *self*-contribution of the right child. But, at the same time, the *other*-contributions of the children are computed out of the *other*-contribution of the parent, to account for the fact that *upon forking the left child becomes part of the other-environment for the right child, and vice-versa*. Upon joining, the *self*-contributions of the children are summed-up into the ending *self*-contribution of the parent, and *we are guaranteed* the existence of an *other*-contribution for the parent, which makes the *self* and *other*-contributions of the children coherent. In contrast to the pointer-based approaches outlined above, we are not focused on managing the ownership of a_s and a_o, but on managing their *contents*.

If we want to formalize this intuition, we first need to define a new operator $\circledast$ on assertions. Unlike in separation logic where the preconditions and postconditions were predicates over heaps, now we have them be predicates over a_s and a_o (ignoring heaps for a second).

$$(a_s, a_o) \models P \circledast Q \;\widehat{=}\; \exists b_1\ b_2.\, a_s = b_1 + b_2 \wedge (b_1, b_2 + a_o) \models P \wedge (b_2, a_o + b_1) \models Q$$

And then the inference rule for parallel composition becomes:

$$\frac{e_1 : \{P_1\}\ \{Q_1\} \qquad e_2 : \{P_2\}\ \{Q_2\}}{e_1 \parallel e_2 : \{P_1 \circledast P_2\}\{Q_1 \circledast Q_2\}}$$

The rule is clearly almost the same as the rule for parallel composition from CSL, with two important differences.

- First, the CSL rule talks about splitting *heaps*, not splitting values of auxiliary variables. The fully general rule should allow for splitting both. We achieve such a general rule by noticing that integers (under addition, with 0 as a unit element) and heaps (under disjoint union, with empty as the unit element) both form PCMs, and that, in fact, the definition of $\circledast$ can be changed to talk about PCMs in the abstract, with $\circledast$ being parametric in the choice of the PCM. Assuming that $\mathsf{valid}(a_s \bullet a_o)$, we define:

$$(a_s, a_o) \models P \circledast Q \;\widehat{=}\; \exists b_1\ b_2.\, a_s = b_1 \bullet b_2 \wedge (b_1, b_2 \bullet a_o) \models P \wedge (b_2, a_o \bullet b_1) \models Q$$

  In particular, if we want to have a parallel composition rule that splits both heaps and auxiliaries of type nat, we can take the PCM be the Cartesian product $\mathsf{heap} \times \mathsf{nat}$, which is a PCM under join and unit operations defined pointwise in terms of the joint and unit operations for heap and nat.
  It will turn out that the incr_2 example requires yet another auxiliary component, so we take our PCM to be $\mathsf{heap} \times \mathsf{nat} \times \mathsf{mutex}$, where the PCM mutex will be defined shortly. We will then refer to the three projections out of a_s, of types heap, nat and mutex, respectively, as σ_s, α_s, and μ_s, and dually, we refer to the projections out of a_o as σ_o, α_o, and μ_o. Thus, from now on, we retire the label σ when talking about the private heap of threads, and instead use σ_s. We reserve σ to talk about the heap of a resource.
  In this chapter, where we discuss lock-based concurrency, we will always assume that a_s (dually a_o) have these three projections, but the user will be free to choose the PCM type for α. This type will be nat in the case of incr, but we will see examples where that type is chosen differently. Also, the resource invariant is a predicate over σ and the α only, where we change the Owicki-Gries-O'Hearn protocol slightly, so that now $I[\alpha_s \bullet \alpha_o/\alpha]$ holds of the resource heap when unlocked.
  The particular choice of components with which to parametrize I may look unmotivated now (i.e., why parametrize wrt. σ and α, but not with μ), but we will explain this issue in more detail in the next chapter.
- Second, the CSL rule for parallel composition only splits the self heaps, but does not contain what would correspond to the σ_o component. This is so, because the σ_o projection is not actually all that useful, and it is not useful because we cannot impose any bounds on its evolution: any thread can modify its heap in any way it sees fit, e.g., it can modify, or even deallocate, all of its pointers. But, the other projections are not so ill-behaved. For example, the α_o component in the incr example may *only increment*. Thus, while σ_o does not appear in separation logic, and will never appear in our specifications, because there is nothing meaningful that can be said about it that is stable under interference, α_o will appear in the specifications, and crucially so.

Note 2.3.2 (Erasure).

Since now we have non-heap components in our state, it becomes important to differentiate between real state (i.e., heap) that the program uses during execution, and auxiliary state, which is introduced merely for the purposes of logical specification.

In particular, we must prevent that the real component becomes dependent on values in auxiliary components, i.e., we don't want to read a value from an auxiliary component, and then write that value into a heap location, as in that case, the auxiliary component is not auxiliary at all, but has influence on run time.

A similar requirement exists in the classical Owicki and Gries method [26], where they consider an inference rule, called *Auxiliary Variable Axiom*, stated as follows:

> If AV is an auxiliary variable set for S, let S' be obtained from S by deleting all assignments to variables in AV [...]. Then, if $\{P\}\ S\ \{Q\}$ is true and P and Q do not refer to any variables from AV, $\{P\}\ S'\ \{Q\}$ is also true.

Then, they also need to formally ensure that execution of one thread does not *interfere* with the proof of another, precisely in the use of auxiliary variables. Typically, then one has to make sure that the spec of one thread does not depend on auxiliary variables of another.

In the next chapter, when we discuss fine-grained concurrency, we will introduce proof obligations that ensure that our atomic commands respect the erasure property; that is, if run in two states which have equal

heap components, an atomic command produces two ending state with equal heap components, and also equal return results.

In this chapter, where we focus on lock-based concurrency, we instead ensure erasure as follows. While we provide commands for lookup and mutation that work on the heap component of σ, we do not provide commands for lookup and mutation on the components α and μ. Instead, we provide restricted, but sufficiently strong, commands which read the auxiliary and then immediately modify it, in one single, simultaneous operation. The value of the auxiliary that has been read will not be kept in any temporary variables, so it cannot be later written into the heap (which would invalidate erasure).

2.4 Subjective proof outlines

Let us take a look at a proof outline for incr_2 which uses these new abstractions. From there, we will then reverse-engineer the inference rules for each particular command involved.

As already mentioned, we take the type of a_s and a_o to be $\mathsf{heap} \times \mathsf{nat} \times \mathsf{mutex}$, and name the the three projections out of this type as σ, α and μ, with s and o as a subscript to σ, α and μ to indicate a projection out of a_s or a_o, respectively. The components σ and α are drawn from PCMs $(\mathsf{heap}, \cup, \mathsf{empty})$, and $(\mathsf{nat}, +, 0)$.

We next define the PCM mutex. The carrier is the two-element set $\mathsf{mutex} = \{\mathsf{Own}, \cancel{\mathsf{Own}}\}$. The value Own indicates that the thread which has $\mu_s = \mathsf{Own}$ is the one that currently holds the lock, and thus owns the shared resource x. Dually, $\cancel{\mathsf{Own}}$ indicates that the thread does not hold the lock.

The monoid operation $\bullet$ is defined with unit $\cancel{\mathsf{Own}}$, but $\mathsf{Own} \bullet \mathsf{Own}$ is undefined, thus formally capturing that two different threads cannot simultaneously hold the lock. This property of the definition will enable us to model the mutual exclusion between threads.

Then we can write the specification for the *single* incrementation thread, as follows, where we omit the resource context, assuming that it contains a binding $l \mapsto I$, for $I \mathrel{\widehat{=}} (\sigma = x \Mapsto \alpha)$, as before.

$$\mathsf{incr} : \{\sigma_s = \mathsf{empty} \wedge \alpha_s = 0 \wedge \mu_s = \cancel{\mathsf{Own}}\}\ \{\sigma_s = \mathsf{empty} \wedge \alpha_s = 1 \wedge \mu_s = \cancel{\mathsf{Own}}\}$$

In other words, we start with the empty heap, with no self-contribution to x, and without holding the lock. We finish with the empty heap and self-contribution of 1 (i.e., we added 1 to x), and we again don't hold the lock. But notice:

1. The specification of incr is agnostic to whether we run incr as the left, or as the right thread, unlike what we had before.
2. We start with empty input heap, which means that, by framing, we can start with any input heap. That aspect is unchanged from sequential separation logic, or from CSL. But, now we can also frame with respect to the components α_s and μ_s. In particular, by using a modified version of the frame rule (obtained by restricting our modified rule of parallel composition to idle c_2), the above specification can be transformed into the following one which makes it explicit that we can start with any self-contribution $\alpha_s = k$.

$$\mathsf{incr} : [k].\{\sigma_s = \mathsf{empty} \wedge \alpha_s = k \wedge \mu_s = \cancel{\mathsf{Own}}\}\ \{\sigma_s = \mathsf{empty} \wedge \alpha_s = k+1 \wedge \mu_s = \cancel{\mathsf{Own}}\}$$

Exercise 2.4.1. What is the frame predicate R needed to obtain this? A: $R = \sigma_s = \mathsf{empty} \wedge \alpha_s = k \wedge \mu_s = \cancel{\mathsf{Own}}$.

Exercise 2.4.2. What if we framed wrt. the mutex component also, to obtain:

$$\mathsf{incr} : [k].\{\sigma_s = \mathsf{empty} \wedge \alpha_s = k \wedge \mu_s = \mathsf{Own}\}\ \{\sigma_s = \mathsf{empty} \wedge \alpha_s = k+1 \wedge \mu_s = \mathsf{Own}\}$$

What does this spec says, and why is it valid?

A: The specification says, informally, that if a client calls incr while already holding the lock, then it will increment x by one. Of course, when read informally like this, the property is clearly not true, as no

incrementation of x actually occurs. We should not forget that the spec expresses partial correctness, i.e., the Hoare triple is valid, for it says that the command increments x only if it terminates. But if called when owning the lock, then incr doesn't terminate; it loops forever waiting for the lock to be released, which it never will, since it is the incrementing thread itself that's holding it.

We will establish the above spec shortly, but we can already use it to verify incr_2, since the proof outline only relies on the rule of parallel composition, and the definition of $\circledast$.

$$
\begin{array}{c}
\{\sigma_s = \mathsf{empty} \wedge \alpha_s = 0 \wedge \mu_s = \cancel{\mathsf{Own}}\} \\
\{(\sigma_s = \mathsf{empty} \wedge \alpha_s = 0 \wedge \mu_s = \cancel{\mathsf{Own}}) \circledast (\sigma_s = \mathsf{empty} \wedge \alpha_s = 0 \wedge \mu_s = \cancel{\mathsf{Own}})\} \\
\text{// Notice, the spec doesn't mention } \textit{other}\text{, so } \circledast \text{ behaves like } * \\
\begin{array}{c|c}
\{\sigma_s = \mathsf{empty} \wedge \alpha_s = 0 \wedge \mu_s = \cancel{\mathsf{Own}}\} & \{\sigma_s = \mathsf{empty} \wedge \alpha_s = 0 \wedge \mu_s = \cancel{\mathsf{Own}}\} \\
\mathsf{incr} & \mathsf{incr} \\
\{\sigma_s = \mathsf{empty} \wedge \alpha_s = 1 \wedge \mu_s = \cancel{\mathsf{Own}}\} & \{\sigma_s = \mathsf{empty} \wedge \alpha_s = 1 \wedge \mu_s = \cancel{\mathsf{Own}}\}
\end{array} \\
\{(\sigma_s = \mathsf{empty} \wedge \alpha_s = 1 \wedge \mu_s = \cancel{\mathsf{Own}}) \circledast (\sigma_s = \mathsf{empty} \wedge \alpha_s = 1 \wedge \mu_s = \cancel{\mathsf{Own}})\} \\
\{\sigma_s = \mathsf{empty} \wedge \alpha_s = 2 \wedge \mu_s = \cancel{\mathsf{Own}}\}
\end{array}
$$

The whole proof looks simple, as it does not contain any *other* components, just *self* ones. However, we will promptly see that *other* components (specifically, α_o) appears in the proof of incr itself.

Note 2.4.1 (On encapsulation).
What is important to notice, however, is that by moving away from global auxiliary state (the variables a and b) to subjective ones (local variables a_s and a_o, and projections out of them), we have made the proofs of the two threads above completely identical. Thus, incr needs to be *annotated only once*, and then *verified only once*, and its annotation and proof can be *reused many times, as a black box*. This is why we were able in the above example to actually use the name incr to abstract the program's code, auxiliary code, and proof, whereas in the classical setting, the code of the two threads had to be different, due to the different auxiliary state on the two sides (a on the left, b on the right).

Another way of saying the same thing, is that we have managed to get the program, it's auxiliary state, auxiliary code, and the proof, to be *encapsulated*, as they can all be packaged together into one indivisible whole, here called incr. This encapsulated whole can be substituted into any larger context expecting a program of a given precondition and postcondition, without invalidating the proof derivation of the program, or requiring a new annotation with different auxiliary state and auxiliary code.

Yet another way of saying this is that we have managed to make the code, auxiliary code, and the proofs of our programs become *equivalent*, since they can be composed together into an encapsulated whole, and they are all subject to the same substitution principle. This is yet another connection between separation logic and type theory, where the similar property of equivalence between programs and proofs, we have already mentioned in the previous lecture, under the name of Curry-Howard isomorphism.

We now turn to writing out a proof outline for incr, which will also tell us how to modify several commands and their inference rules. First, the code of incr must change somewhat. As we already mentioned, we do not have commands to directly update auxiliary state. Rather, we modify the old unlock command, to update α_s simultaneously with exiting the critical section. That is, the unlock command now comes annotated with a λ-expression that says how α_s should be changed. In particular, the program looks like

$$
\begin{array}{lll}
\mathsf{incr} = & \mathsf{lock}\ l; & \\
& t \leftarrow !x; & \\
& x := t + 1; & \\
& \mathsf{unlock}\ l\ (\lambda \alpha_s.\, \alpha_s + 1); &
\end{array}
$$

where the second argument to unlock says that α_s should be increased by one, as the lock is released. Here's how the proof outline should look like, wrt. the specification from before.

$\{\sigma_s = \mathsf{empty} \land \alpha_s = 0 \land \mu_s = \cancel{\mathsf{Own}}\}$
lock l;
$\{\sigma_s = x \Mapsto \alpha_o \land \alpha_s = 0 \land \mu_s = \mathsf{Own}\}$
$t \leftarrow !x$;
$\{\sigma_s = x \Mapsto \alpha_o \land \alpha_s = 0 \land \mu_s = \mathsf{Own} \land t = \alpha_o\}$
$x := t + 1$;
$\{\sigma_s = x \Mapsto (\alpha_o + 1) \land \alpha_s = 0 \land \mu_s = \mathsf{Own}\}$
unlock l $(\lambda\alpha_s.\, \alpha_s + 1)$;
$\{\sigma_s = \mathsf{empty} \land \alpha_s = 1 \land \mu_s = \cancel{\mathsf{Own}}\}$

The proof outline looks mostly natural. The one point that needs to be emphasized is that all the variables we use in the proof, such as σ_s, α_s, μ_s, and α_o remain fixed from one line to the other, unless the command explicitly changes them. This is especially important for α_o, which the specified program never changes – but the environment program does. However, to carry out the proof outline, we need to know that α_o remains fixed in critical sections.

We now consider how we need to change the inference rules that we have introduced so far, to make the above proof outline for incr go through (again, for simplicity, we ignore the context of resources Γ). We make the rules parametric in the PCM that is the type of α_s and α_o. In incr that PCM is natural numbers, but we can present the rules with an abstract PCM.

The rule for lock says that we start with an empty heap, unit α_s contribution, and not owning the lock. After we terminate, we have the lock ($\mu_s = \mathsf{Own}$), we haven't changed the contribution α_s, but now we have a heap σ_s which satisfies the invariant I, when we substitute σ_s for σ and α_o for α.

$$\begin{array}{rl} \mathsf{lock}\ l: & \{\sigma_s = \mathsf{empty} \land \alpha_s = \mathbb{1} \land \mu_s = \cancel{\mathsf{Own}}\} \\ & \{I[\sigma_s/\sigma, \alpha_o/\alpha] \land \alpha_s = \mathbb{1} \land \mu_s = \mathsf{Own}\} \end{array}$$

Again, by framing, we can change any and all of these components. Particularly interesting is changing α_s from unit to an arbitrary value a, but then instead of α_o as an argument to I, we have to pass $a \bullet \alpha_o$, i.e., obtain:

$$\begin{array}{rl} \mathsf{lock}\ l: [a]. & \{\sigma_s = \mathsf{empty} \land \alpha_s = a \land \mu_s = \cancel{\mathsf{Own}}\} \\ & \{I[\sigma_s/\sigma, a \bullet \alpha_o/\alpha] \land \alpha_s = a \land \mu_s = \mathsf{Own}\} \end{array}$$

This indicates that framing a value a into α_s works by taking the same value out of α_o. Thus, in order to make sure that the invariant I holds, we need to apply I to the old value of α_o, and the latter is obtained by joining a to the new value of α_o.

Exercise 2.4.3. Derive the framed version of the spec for lock. Use the framing predicate $R = (\sigma_s = \mathsf{empty} \land \alpha_s = a \land \mu = \cancel{\mathsf{Own}})$, similar to what we used in the framing of incr. Work out how the argument to the invariant I in the postcondition changes from α_o to $a \bullet \alpha_o$.

A: By applying the rule of frame to the unframed spec, and looking at the postcondition, we get that $(a_s, a_o) \models (I[\sigma_s/\sigma, \alpha_o/\alpha] \land \alpha_s = \mathbb{1} \land \mu_s = \mathsf{Own}) \circledast R$. We want to show that we can weaken this postcondition as desired in the second spec. That is, we want to show:

$$(a_s, a_o) \models I[\sigma_s/\sigma, a \bullet \alpha_o/\alpha] \land \alpha_s = a \land \mu_s = \mathsf{Own}$$

To start with, let us name the individual components of a_s and a_o as: $a_s = (\sigma, \alpha, \mu)$ and $a_o = (\theta, \beta, \nu)$.

By definition of $\circledast$, we can split a_s into a join $a_s = (\sigma, \alpha, \mu) = (\sigma_1, \alpha_1, \mu_1) \bullet (\sigma_2, \alpha_2, \mu_2)$, such that:

$$((\sigma_1, \alpha_1, \mu_1), (\theta \bullet \sigma_2, \beta \bullet \alpha_2, \nu \bullet \mu_2)) \models I[\sigma_s/\sigma, \alpha_o/\alpha] \land \alpha_s = \mathbb{1} \land \mu_s = \mathsf{Own}$$

and

$$((\sigma_2, \alpha_2, \mu_2), (\theta \bullet \sigma_1, \beta \bullet \alpha_1, \nu \bullet \mu_1)) \models R$$

From the first equation, we get: $\alpha_1 = \mathbb{1}$, $\mu_1 = \mathsf{Own}$ and $I[\sigma_1/\sigma, \beta \bullet \alpha_2/\alpha]$. From the second equation, we get: $\sigma_2 = \mathsf{empty}$, $\alpha_2 = a$, $\mu_2 = \cancel{\mathsf{Own}}$. Substituting these equalities into the expression about the invariant, we get:

$$I[\sigma_1 \bullet \sigma_2/\sigma, a \bullet \beta/\alpha]$$

But then, also:

$$((\sigma_1 \bullet \sigma_2, \alpha_1 \bullet \alpha_2, \mu_1 \bullet \mu_2), (\theta, \beta, \nu)) \models I[\sigma_s/\sigma, a \bullet \alpha_o/\alpha] \wedge \alpha_s = a \wedge \mu_s = \mathsf{Own}$$

In other words:

$$(a_s, a_o) \models I[\sigma_s/\sigma, a \bullet \alpha_o/\alpha] \wedge \alpha_s = a \wedge \mu_s = \mathsf{Own}$$

which is what we wanted to show.

The rule for unlock says that we start with owning the lock, and with some contribution a, and with a heap σ_s satisfying $I[\sigma_s/\sigma, \Phi(a) \bullet \alpha_o/\alpha]$. That is, the heap is such that the invariant I holds, but with the argument α_s taken to be $\Phi(a)$. When we unlock, we release the heap (σ_s becomes empty), and the lock (μ_s becomes ~~Own~~), and, importantly, the self-contribution α_s is simultaneously changed from a into $\Phi(a)$, to be coherent with the resource heap, as required by the Owicki-Gries-O'Hearn protocol. The postcondition does not say anything about α_o, because once the lock is released, α_o can be changed arbitrarily by the environment.

$$\mathsf{unlock}\ l\ \Phi : \ [a].\{I[\sigma_s/\sigma, \Phi(a) \bullet \alpha_o/\alpha] \wedge \mu_s = \mathsf{Own} \wedge \alpha_s = a\}\{\sigma_s = \mathsf{empty} \wedge \alpha_s = \Phi(a) \wedge \mu_s = \cancel{\mathsf{Own}}\}$$

Notice, in order for this rule to be sound, Φ cannot be arbitrary, but has to satisfy the side-condition that it is *local*. We will define this notion further below, and discuss some of its properties.

The rules for the primitive commands get extended with a clause which says that unmodified self fields remain fixed, whereas α_o remains fixed only when in critical section, i.e., when $\mu_s = \mathsf{Own}$. Of course, we cannot guarantee that α_o remains fixed when we don't own the pointer x. But, we do need that α_o remains fixed in critical sections, in order to carry out the proof of incr.

We use the lookup and mutation commands in the example, so let us write out their rules in the new format.

Lookup:

$$\begin{aligned} !\ :\ \Pi x : \mathsf{ptr}.\, [v : A, m, a].\, &\{\sigma_s = x \mapsto v \wedge \alpha_s = \mathbb{1} \wedge \mu_s = m \wedge (m = \mathsf{Own} \implies \alpha_o = a)\}\ A \\ &\{r = v \wedge \sigma_s = x \mapsto v \wedge \alpha_s = \mathbb{1} \wedge \mu_s = m \wedge (m = \mathsf{Own} \implies \alpha_o = a)\} \end{aligned}$$

Mutation:

$$\begin{aligned} :=\ :\ \Pi x : \mathsf{ptr}.\, \Pi v : A.\, [m, a].\, &\{\sigma_s = x \mapsto - \wedge \alpha_s = \mathbb{1} \wedge \mu_s = m \wedge (m = \mathsf{Own} \implies \alpha_o = a)\} \\ &\{\sigma_s = x \mapsto v \wedge \alpha_s = \mathbb{1} \wedge \mu_s = m \wedge (m = \mathsf{Own} \implies \alpha_o = a)\} \end{aligned}$$

Finally, the rule for return also has to talk about the fixity of α_o:

$$\begin{aligned} \mathsf{return}\ :\ \Pi v : A.\, [m, a].\, &\{\sigma_s = \mathsf{empty} \wedge \alpha_s = \mathbb{1} \wedge \mu_s = m \wedge (m = \mathsf{Own} \implies \alpha_o = a)\}\ A \\ &\{r = v \wedge \sigma_s = \mathsf{empty} \wedge \alpha_s = \mathbb{1} \wedge \mu_s = m \wedge (m = \mathsf{Own} \implies \alpha_o = a)\} \end{aligned}$$

The condition $(m = \mathsf{Own} \implies \alpha_o = a)$ which describes that α_o remains fixed when we're in the critical section, expresses what is technically called "stability". In general, a predicate R is stable if it remains invariant under the interference of the other threads. For example, all the assertions in the above specs are stable, because we only provide equations about α_o when we know that we are in a critical section, and hence α_o doesn't change. But, for example, the following predicate, which doesn't guard for a critical section, is not stable, because there is no guarantee that α_o will remain equal to a. Indeed, if $m = \cancel{\mathsf{Own}}$, other threads may acquire the resource, and make additional contributions, thus modifying α_o.

$$\sigma_s = x \mapsto - \wedge \alpha_s = \mathbb{1} \wedge \mu_s = m \wedge \alpha_o = a$$

In general, in a lock-based setting, where we only acquire and release the state of one shared resource, the following is a concrete definition of stability (though, the definition will change significantly in the fine-grained setting).

Definition 2.4.1 (Stability in a lock-based setting)**.** *Predicate R over σ_s, α_s, μ_s and α_o is* stable *if R implies* [if $\mu_s = \mathsf{Own}$ then α_o else $-/\alpha_o$]R. *In other words, if R holds of the state (real and auxiliary taken together, but ignoring σ_o and μ_o since these are never useful) at some point in time, then it continues to hold under the modifications to this state incurred by the other threads. In particular, if the* self *thread owns the lock, then the other threads cannot incur any modifications to α_o.*

Notice that any predicate R that only talks about self variables, e.g., σ_s, α_s and μ_s is stable, because self variables can only be modified by the specified thread, and not by the interfering threads.

We close the discussion on stability by emphasizing that the frame rule has the form which accounts for stability of R. It looks like:

$$\frac{c : \{P\}A\{Q\} \qquad R \text{ is stable}}{c : \{P \circledast R\}\ A\ \{Q \circledast R\}}$$

We have used the frame rule before to frame incr and lock without explicitly stating the stability of the frame predicate R, but those predicates indeed were stable, since they only talked about self variables.

We next discus the side conditions on Φ that we omitted in the unlock rule. Intuitively, parallel composition is an operation on threads which is commutative and associative. Correspondingly, the inference rule for parallel composition, relies solely on $\bullet$ being a PCM operation. Since Φ operates on α_s, and the modification done by Φ may appear deeply inside a thread, the modification done by Φ has to commute and associate with whatever is being done on α_o by other threads. Otherwise, the rule for parallel composition will not be sound. Φ's of this nature, which we will formally define promptly, go by the name of *local functions*, and are the usual notion of morphism that goes with PCMs, just like monotone functions go with lattices, or continuous functions go with CPOs.

Here is the formal definition.

Definition 2.4.2. *A function Φ on a PCM U is local if for all $\alpha, \beta \in U$, such that* $\mathsf{valid}(\alpha \bullet \beta)$ *and* $\mathsf{valid}(\Phi(\alpha))$*:*

$$\Phi(\alpha \bullet \beta) = \Phi(\alpha) \bullet \beta$$

The definition says that β can be treated as a "frame" when applying Φ, and it won't influence the result of Φ. Clearly, the function $\lambda\alpha_s.\, \alpha_s + 1$ is local in the PCM of natural numbers with addition.

2.5 Example: coarse-grained set with disjoint interference

It may seem that the locality requirement imposed on function Φ, makes Φ essentially just provide an increment to the existing value of α_s. In other words, it may seem that Φ is uniquely determined by its value at the unit element $\mathbb{1}$. This is not quite true, because local functions may be partial; it may even be possible that Φ is undefined on $\mathbb{1}$. To illustrate a situation like that, we next present a more involved example, compared to incr_2. The example illustrates a situation where Φ does not just add to α_s, but modifies α_s in a more intricate, but still local, way.

The example is about a coarse-grained set data structure. The structure keeps a set, and provides methods for adding and removing an element from a set. The interference is disjoint, because each thread will need to prove that: (1) the element it wants to remove is already in the set, and none of the concurrent threads intends to remove it, or add another copy, and dually (2), that the element it wants to add is not already in the set, and none of the concurrent threads intends to add it, or remove the copy that is planned for addition. Thus, two parallel threads cannot interfere trying to add or remove the same element.

We show how we can reason about parallel composition of such operations in isolation of each other. This is possible even if the operations interfere physically on the data structure; for example, if the set is implemented with a splay tree, the shape of the memory will depend on the order of operations. Fortunately, at the abstraction level of set membership, the proof isn't sensitive to the non-commutativity of physical operations.

We start by assuming a sequential set library, whose procedures sadd and sremove we will run in a critical section. We have a predicate set which takes pointer x and a mathematical set S of, say, integers, and describes that the heap σ describes a structure, headed at pointer x, storing the set. For example, one possible implementation of set may be as follows, though the exact definition does not matter for the example.

$$\mathsf{set}(x, S) \ \widehat{=}\ \exists\alpha{:}\mathsf{intlist}.\ \mathsf{permutes}(\alpha, S) \wedge \mathsf{is\_list}\ x\ \alpha$$

The method specs show that we're adding/removing from the set, in a critical section, while preserving the auxiliaries (strictly speaking, we need to rename $[\sigma_s/\sigma]\mathsf{set}(x, S)$, but we don't bother with that detail

now).

$$\begin{array}{rcl} \mathsf{sadd}(x,v) & : & [a,b].\{v \notin S \wedge \mathsf{set}(x,S) \wedge \mu_s = \mathsf{Own} \wedge \alpha_s = a \wedge \alpha_o = b\} \\ & & \quad \{\mathsf{set}(x, S \cup \{v\}) \wedge \mu_s = \mathsf{Own} \wedge \alpha_s = a \wedge \alpha_o = b\} \\ \mathsf{sremove}(x,v) & : & [a,b].\{\mathsf{set}(x, S \cup \{v\}) \wedge \mu_s = \mathsf{Own} \wedge \alpha_s = a \wedge \alpha_o = b\} \\ & & \quad \{\mathsf{set}(x,S) \wedge \mu_s = \mathsf{Own} \wedge \alpha_s = a \wedge \alpha_o = b\} \end{array}$$

We obtain a coarse-grained concurrent library, by wrapping each sequential method into a critical section, i.e. $\mathsf{cadd}(v) = \langle \mathsf{sadd}(x,v) \rangle$ and $\mathsf{cremove}(v) = \langle \mathsf{cremove}(x,v) \rangle$.

Although the coarse-grained implementation may be inefficient because threads must contend for the entire data structure, we can choose suitable auxiliaries $\mathsf{in}(v)$ and $\mathsf{out}(v)$ to specify the procedures in a local manner that only depends on the element being manipulated. In this case, $\mathsf{in}(v)$ states that v is in the set, and nobody else will remove it, and $\mathsf{out}(v)$ states that v is not in the set, and nobody else will add it. We abbreviate $\alpha_s \doteq v$ to mean $(\sigma_s = \mathsf{empty} \wedge \alpha_s = v \wedge \mu_s = \cancel{\mathsf{Own}})$.

$$\begin{array}{rcl} \mathsf{cadd}(v) & : & \{\alpha_s \doteq \mathsf{out}(v)\}\{\alpha_s \doteq \mathsf{in}(v)\} \\ \mathsf{cremove}(v) & : & \{\alpha_s \doteq \mathsf{in}(v)\}\{\alpha_s \doteq \mathsf{out}(v)\} \end{array}$$

We can thus reason locally in each thread to prove the effect of concurrently adding or removing three distinct elements:

$$\begin{array}{c} \{\alpha_s \doteq \mathsf{in}(a) \bullet \mathsf{out}(b) \bullet \mathsf{in}(c)\} \\ \begin{array}{ccc|ccc|ccc} \{\alpha_s \doteq \mathsf{in}(a)\} & & & \{\alpha_s \doteq \mathsf{out}(b)\} & & & \{\alpha_s \doteq \mathsf{in}(c)\} \\ \mathsf{cremove}(a) & \| & & \mathsf{cadd}(b) & \| & & \mathsf{cremove}(c) \\ \{\alpha_s \doteq \mathsf{out}(a)\} & & & \{\alpha_s \doteq \mathsf{in}(b)\} & & & \{\alpha_s \doteq \mathsf{out}(c)\} \end{array} \\ \{\alpha_s \doteq \mathsf{out}(a) \bullet \mathsf{in}(b) \bullet \mathsf{out}(c)\} \end{array}$$

To verify the methods, the auxiliaries should maintain the distributed, partitioned knowledge of which elements are in or out of the data structure. We pick the auxiliary PCM

$$((\mathsf{nat} \rightharpoonup_{\mathsf{fin}} \mathsf{bool}) \cup \{\mathsf{undefined}\}, \cup, \mathsf{empty}, \mathsf{valid})$$

of finite partial maps from nat to bool, with the join $\cup$ taking the union of maps with disjoint domain, the empty map empty as unit, and the valid being true on all maps aside from undefined. We define $\mathsf{in}(v), \mathsf{out}(v) : \mathsf{nat} \rightharpoonup_{\mathsf{find}} \mathsf{bool}$ to map v to true and false, respectively, and be undefined elsewhere. It is immediately clear that these elements are valid, and moreover, if we take one of them, and flip its value at v, we obtain the other:

$$\begin{array}{rcl} \mathsf{in}(v)[v \mapsto \mathsf{false}] & = & \mathsf{out}(v) \\ \mathsf{out}(v)[v \mapsto \mathsf{true}] & = & \mathsf{in}(v) \end{array}$$

Intuitively, a map α will indicate which elements our set contains, but also our intention of changing the status of the element. For example, if α sends v to true, that will indicate that v is in the set. Moreover, a thread that wants to remove v will have $\mathsf{in}(v)$ in the precondition, and will thus have exclusive ownership of v. Because the join operation of the PCM is disjoint union, no other thread can contain in its precondition a map that has v in its domain (hence, no $\mathsf{in}(v)$ or $\mathsf{out}(v)$). Thus, no other thread will interfere with manipulating v. Dually, if v is not in the set, that may be indicated by α being undefined on v, or by having α send v to false. The difference between the two choices is that the latter may be used to indicate the ownership of v, even if v is not in the set. One may say that we indicate the *intention* to manipulate v. If a thread's precondition contains $\mathsf{out}(v)$, no other thread's precondition can contain $\mathsf{in}(v)$, or $\mathsf{out}(v)$, or any other map with v in the domain. Hence hence no other thread will interfere with manipulating v.

The resource invariant I is the following predicate over σ and α, where α is a finite map from the above PCM:

$$I \;\widehat{=}\; \mathsf{set}(x, \alpha^{-1}(\mathsf{true})) \wedge \mathsf{valid}(\alpha)$$

I states that the set contains exactly the elements that α maps to true. Then we can prove cadd (and cremove similarly) as follows (where we suppress the various valid statements, for simplicity):

$\{\alpha_s \doteq \mathsf{out}(v)\}$
$\{\sigma_s = \mathsf{empty} \wedge \alpha_s = \mathsf{out}(v) \wedge \mu = \cancel{\mathsf{Own}}\}$
lock l;
$\{\mathsf{set}(x, (\mathsf{out}(v) \bullet \alpha_o)^{-1}(\mathsf{true})) \wedge \alpha_s = \mathsf{out}(v) \wedge \mu = \mathsf{Own}\}$
$\{v \notin (\mathsf{out}(v) \bullet \alpha_o)^{-1}(\mathsf{true}) \wedge \mathsf{set}(x, (\mathsf{out}(v) \bullet \alpha_o)^{-1}(\mathsf{true})) \wedge \alpha_s = \mathsf{out}(v) \wedge \mu = \mathsf{Own}\}$
sadd(x, v)
$\{\mathsf{set}(x, (\mathsf{out}(v) \bullet \alpha_o)^{-1}(\mathsf{true}) \cup \{v\}) \wedge \alpha_s = \mathsf{out}(v) \wedge \mu = \mathsf{Own}\}$
$\{\mathsf{set}(x, (\mathsf{in}(v) \bullet \alpha_o)^{-1}(\mathsf{true})) \wedge \alpha_s = \mathsf{out}(v) \wedge \mu = \mathsf{Own}\}$
unlock l Φ_v;
$\{\sigma_s = \mathsf{empty} \wedge \alpha_s = \mathsf{in}(v) \wedge \mu_s = \cancel{\mathsf{Own}}\}$
$\{\alpha_s \doteq \mathsf{in}(v)\}$

But, now we need to engineer the function $\Phi(v)$ that serves as auxiliary code, so that we can satisfy the precondition of unlock. What's required for this is that Φ_v sets to true at point v the argument α. Specifically, we need:

$$\Phi_v(\mathsf{out}(v)) = \mathsf{in}(v)$$

but, the function Φ should work over all maps that contain v, and the function should be local.

We define Φ_v to take a map α and return another map that behaves like α on all elements, except on v, which it sets to true. However, if v is not defined in α, then the whole of $\Phi_v(\alpha)$ is the undefined element. Notice that the undefined element is not the same as everywhere-undefined-map. In analogy with heaps, the latter we basically call empty, and its a valid map in the technical sense (i.e. valid(empty)). The former is not actually a map; it's a dummy element we added to the PCM, just to be able to make Φ_v a total function, i.e., to have an element to return when no appropriate map exists.

$$\Phi_v(\alpha) \;\widehat{=}\; \begin{cases} \alpha[v \mapsto \mathsf{true}] & \text{if } v \in \mathsf{dom}\ \alpha \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

Clearly, this function gives us the desired $\Phi_v(\mathsf{out}(v)) = \mathsf{in}(v)$, but why is Φ_v local?

The proof goes as follows. Let us assume that $\mathsf{valid}(\alpha \bullet \beta)$ and $\mathsf{valid}(\Phi_v(\alpha))$. The latter means $v \in \mathsf{dom}(\alpha)$. Thus, also $v \in \mathsf{dom}(\alpha \bullet \beta)$. Then, by definition, $\Phi_v(\alpha \bullet \beta)$ equals $(\alpha \bullet \beta)[v \mapsto \mathsf{true}] = \alpha[v \mapsto \mathsf{true}] \bullet \beta = \Phi(\alpha) \bullet \beta$.

As an interesting discussion, consider what would happen if we didn't require $\mathsf{valid}(\Phi_v(\alpha))$ as a precondition in the definition of locality? Then we would not know that $v \in \mathsf{dom}(\alpha)$, and in particular, it could be $\mathsf{valid}(\alpha \bullet \beta)$ and $v \in \mathsf{dom}(\beta)$. But then $\Phi_v(\alpha) \bullet \beta$ is undefined, because $\Phi_v(\alpha)$ is undefined, and $\bullet$ preserves undefinedness, but $\Phi_v(\alpha \bullet \beta)$ is defined, as it sets v in the β part of $\alpha \bullet \beta$. Thus, the locality equation does not hold.

On the other hand, we may consider changing the definition of $\Phi_v(\alpha)$ as follows. Instead of returning an undefined element when $v \notin \mathsf{dom}(\alpha)$, we could just ignore v, and return the whole α unchanged. But then, the locality equation again fails to hold, since when $\beta = \mathsf{out}(v)$, then $\Phi_v(\alpha \bullet \beta)$ sends v to true, but $\Phi_v(\alpha) \bullet \beta$ sends v to false.

At any rate, it is important to notice that auxiliary local functions do not need to be merely simple additions, but that we can employ partiality, as a form of a conditional, to determine when the auxiliary function should be applied.

Chapter 3

Fine-grained Concurrent Data Structures

We have so far considered examples that lock the whole shared data structure before proceeding to read from it, or modify it. Obviously, such coarse-grained locking is very inefficient, as it creates contention on the structure's lock. This is why state-of-the-art concurrent data structures are *lock free*, aka. *fine-grained*. Such structures start from the observation that often times only pieces of the structure need to be locked, while other threads can continue to work on the unlocked parts. Most of the times, no actual locks are even used, but the exclusive access to the critical parts of the data structure is achieved by hardware-provided synchronization operations, such as e.g., CAS (compare-and-swap and compare-and-set) operation.

We will see an example of a fine-grained data structure, and of the CAS operation, shortly. We will be able to verify such programs by systematically generalizing some aspects of the coarse-grained program logic from the previous chapter. Most of the important inference rules, such as the rule for sequential composition, parallel composition, framing, and other structural rules, will remain intact. What will change is the way we describe a shared resource, and the way we specify the primitive commands, such as lookup and mutation, lock, and unlock.

The material in this chapter is based on the design of Fine-grained Concurrent Separation Logic (FCSL) [18, 19], which has been used to mechanically verify in Coq a number of examples [31–33], including those presented in these notes. For simplicity, we do not present FCSL in full, but only selected aspects of it. Thus, the presentation will omit a number of formalization details that are not relevant for the omitted parts. Thus, the text in this chapter should be read as explanatory prose to describe some aspects of a larger system, and not as a full formal reference for FCLS. For the latter, the above papers remain the main reference.

3.1 Treiber's stack

To make matters concrete and illustrate the basic concepts, we consider the "Hello world" example of fine-grained data structures: Treiber's stack [35], which works as follows. Physically, the stack is kept as a singly-linked list in the heap, with a sentinel pointer *snt* pointing to the stack top. To simplify matters a bit, we store the pair (v, q) which includes the value v of the node (of type A), and the tail pointer q, in a single cell, rather than in consecutive locations in the heap.

Placeholder for a figure of snt variable pointing to the stack top; stack encoded as a linked list.

The structure exports two methods, push and pop, which we explain in turn.

```
push (v : A) : unit :=
  q <- alloc (v, null);
  let recfun loop () :=
    p <- !snt;
    q := (v, p);
    ok <- CAS (snt, p, q);
    if ok then return () else loop ()
  in
    loop ()
  end
```

The call to push(v) first allocates a node q, storing v. This node is supposed to become the new top of the stack. push attempts to en-link the node into the stack, by changing the contents p of the sentinel to q, via CAS(snt, p, q).

In this example, CAS stands for "compare-and-set" and is a standard multi-threading synchronization primitive implemented on many hardware architectures. It is one of the *read-modify-write* operations [11] that read and modify a memory cell, atomically, i.e., without interference from other threads. Operationally, CAS(snt, p, q) behaves as follows: it reads the contents of snt, and if the contents equals p, then it writes q into snt, and returns true to indicate success. Otherwise, it returns false to indicate failure. CAS is atomic, but in pseudo-code, absent any interference, we can represent it as:

$$
\begin{array}{l}
\mathsf{CAS}(snt, p, q) = \\
\qquad t \leftarrow !snt; \\
\qquad \mathsf{if}\ t = p\ \mathsf{then}\ snt := q; \\
\qquad \mathsf{return}\ (t = p)
\end{array}
$$

Because CAS is atomic, no other threads can modify snt during its execution. One can thus think of CAS as an operation that locks snt before modifying it, which is actually how CAS is implemented on architectures that do not provide it natively. In these lectures, we assume that CAS, as well as other read-modify-write operations that we may need, is given to us as a primitive. We mention here that there is a somewhat different version of CAS which does not return a boolean, but returns the value read.

$$
\begin{array}{l}
\mathsf{CAS}(snt, p, q) = \\
\qquad t \leftarrow !snt; \\
\qquad \mathsf{if}\ t = p\ \mathsf{then}\ snt := q; \\
\qquad \mathsf{return}\ t
\end{array}
$$

This version of CAS is called "compare-and-swap".

In this implementation of push, we use the boolean version of CAS (though, obviously, we could have used the other version too). Be it as it may, it is important that we use CAS instead of simply writing q to snt, in order to be able to recognize if some interfering thread has changed the top of the stack during the execution of the several commands that attempt to en-link q. If an interference occurred, enlinking q may lead to an inconsistent stack, thus we want to recognize such situations and backtrack. CAS signals such interference, by returning false if the contents of snt has changed away from p, in which case push loops trying to enlink again.

```
pop(): option A :=
  p <- !snt;
  if p == null then return None
  else
    (v, q) <- !p;
    ok <- CAS (snt, p, q);
    if ok then return Some v else pop();
```

Pop reads the sentinel, and records the address of the top node into p. If the stack is not empty, pop reads the contents of the first node into v, and the address of the second node into q. Note that by this moment

in the execution, p need not be the top node anymore, because interfering threads may have pushed other nodes onto the stack, or even popped p itself. Thus, when pop tries to de-link p by writing q to snt, it does so using CAS, looping, as in the case of push if it detects interference.

Note that pop does not deallocate the de-linked node p, which remains in the data structure as garbage (thus, the above implementation has a memory leak). This is part of the design, to prevent the so-called ABA problem [11, §10]: if p is deallocated, then some other push may allocate it again, and place it back on top of the stack, possibly after several other pushes. A procedure that observed p on top of the stack, will be tricked into ignoring this interference, and producing an invalid stack, if its CAS sees p on top of the stack again.

Note 3.1.1 (ABA problem).
The ABA problem has several workarounds, and designing new ones is an active area of research. We mention one class of workarounds, known under the name of "deferred reclamation". One implementation of deferred reclamation is garbage collection. Once a memory cell is garbage collected, we know that no references to it exist, and in particular, *snt* cannot hold a reference to a reclaimed node. Thus, in garbage-collected languages, the CAS in the implementation of pop, cannot be tricked in the way described above. Another possible implementation of deferred reclamation, which does not rely on GC, is called epoch-based reclamation [8]. For simplicity, we ignore the issues of reclamation in these notes, and just embrace the described memory leak.

From the above description, one can see that the heap that Treiber's stack occupies during its use has the structure of a forest of inverted tree, as the following picture illustrates.

Placeholder for a figure of the inverted tree that is the Treiber's shared heap.

This, for example, makes it simple to implement a snapshot procedure, which traverses the stack, without pushing or popping any elements, and without locking the stack, and returns the contents of the stack *that was current when the snapshot procedure was invoked*. Such a procedure will be resistant to any stack modifications that occurred concurrently with it, because the nodes, once pushed onto the stack, are never actually changed, even if they are eventually popped.

3.2 Method specification

How should we proceed to specify and verify this example? There are several problems that we have to solve towards that goal, which we proceed to discuss and gradually formalize.

First, obviously, in Treiber's example, unlike in other examples we considered, no global locking of the structure takes place. Thus, we cannot rely on Owicki-Gries-O'Hearn model of transferring a structure from shared to private state before modification. Even more, although one may consider that CAS-ing on the sentinel corresponds to locking, it seems pointless to model such locking by ownership transfer to private state, just to subsequently execute the simple mutation of CAS. Thus, the first generalization that we need to add to our previous logic is to allow that pointers that reside in shared state (i.e., in the resource) can be read and mutated directly, without insisting on a prior transfer to private state. Such accesses to shared state will be useful not just for CAS-ing on the sentinel, but also for accessing the nodes of the stack (e.g., if we wanted to implement a snapshot operation that we mentioned before).

Thus, we will modify the notion of resource and update the Owicki-Gries-O'Hearn protocol, as follows. Previously, we used the resource invariant as a description of a set of states of the concurrent data structure. Thus, the resource invariant was one-half of a state transition system (STS). The other half, the transitions, we did not need to explicitly specify in the coarse-grained setting, because what the threads could do the resource was restricted by the language to just locking and unlocking. In the fine-grained setting, we want to allow the user to specify the transitions in a way required by the application. If the user wants to implement

locks, a-la Owicki-Gries-O'Hearn, the transitions will describe the locking, unlocking, and heap ownership change. But in the case of Treiber's stack, the transitions will specify something else: the state modifications that correspond to pushing and popping of elements, as well as a few other simple operations.

Second, we need to decide on how to specify the stack methods.

This is a problem that we can already discuss in the sequential and in the coarse-grained setting, where push and pop first lock the whole stack before modifying it. The solution we come up with for the coarse-grained setting, will already be close to what we need for the fine-grained setting.

Just for a warm-up, let us start with the sequential implementation of a stack. In the sequential case, the following is a reasonable specification, where we assume that the stack stores elements of type A.

$$\begin{array}{rcl} \mathsf{push}(v) & : & [\alpha].\,\{\mathsf{stack}(\alpha)\}\{\mathsf{stack}(v :: \alpha)\} \\ \mathsf{pop} & : & [\alpha].\,\{\mathsf{stack}(\alpha)\}\ \mathsf{option}\ A\,\{r = \mathsf{None} \wedge \alpha = \mathsf{nil} \wedge \mathsf{stack}(\alpha) \vee \\ & & \qquad \exists v\ \beta.\, r = \mathsf{Some}\ v \wedge \alpha = v :: \beta \wedge \mathsf{stack}(\beta)\} \end{array}$$

Here $\mathsf{stack}(\alpha)$ is a predicate describing the layout of the stack, here we take it to be a description of a singly-linked list, rooted in snt:

$$\mathsf{stack}(\alpha) \ \widehat{=}\ \exists p.\, snt \mapsto p * \mathsf{is\_list}\ p\ \alpha$$

However, this specification is not very useful in the concurrent case. Imagine starting from an empty stack ($\mathsf{stack}(\mathsf{nil})$), and executing a $\mathsf{push}(2)$. We obtain the postcondition $\mathsf{stack}(2 :: \mathsf{nil})$, but this postcondition does not tell us anything about the stack, as by the time $\mathsf{push}(2)$ terminates, the interfering threads could have pushed and popped many other elements, including popping the 2 that we just pushed.

In the coarse-grained concurrent examples, we tracked the effects of the methods by adding PCM elements to the α_s component. The commutativity and associativity was essential, because it abstracts the order in which concurrent threads are scheduled and interleaved. But, what should we do with the example such as the stack, where pushing and popping in a stack obviously do not commute and associate.

As it turns out, it is still possible to find a PCM which will enable us to see the effects of pushing and popping as commutative and associative operations. We use the algebraic structure of *histories*, which encode, by a timestamp, the moment in time when a certain event, a push, or a pop, occurred. Similar data structures are quite common in concurrency. They are often called histories, or, alternatively, traces, and are finite or infinite lists of events that the program performed. They can also often be time-stamped, but need not be so. What is new in our approach, is the recognition that such structures, simply because they form a PCM, naturally arise from separation logic and combine well with other separation logic infrastructure. For example, we will be able to use the same notation for histories that we use for heaps, and the same rule of frame will apply to both.

There are several ways in which we can define the PCM of histories, neither of them necessarily better than the others. Here's one that we use for Treiber's example. The carrier set is

$$(\mathsf{nat} \rightharpoonup_{\mathsf{fin}} \mathsf{list}\ A \times \mathsf{list} A) \cup \{\mathsf{undefined}\},$$

the set of finite maps from nats to pairs of lists, with an undefined element added on top. Each finite map takes a timestamp t (drawn from nat), and maps it into an operation on stacks that we want to record as having occurred at time t. For example, the singleton map $t \Mapsto (vs, 3 :: vs)$ is a singleton history, saying that at time moment t, 3 was pushed onto the stack. The stack had the initial value represented by the list vs, and, appropriately, the ending value changed to $3 :: vs$. Similarly, $t \Mapsto (2 :: vs, vs)$ represents a pop at time t.

Notice how the notation $t \Mapsto op$ for singleton histories is the same as the notation $p \Mapsto v$ for singleton heaps. We are justified in using identical notation, because heaps and histories are essentially the same algebraic structure. Both are finite maps (albeit with somewhat different ranges), the join operation in both cases is disjoint union $\cup$, and both have only one non-valid element undefined. The experience shows that this is a rather general pattern; in many examples, the PCM that ends up being appropriate for specifying some example, will either be a PCM of finite maps of some kind (witness the disjoint set example), or a PCM of finite sets of some kind, of a PCM of nats, or a product thereof.

One difference between heaps and histories will be in the use. The examples will typically impose, in the resource invariants, many properties on histories that are not usually required of heaps. One such example property is *continuity*. By means of example, the global history of a stack, will look something like follows, with the ending state of one timestamp always matching the beginning state of the next.

$$1 \Mapsto (\mathsf{nil}, 3 :: \mathsf{nil}) \cup 2 \Mapsto (3 :: \mathsf{nil}, 42 :: 3 :: \mathsf{nil}) \cup 3 \Mapsto (42 :: 3 :: \mathsf{nil}, 3 :: \mathsf{nil}) \cup \cdots$$

Definition 3.2.1. *Given a history τ and timestamp $t \in \mathsf{dom}(\tau)$, such that $\tau(t) = (xs, ys)$, we write $\overleftarrow{\tau}(t)$ for xs, and $\overrightarrow{\tau}(t)$ for ys.*

Definition 3.2.2 (Continuous history)**.** *A continuous history τ is one that:*

1. *Contains no gaps between timestamps. If a timestamp $t \in \mathsf{dom}(\tau)$, then for every $t' < t$, $t' \in \mathsf{dom}(\tau)$.*
2. *For every t such that $t + 1 \in \mathsf{dom}(\tau)$, the beginning list at timestamp $t + 1$ equals the ending list at timestamp t. More formally, $\overleftarrow{\tau}(t + 1) = \overrightarrow{\tau}(t)$.*

We will use histories to represent the operations performed by a given thread, as follows. Let us assume that our a_s and a_o contain components drawn from the PCM of histories (in addition to σ_s, σ_o, and the others), and let us name these components τ_s and τ_o. Here, the letter τ is meant to evoke *time*, just like the letter σ was meant to evoke *space* in the previous chapter. Then τ_s encodes the operations, and their timestamps, performed by the self thread, and τ_o represents the operations, and their timestamps, performed by all other threads combined. Each of τ_s and τ_o need not be continuous; indeed, each may have gaps between its timestamps, corresponding to the times when the thread was preempted by the scheduler, and thus, inactive. However, we will insist, via the resource invariants, that $\tau_s \bullet \tau_o$ is continuous in the above sense, to reflect that τ_s and τ_o, taken together, hold the full history of the data structure.

Depending on the example, we may further restrict, via the resource invariant, the operations that appear in the range of a history. For example, the histories we use to specify stacks will, at each timestamp, contain lists that encode either a push or a pop, i.e., they are of the form $(vs, v :: vs)$ or $(v :: vs, vs)$, respectively. The exception will be the timestamp 0, which will always contain a pair of equal components (vs, vs), where vs encodes the initial contents of the stack.

At this point, we introduce some further notation.

Definition 3.2.3 (PCM-induced ordering)**.** *Given a PCM U, and $x, y \in U$, we write $x \sqsubseteq y$ if there exists $z \in U$, such that $y = x \bullet z$.*

For example, in the case of heaps, we have $p_1 \Mapsto 3 \sqsubseteq (p_2 \Mapsto 42 \cup p_1 \Mapsto 3 \cup p_3 \Mapsto 27)$. We use similar notation on histories. For example, $\tau \sqsubseteq \mathsf{undefined}$ for every history τ.

Definition 3.2.4 (History prefix)**.** *Given a history τ, and a timestamp t, we write $\tau^{\leqslant t}$ for the sub-history of τ containing only timestamps not greater than t:*

$$\tau^{\leqslant t} = \lambda t'.\ \begin{cases} \tau(t') & t' \leqslant t \\ \mathit{undefined} & \mathit{otherwise} \end{cases}$$

Similarly, we denote by $\tau^{<t}$ the sub-history of τ containing only timestamps smaller than t.

As one can easily prove, a prefix of a history is ordered below the original history in the PCM-induced ordering, that is:

$$\begin{array}{lcl} \tau^{\leqslant t} & \sqsubseteq & \tau \\ \tau^{<t} & \sqsubseteq & \tau \end{array}$$

With these notations and properties, we can now introduce the following specification for the stack methods, using τ_s and τ_o in the pre- and postconditions. We omit the resource invariant for now, but will fill it in later, where it will take a form of an STS.

$$\begin{array}{rcl} \mathsf{push}(v) & : & [\tau].\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o\} \\ & & \quad \{\sigma_s = \mathsf{empty} \wedge \exists t\ vs.\ \tau_s = t \Mapsto (vs, v :: vs) \wedge \tau \sqsubseteq \tau_o^{<t}\} \\ \mathsf{pop} & : & [\tau].\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o\}\ \mathsf{option}\ A \\ & & \quad \{\sigma_s = \mathsf{empty} \wedge \exists t.\ r = \mathsf{None} \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o^{\leqslant t} \wedge \overrightarrow{\tau_o}(t) = \mathsf{nil} \vee \\ & & \qquad \exists v\ vs.\ r = \mathsf{Some}\ v \wedge \tau_s = t \Mapsto (v :: vs, vs) \wedge \tau \sqsubseteq \tau_o^{<t}\} \end{array}$$

What do these specifications say?

In both methods, we start with the empty self heap and empty self history. In particular, the empty self history means that the caller of the method has not performed any previous stack operations. This is not a

loss of generality, however, as by framing, we can always increase the initial self history. Similarly, we can frame the initial self heap, as standard in separation logic.

In both methods, we use the logical variable τ to bind a subset of the initial τ_o. We will use τ in the postcondition, to directly relate the effects of the methods with the interference of other thread, which is what the variable τ_o represents.

In the postcondition of $\mathsf{push}(v)$, the self history is extended by a timestamp t, the time at which the push occurred; that is, the contents of the heap was changed from vs to $v :: vs$. Moreover, the timestamp t is larger than all the timestamps in τ, which is what the conjunct $\tau \sqsubseteq \tau_o^{\leqslant t}$ says. We emphasize at this point that the same conjunct says one more thing: namely, that the other history τ_o grows monotonically. Indeed, we chose $\tau \sqsubseteq \tau_o$ in the precondition. From the postcondition, the new τ_o is such that $\tau \sqsubseteq \tau_o^{\leqslant t} \sqsubseteq \tau_o$. Since the choice of τ in the precondition is arbitrary (i.e., we could have chosen the full initial τ_o), it means that τ_o could only have grown by the time push terminates. In other words, we can only add new operations to the history τ_o (and by duality, to τ_s), but we cannot remove any that have already been placed into it.

In the postcondition of pop, the procedure can either return $r = \mathsf{None}$, if pop is executed at the time when the heap is empty, or return $r = \mathsf{Some}(v)$. In the former case, we do not extend the self history τ_s, because no actual change to the heap takes place. We do know, however, that there exists a timestamp t, at which the heap was empty. Moreover, if the heap was empty, then it was emptied by other threads, and hence $\overrightarrow{\tau_o}(t) = \mathsf{nil}$. Similarly as for push, we know that τ_o must grow, hence $\tau \sqsubseteq \tau_o^{\leqslant t}$. However, we want to allow that the timestamp t may possibly appear in τ; that is, t need not be generated strictly *after* pop was called. The stack may have been emptied before the call to pop, and t is the last timestamp generated just before the call to pop. Hence, we use $\tau \sqsubseteq \tau_o^{\leqslant t}$, rather than the stronger $\tau \sqsubseteq \tau_o^{<t}$ that we used in the case of push.

On the other hand, if pop returns $\mathsf{Some}(v)$, then the self history τ_s is increased to $t \Mapsto (v :: vs, vs)$, to indicate that a pop occurred at time t. Moreover, the timestamp t is generated during the call to pop, hence, as in the case of push, $\tau \sqsubseteq \tau_o^{<t}$.

Exercise 3.2.1. Give a specification in the above style to the snapshot procedure that traverses the stack and returns a mathematical sequence representing stack's contents at the time of invocation.

Note 3.2.1 (Relationship to linearizability).

Linearizability, introduced by Herlihy and Wing [12], is the most common correctness criterion for the implementation of concurrent data structures. Without getting into the details of its definition and properties, we informally describe the key ideas behind linearizability, and relate them to our stack specification.

A concurrent data structure with a number of methods in its API (e.g., push and pop in the case of a stack) is linearizable, if every execution of a number of concurrent method calls can be "sequentialized". Sequentializing means rearranging the execution order of the methods into a sequence, as follows. If a method A finished before another concurrent method B started, then A must be sequenced before B. If the methods A and B overlapped, then they may be sequenced in either order.

Each concurrent method is given a specification in the form of a sequential method that performs the same effect as the concurrent method, had there been no interference from other threads. For example, in the case of stacks, the concurrent push would be related to some sequential implementation of push, and similarly for pop. Often times, the connection is not established to sequential methods, but to sequential specifications, such as the one we provided for sequential stacks. Linearizability requires that the concurrent execution we are trying to sequence, produces the same result as the execution of the sequential variants of each method, in the order given by the sequence.

If we can prove that such a sequence can always be found, then we say that concurrent structure is linearizable wrt. the sequential variants of the methods. The important property of a linearizable structure is that, when we want to reason about the clients of the structure, we can pretend that clients actually call the sequential variants of the methods, not the concurrent ones. The distinction between the two variants will not be observable, but relying on sequential calls significantly simplifies reasoning about the clients. Thus, the clients never need to see the concurrent code of the data structure.

Now, notice how our specification of push and pop above states *precisely the same* properties, though it states them somewhat differently. In particular, instead of quantifying over all method executions, the desired properties are stated by the specs, locally, for one push, and for one pop.

Indeed, focusing on the specification of push (and pop is similar), we see that the execution of push produces a distinct point in time, which we denote by the timestamp t, at which the effect of push takes place, logically. The time-stamp t appears in a larger *sequence* that is the history of other stack operations. The operation identified by the timestamp t appears *after* the operations that finished before it, which is captured by the conjunct $\tau \sqsubseteq \tau_o^{<t}$, as we explained. Moreover, the operation at time t was a push, as the stack was changed from vs to $v :: vs$.

Since our system satisfies a substitution principle as usual with type systems, we obtain the last reasoning component provided by linearizability. That is, the reasoning about clients can be carried out of the specifications (i.e., types) of the object's methods, but need not consider the code of the methods, as is usual with type systems. Also as usual with type systems, any other implementation for of the methods that can be ascribed the given types, can be substituted into the clients, without invalidating the clients' type derivation (which corresponds to the client's proof).

As we shall see in the next chapter, however, we can use our subjective Hoare types to specify programs that are not actually linearizable, but are still useful and interesting. Thus, our Hoare style approach seems strictly more general than linearizability, although that remains to be formally proved.

3.3 The resource definition: resource invariant and transition

The missing piece in the above specifications is the definition of the resource that, as we announced, will be given in the form of a state transition system (STS).

In the coarse-grained setting, the set of states was described by a resource invariant, a predicate over σ and α. σ that was interpreted as the shared heap that is transferred into the private state of the thread in the critical section. α was interpreted as the join $\alpha_s \bullet \alpha_o$ in the critical section.

In the fine-grained setting, we make it explicit that a resource contains shared state that may be arbitrary, not just a heap, and we abstract over both a_s and a_o, rather than over their join. This will be important later on when we consider transitions, where we will need to say that we increase a_s while keeping a_o fixed, which we cannot say if we are only given their join to work with. Thus, our state is now divided into a_s (self state), a_o (other state) and a_j (shared, or joint state). The components a_s and a_o have a common type, which is, moreover, a PCM. The type of a_j is separate, and need not be a PCM. Each of the three components can have named projections, as customary from the coarse-grained setting. When we need to refer to the whole triple of state components, we use s to range over it, e.g., $s = (a_s, a_j, a_o)$.

For example, in the case of Treiber's stack, we name the following components to the state.

1. σ_s will name the private heap of a thread. In this heap, the push method will allocate the node to be enlinked into the stack. Dually, there is a σ_o projection as well, but, as usual from the coarse-grained setting, we do not use it.
2. τ_s will name the private history of a thread, i.e., the set of timestamped operations (push and pop) that the thread in question has performed. Dually, τ_o names the private history of the environment.
3. σ_j is the shared heap storing the sentinel of the stack, the linked list implementing the stack itself, and the garbage state of popped, but non-deallocated nodes. In the coarse-grained setting, we would have called this component σ in the definition of the resource. However, here, the threads will not be acquiring σ_j in critical sections, since we do not have critical sections, but the modifications to σ will be done directly by CAS.
4. We use π to name the contents of the sentinel, χ to name the part of σ_j where the stack is laid out, γ for the area of σ_j containing the popped, but non-deallocated nodes (i.e., the garbage), and α for the abstract contents of the stack.

Thus, σ_s and τ_s are projections out of a_s (dually for σ_o, τ_o and a_o), and σ_j, π, χ, γ and α, are the projections out of a_j.

Note 3.3.1 (Convention).

By convention, in the proof outlines, we assume, and thus not explicitly state, that $a_s \bullet a_o$ is valid in its PCM, and that the heap components of a_s, a_o and a_j, are all disjoint. These properties will be required as a side condition on resource invariants, when defining a resource.

Also by convention, in each of these components we always have to identify the heap sub-components that are to be considered as real state, and thus survive erasure of auxiliary state.

In Treiber's example, the real components are σ_s, σ_j and σ_o, while all the other ones: τ_s, τ_o, π, χ, γ and α, are auxiliary. Notice that χ and γ are auxiliary even if they are of type heap, because we just use them to name sub-components of σ_j.

Given a state $s = (a_s, a_j, a_o)$ with the components as described above for Treiber's stack example, we define the erasure of s, denoted as $\bar{s}$ to be the heap

$$\bar{s} = \sigma_s \cup \sigma_j \cup \sigma_o$$

With this notation, we can now define the resource invariant for Treiber's stack as follows, where we abbreviate $\tau_s \bullet \tau_o$ as τ.

$$I \;\widehat{=}\; \sigma_j = (snt \Mapsto \pi) \cup \chi \cup \gamma \wedge \mathsf{is\_list}\ \pi\ \alpha\ \chi \wedge \alpha = \overrightarrow{\tau}(\mathsf{last}\ \tau) \wedge \mathsf{continuous}\ \tau \wedge \mathsf{pushpop}\ \tau$$

In prose, we say that the joint heap σ_j contains a sentinel snt, pointing to π. In turn, π heads a singly-linked list, and the heap that contains this singly-linked list is labeled χ. This is stated using $\mathsf{is\_list}\ \pi\ \alpha\ \chi$. We previously used $\mathsf{is\_list}$ as a predicate with two arguments, π and α, and kept the heap implicit in separation logic style, but now we pass the heap χ explicitly. The abstract contents α of the linked list equals the second projection (i.e., the ending list) read from the last timestamp in the history τ. Here, $\mathsf{last}\ \tau$ is the largest timestamp in τ (or 0 in the case of empty history). $\mathsf{Pushpop}\ \tau$ is defined to say that τ contains, at 0, the initialization pair, and that at every other timestamp, the pair represents a push or a pop.

$$\mathsf{pushpop}\ \tau \;\widehat{=}\; (\exists vs.\, \tau(0) = (vs, vs)) \wedge \forall t > 0.\, \exists v\ vs.\, \tau(t) = (vs, v :: vs) \vee \tau(t) = (v :: vs, vs)$$

We also say that the resource's heap σ_j contains an extra chunk γ, which stands for the garbage space, but we do not relate the value of γ to τ. In principle, we could say that γ stores all the values that the history τ record as popped, or we can also describe that γ has the structure of the inverted tree, as described before, but, for simplicity, we do not bother with that detail now.

Exercise 3.3.1. Write out the formal definitions of the predicates that would relate grb to τ as in what we omitted above. Why would such relating of grb and τ be useful?

Notice that the resource invariant doesn't say anything about σ_s, which remains unrestricted, in order to allow for threads to keep arbitrary contents in their private heaps. This component, however, will be used in the definition of transitions, as we shall see.

A transition of a resource is a relation between resource states (hence, triples $s = (a_s, a_j, a_o)$ satisfying I). In the particular case of Treiber's stack, they encode that the program can push, pop, read the sentinel, or read the contents of another node in shared state. Thus, unlike with the coarse-grained setting, in the fine-grained setting, we declare even the most basic memory operations. This is natural, since the auxiliary code that needs to go with these basic memory operation may be different from application to application.

For convenience of use, we allow that a transition may be indexed by a number of inputs, thus we define functions from the inputs to transitions. We define the transitions, using Hoare-like notation again, though with a somewhat different syntax. I.e., we write $P \rightsquigarrow Q$, potentially with logical variables $\bar{v}$ bound up-front. Semantically, $[\bar{v}].P \rightsquigarrow Q$ denotes the following relation in set theory:

$$[\bar{v}].\, P \rightsquigarrow Q = \{(s_1, s_2) \mid \exists \bar{v}.\, P\ s_1 \wedge Q\ s_2\}$$

Here, s_1 and s_2 are drawn from the state space described by the resource invariant.

With this notation, the pop family of transitions, indexed by pointers p and q, denoting the current and the new top of the stack, respectively, can be written as:

$$\begin{aligned} pop(p,q) \quad \widehat{=} \quad [v, vs, h', grb, T].\, & (\pi = p \wedge \chi = p \Mapsto (v, q) \cup h' \wedge \gamma = grb \wedge \alpha = v :: vs \wedge \tau_s = T) \rightsquigarrow \\ & (\pi = q \wedge \chi = h' \wedge \gamma = (p \Mapsto (v, q) \cup grb) \wedge \alpha = vs \wedge \\ & \tau_s = T \cup \mathsf{fresh}(\tau) \Mapsto (v :: vs, vs)) \end{aligned}$$

The projection σ_s does not feature in *pop*, which, by convention, we take it to mean that the projection remains unchanged by the transition. The part to the left of $\rightsquigarrow$ is like a precondition, and serves to name, using the variables v, h', vs, T, the components of the pre-state that we want to rearrange in the post-state. In the case of the pop transition, the contents π of the sentinel is changed to point to the second node q in the stack, and the self history τ_s is extended with a new item signifying that the abstract contents of the stack is changed from $v :: vs$ to vs. Here $\mathsf{fresh}(T)$ denotes the smallest unused timestamp in the history T (i.e. $\mathsf{fresh}(T) = \mathsf{last}(T) + 1$). Simultaneously, the old top node p of the stack is retired into the garbage chunk γ of the shared heap.

For the push transition, we have the following, where p denotes the current, and q the new top of the stack.

$$\begin{aligned} push(p,q) \quad \widehat{=} \quad [v, vs, h_1, h_2, T].\, &(\sigma_s = q \Mapsto (v,p) \cup h_1 \wedge \pi = p \wedge \chi = h_2 \wedge \alpha = vs \wedge \tau_s = T) \rightsquigarrow \\ &(\sigma_s = h_1 \wedge \pi = q \wedge \chi = q \Mapsto (v,p) \cup h_2 \wedge \alpha = (v :: vs) \wedge \\ &\tau_s = T \cup \mathsf{fresh}(\tau) \Mapsto (vs, v :: vs)) \end{aligned}$$

The transition says that we take a node headed at q, with contents v and next node p (the old head of the stack), and we make q the new head, while simultaneously removing p from the the private state σ_s. The abstract contents of the stack is appropriately changed from vs to $v :: vs$ to reflect that a push happened, and similarly for the history. The γ is unmentioned, and hence remains unchanged.

By default, we also assume that every resource we will define contains the idle transition, which is the diagonal relation, hence, encodes no changes to the state. In our notation, where by convention, unmentioned projections out of the state remain invariant, we can write this transition as follows, not mentioning any labels:

$$idle \quad \widehat{=} \quad \top \rightsquigarrow \top$$

On occasion, we will want to use the subset of the *idle* relation, that is, activate the *idle* transition only under some condition. We will write such a subset as *idle* P, which is defined as

$$idle\ P = \{(s,s) \mid s \in I \wedge s \in P\}$$

3.4 Stability

We close the discussion of the state transition systems by defining the notion of stability with respect to an STS. In the following definitions, C is an STS whose state space is given by I and the set of transitions is T.

Definition 3.4.1 (Transposed state)**.** *Given a state $s = (a_s, a_j, a_o)$, the transposed state $s^\top$ of s is defined as*

$$s^\top \ \widehat{=}\ (a_o, a_j, a_s)$$

Definition 3.4.2 (Transposed transition)**.** *Given a transition $\rho \in T$, the transposed transition $\rho^\top$ is defined as*

$$(s_1, s_2) \in \rho^\top \ \widehat{=}\ (s_1^\top, s_2^\top) \in \rho$$

Definition 3.4.3 (C-reachable)**.** *Starting from the state s_1, the state s_2 is C-reachable in one step, written*

$$s_1 \underset{C}{\rightarrow} s_2$$

iff there exists a transition $\rho \in T$, such that $(s_1, s_2) \in \rho^\top$.

We also say that s_2 is "reachable by other threads" from s_1, or "reachable by others".

Definition 3.4.4 (Stability)**.** *A predicate P is stable under C iff*

$$\forall s_1\ s_2\ .\, P\ s_1 \wedge s_1 \underset{C}{\rightarrow}{}^* s_2 \implies P\ s_2$$

Note 3.4.1 (Mutation to private state).

Treiber's example uses two primitives: alloc and :=, which operate over private state. That is, alloc extends σ_s with a new pointer, and := mutates a pointer that is already in σ_s.

A careful reader may have noticed that in the previous chapter, we did not consider transitions that correspond to allocation and mutation. We now explain the reasons.

In FCSL, these two operations can actually be accounted for by defining a resource, and transitions in that resource, which implement allocation, deallocation, lookup, and mutation. We can call such a resource "Private state" and provide it with the projections σ_s, σ_j and σ_o. This resource keeps in σ_j the information about un-allocated pointers (i.e., the free list). Upon allocation, moves the first free pointer into σ_s, and dually upon deallocation.

Under such a resource, the alloc and := operations have the expected specifications:

$$\mathsf{alloc}(v) : \{\sigma_s = \mathsf{empty}\}\ \mathsf{ptr}\ \{\sigma_s = r \Mapsto v\}$$

and

$$p := v : \{\sigma_s = p \Mapsto -\}\ \{\sigma_s = p \Mapsto v\}.$$

In FCSL one can combine resources together by means of algebraic operations. For example, the resource for private state can be combined with a resource using the other components from Treiber's example, e.g. π, χ, τ, etc. Such a combination essentially corresponds to a Cartesian product of STSs, and will give us the resource for Treiber as given above (plus transitions for allocation, deallocation, and mutation, which we omit for simplicity).

We do not discuss the algebraic operations for STS combinations in these lectures, but note that adding the Treiber's components to the specs for Allocate and := above, will extend the specifications as follows.

$$\begin{aligned}\mathsf{alloc}(v) : [\tau].\ &\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o\}\\ &\{\sigma_s = r \Mapsto v \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o\}\end{aligned}$$

and

$$\begin{aligned}p := v : [\tau].\ &\{\sigma_s = p \Mapsto - \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o\}\\ &\{\sigma_s = p \Mapsto v \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o\}\end{aligned}$$

One may say that we are *framing* the first specification above, which used only σ, with new components of τ. In particular, alloc and := do not interact with τ, but, if we want to use alloc and := in a resource that contains τ components, we need to explicitly provide *stable* description of the behavior of the τ components.

In this particular case, we say that τ_s does not change, but that τ_o can only grow. These are indeed stable properties under the transitions of the Treiber resource that we just introduced. The conjunct $\tau_s = \mathsf{empty}$ is stable because τ_s is a *self* component, and thus cannot be changed by the execution of *other* threads. The conjunct $\tau \sqsubseteq \tau_o$ is stable because the transitions of Treiber's resource can only add *push* or *pop* items into the history τ_o, but there is no transition that allows items to be removed. Thus τ_o only grows under interference.

3.5 Auxiliary code and primitive actions

We next modify the Treiber's methods with auxiliary state and code. The key places for annotation are the occurrences of the atomic primitives for accessing the memory. We rewrite the methods as shown below, where we name by readSentinel, tryPush, readNode, and tryPop, the annotated versions of the primitives, which we define promptly.

```
push (v : A) : unit :=
  q <- alloc (v, null);
  let recfun loop () :=
    p <- readSentinel;
    q := (v, p);
    ok <- tryPush (p, q);
    if ok then return () else loop ()
  in
    loop ()
  end
```

```
pop(): option A :=
  p <- readSentinel;
  if p == null then return None
  else
    (v, q) <- readNode p;
    ok <- tryPop (p, q);
    if ok then return Some v else pop();
```

A common way in the literature of defining the annotated versions of primitive commands is to simply pair up the auxiliary code in atomic brackets. That is, use notation of the form $\langle op_1; op_2 \rangle$, to denote that the primitive physical operation op_1 is execute simultaneously with the operation op_2 which works strictly over auxiliary state. Often, the notation is somewhat generalized to allow the use of conditionals as well. For example, to describe the behavior tryPush, one can write:

$$\begin{aligned}\mathsf{tryPush}(p,q) = \langle & r \leftarrow \mathsf{CAS}(snt,p,q);\\ & \mathsf{if}\ r\ \mathsf{then}\ \text{modify } \pi,\ \chi, \text{ etc. as described by transition push}\\ & \mathsf{else}\ \text{keep state invariant;}\\ & \mathsf{return}\ ok\rangle\end{aligned}$$

This captures that when restricted to non-auxiliary state, $\mathsf{tryPush}(p,q)$ performs a $\mathsf{CAS}(snt,p,q)$, as one would expect from the non-annotated version of the method. But, $\mathsf{tryPush}(p,q)$ also modifies the auxiliary state, to match the described semantics of this auxiliary state. E.g., q is moved from private to shared state, π changed from p to q to reflect the change of the sentinel, etc., as per the definition of the transition $push(p,q)$. If the CAS was unsuccessful then nothing is modified.

One can see from the above description that an annotated atomic primitive is very similar to a transition, as it encodes a relation between the input state, output state, and return result r (whereas a transition does not have a return result r). Moreover, the atomic primitive is not an arbitrary relation, but a choice between transitions of the given resource STS. In the case of tryPush, we chose to transition either by $push$, or by $idle$, depending on r.

We may thus say that the STS-style description of a shared resource encodes the space of the possibilities in which a data structure may evolve. But an atomic primitive, and by extension, a general program, which is essentially just a compositions of atomic primitives, describes paths through the such an STS. This generalizes to the fine-grained case the notion of Owicki-Gries-O'Hearn protocol. In particular, a specification with a precondition P and postcondition Q, in the STS $C = (I, T)$ is ascribed to a program c that, if executed in a state satisfying P, either diverges, or terminates in a state satisfying Q, and throughout the execution only goes through states satisfying I, taking transitions from T. The program c can be preempted during the execution, and other processes scheduled to operate on the resource while c is idle, but such processes are also restricted to take transitions only from T. Nevertheless, the system remains fault-avoiding; that is, the program c preserves safety despite the interference of others (which is bounded by C, as explained).

To emphasize this intuition, we adopt a somewhat different notation from above for defining primitive actions. We do not distinguish between real and auxiliary state and code, and immediately write the action as a conditional that returns the transition that we want. For example, in the case of tryPush:

$$\mathsf{tryPush}(p,q) = \mathsf{if}\ r = \mathsf{true}\ \mathsf{then}\ \mathsf{push}(p,q)\ \mathsf{else}\ \mathsf{idle}$$

In this notation, there is nothing that makes it apparent that tryPush corresponds to a CAS. Thus, we have an obligation to check that, when the auxiliary state is erased, the relation defined by tryPush indeed corresponds to a CAS. More formally, if $(s_1, s_2, r) \in \mathsf{tryPush}(p,q)$, then $\bar{s_1}$ steps by $\mathsf{CAS}(snt,p,q)$ into $\bar{s_2}$, and returns result r. One can easily see that this property holds.

We can similarly define tryPop:

$$\mathsf{tryPop}(p,q) = \mathsf{if}\ r = \mathsf{true}\ \mathsf{then}\ \mathsf{pop}(p,q)\ \mathsf{else}\ \mathsf{idle}$$

and show that it satisfies the similar erasure property.

Sometimes, the action need not choose between transitions, because its transition is uniquely determined. This is the case for readSentinel and readNode, which both invoke the $idle$ relation, but under a conditions.

For example, readSentinel invokes the *idle* transition under the condition that the return result $r = \pi$, in order to signify that it returns the address of the top node of the stack, but otherwise does not modify anything. Similarly readNode(p) equates the result r to the contents of the node p, by invoking the *idle* transition under the condition $(p \Mapsto r \sqsubseteq \chi \cup \gamma)$, to indicate that r equals pair (v, q) for which $p \Mapsto (v, q)$ appears in the shared data structure, but otherwise readNode does not modify anything.

$$\begin{array}{rcl} \mathsf{readSentinel} & = & \mathit{idle}\ (\pi = r) \\ \mathsf{readNode}(p) & = & \mathit{idle}\ (p \Mapsto r \sqsubseteq \chi \cup \gamma) \end{array}$$

Once each atomic operation for Treiber's stack is defined, the next step is to ascribe it a Hoare triple. For example, in the case of tryPush, we have a specification like the following:

$$\begin{array}{l} \mathsf{tryPush}(p, q) : [v, \tau].\ \{\sigma_s = q \Mapsto (v, p) \land \tau_s = \mathsf{empty} \land \tau \sqsubseteq \tau_o\} \\ \qquad\qquad \{\mathsf{if}\ r\ \mathsf{then}\ \sigma_s = \mathsf{empty} \land \exists t\ vs.\ \tau_s = t \Mapsto (vs, v :: vs) \land \tau \sqsubseteq \tau_o^{<t} \\ \qquad\qquad\ \ \mathsf{else}\ \sigma_s = q \Mapsto (v, p) \land \tau_s = \mathsf{empty} \land \tau \sqsubseteq \tau_o\} \end{array}$$

Whereas the definition of an action describes a relation between the pre-state, post-state and r that corresponds to a single operational atomic step on a rich state, the Hoare triple is ascribed to a program under the assumption that the program may be preempted by threads respecting the protocol given by the STS. As a consequence, the preconditions and the postconditions in the Hoare triple are always stable predicates under the considered STS.

In the particular case of tryPush it is easy to see that the above specification has a precondition and a postcondition that are stable. In particular, the conjuncts about the σ_s component are stable because σ_s is a *self* component, and hence invariant under modifications to other and joint state.

On the other hand, the conjuncts about the other component are of the form $\tau \sqsubseteq \tau_o$ and $\tau \sqsubseteq \tau_o^{<t}$. These are stable, because if they hold for a fixed value of τ, t and τ_o, then they will continue to hold under interference of other threads. The interference does not change τ and t, and, while it can change τ_o, it can only *increase it*. Indeed, the later is true, because the transitions declared for the Treiber's STS either do not change the history (*allocate*, *write*, *readSentinel*, *readNode*), or add items to history (*push*, *pop*), but never subtract any items.

An even more interesting is the spec for tryPop:

$$\begin{array}{l} \mathsf{tryPop}(p, q) : [v, \tau].\ \{\sigma_s = \mathsf{empty} \land \tau_s = \mathsf{empty} \land \tau \sqsubseteq \tau_o \land p \Mapsto (v, q) \sqsubseteq \chi \cup \gamma\} \\ \qquad\qquad \{\sigma_s = \mathsf{empty} \land \mathsf{if}\ r\ \mathsf{then}\ \exists t\ vs.\ \tau_s = t \Mapsto (v :: vs, vs) \land \tau \sqsubseteq \tau_o^{<t} \\ \qquad\qquad\qquad\qquad\ \ \mathsf{else}\ \tau_s = \mathsf{empty} \land \tau \sqsubseteq \tau_o\} \end{array}$$

This command tries to de-link the node $p \Mapsto (v, q)$ from the head of the stack. Thus, the precondition is asking us that the node of that form is actually in $(\chi \cup \gamma)$, which is the initial heap of the structure χ, including the garbage γ. Notice, the precondition does not ask us to show that p is the top of the stack, since this is not a stable property. Indeed, even if p at some point is at the top of the stack, an interfering thread could pop it, thereby removing it from the top. Nevertheless, even if the node is popped, it is not deallocated, and its contents is not changed by any transition. Thus, the assertion $p \Mapsto (v, q) \sqsubseteq \chi \cup \gamma$ is stable.

In the postcondition, the $r = \mathsf{true}$ case is identical to the already described case when pop returns $r = \mathsf{true}$. In the $r = \mathsf{false}$ case, we establish the precondition for pop, so that we can safely make the recursive call to pop, and loop.

For readSentinel, we have:

$$\begin{array}{l} \mathsf{readSentinel} : [\tau].\ \{\sigma_s = \mathsf{empty} \land \tau_s = \mathsf{empty} \land \tau \sqsubseteq \tau_o\} \\ \qquad\qquad\ \ \{\sigma_s = \mathsf{empty} \land \tau_s = \mathsf{empty} \land \exists t.\ \tau \sqsubseteq \tau_o^{\leqslant t} \land \\ \qquad\qquad\qquad \mathsf{if}\ r = \mathsf{null}\ \mathsf{then}\ \overrightarrow{\tau_o}(t) = \mathsf{nil}\ \mathsf{else}\ \exists v\ q.\ r \Mapsto (v, q) \in \chi \cup \gamma\} \end{array}$$

In other words, the value r returned by the action is a pointer. If $r = \mathsf{null}$ then the stack was empty at the time of the read, hence there exists a time-stamp t to indicate that. If r is not null, then at the time of the read, the stack was non-empty, and the read value is a node in the stack. Notice that we do not say that the obtained r is, or ever was, the sentinel, though of course, such a property does hold of the readSentinel action.

This indicates a somewhat surprising fact in the implementation of Treiber's stack, that the call to readSentinel, and the underlying physical read, are actually irrelevant for the *partial* correctness of Treiber's methods. The action could return a random pointer from the stack's or the garbage's heap, and we will still be able to derive the specification for the Treiber's methods. Of course, returning such a random pointer will affect liveness, as the implementation of push and pop methods will keep looping until, by chance, their calls to readSentinel pick and return a pointer that is actually at the top of the stack. But, our current specifications do not say anything about liveness.

The specification of readSentinel contains a subset of the conjuncts from the previous specifications; here too, these conjuncts are stable, for the same reasons as before. It is also not difficult to see that the specification is valid.

Indeed, if $r = \mathsf{null}$, then, from the definition of readSentinel action, it encountered an empty stack when invoked. At that very moment, the last entry in the history of the stack must be showing that the stack is empty, since by the resource invariant I, the last entry of the history reflects the actual physical state of the stack. Thus, a timestamp t exists when the stack was empty. Similarly, if r is not null, then the entry $r \mapsto (q, v)$, for some q and v must have been at the top of the stack. Thus, it is certainly in $\chi \cup \gamma$.

For readNode, we have:

$$\begin{aligned}\mathsf{readNode}(p) : [\tau, v, q].\ &\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \wedge p \Mapsto (v, q) \sqsubseteq \chi \cup \gamma\} \\ &\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \wedge r = (v, q) \wedge p \Mapsto (v, q) \sqsubseteq \chi \cup \gamma\}\end{aligned}$$

which says that if the contents of the node p was initially (v, q), then that's what's returned as the result. Moreover, p's contents remains unchanged.

We can now write out proof outline for the Treiber stack methods. First the push.

```
1  { σs = empty ∧ τs = empty ∧ τ ⊑ τo }
2  q <- alloc(v, null);
3  { σs = q ↦ (v, −) ∧ τs = empty ∧ τ ⊑ τo }
4  recfun loop() {
5  { σs = q ↦ (v, −) ∧ τs = empty ∧ τ ⊑ τo }
6  p <- readSentinel;
7  { σs = q ↦ (v, −) ∧ τs = empty ∧ ∃t. τ ⊑ τo^{≤t} ∧
8    if p = null then τo→(t) = nil else ∃v q. p ↦ (v, q) ∈ χ ∪ γ }
9  { σs = q ↦ (v, −) ∧ τs = empty ∧ τ ⊑ τo }
10 q := (v, p);
11 { σs = q ↦ (v, p) ∧ τs = empty ∧ τ ⊑ τo }
12 ok <- tryPush(p, q);
13 { if ok then σs = empty ∧ ∃t vs. τs = t ↦ (vs, v :: vs) ∧ τ ⊑ τo^{<t}
14   else σs = q ↦ (v, p) ∧ τs = empty ∧ τ ⊑ τo}
15 if ok then return ();
16 { σs = empty ∧ ∃t vs. τs = t ↦ (vs, v :: vs) ∧ τ ⊑ τo^{<t}}
17 else
18 { σs = q ↦ (v, −) ∧ τs = empty ∧ τ ⊑ τo }
19 loop();}();
20 { σs = empty ∧ ∃t vs. τs = t ↦ (vs, v :: vs) ∧ τ ⊑ τo^{<t}}
```

Line 1 is the precondition for push, and in Line 3 we have the postcondition for Allocate, except that, by the rule of consequence, we abstract away the fact that the tail pointer of q is null (i.e., we replace null by $-$). The derived assertion will be the precondition for the recursive function loop. The postcondition of loop is the same as the postcondition for push. That is, the type of loop is declared as follows, in the context where we have the variables q : ptr and $v : A$:

$$\begin{aligned}\mathsf{loop}() : [\tau].\ &\{\sigma_s = q \Mapsto (v, -) \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o\} \\ &\{\sigma_s = \mathsf{empty} \wedge \exists t\ vs.\ \tau_s = t \Mapsto (vs, v :: vs) \wedge \tau \sqsubseteq \tau_o^{<t}\}\end{aligned}$$

We next invoke readSentinel, but we frame the σ_s component of its precondition with respect to $q \Mapsto (v, -)$, to match the assertion we have in line 5. The lines 7-8 are the analogous framing of the postcondition for

readSentinel. But, it turns out that this postcondition is too strong for our purposes *here* (we will use the extra strength in the proof of pop). Thus, we immediately weaken the assertion in line 9, to set up the precondition for $q := (v, p)$ in line 10.

In Line 11, we have the postcondition for $q := (v, p)$, which sets up for the call to tryPush, and lines 13-14 give the postcondition of tryPush. If the result ok of tryPush is true, we terminate, with the postcondition of loop, which is also the desired postcondition for push. Otherwise, we obtain a precondition that enables us to call loop recursively.

Now the proof outline for pop.

```
1  { σ_s = empty ∧ τ_s = empty ∧ τ ⊑ τ_o }
2  p <- readSentinel;
3  { σ_s = empty ∧ τ_s = empty ∧ ∃t. τ ⊑ τ_o^{≤t} ∧
4    if p = null then τ⃗_o(t) = nil else ∃v q. p ↦ (v, q) ∈ χ ∪ γ }
5  if p == null then return None
6  { r = None ∧ σ_s = empty ∧ τ_s = empty ∧ ∃t. τ ⊑ τ_o^{≤t} ∧ τ⃗_o(t) = nil }
7  else
8  { σ_s = empty ∧ τ_s = empty ∧ τ ⊑ τ_o ∧ ∃v q. p ↦ (v, q) ∈ χ ∪ γ }
9  (v, q) <- readNode p;
10 { σ_s = empty ∧ τ_s = empty ∧ τ ⊑ τ_o ∧ p ↦ (v, q) ∈ χ ∪ γ }
11 ok <- tryPop (p, q);
12 { σ_s = empty ∧ if ok then ∃t vs. τ_s = t ↦ (v :: vs, vs) ∧ τ ⊑ τ_o^{<t}
13                 else τ_s = empty ∧ τ ⊑ τ_o }
14 if ok then return Some v
15 { σ_s = empty ∧ ∃t vs. r = Some v ∧ τ_s = t ↦ (v :: vs, vs) ∧ τ ⊑ τ_o^{<t} }
16 { σ_s = empty ∧ ∃t v vs. r = Some v ∧ τ_s = t ↦ (v :: vs, vs) ∧ τ ⊑ τ_o^{<t} }
17 else
18 { σ_s = empty ∧ τ_s = empty ∧ τ ⊑ τ_o }
19 pop() ;
20 { the postcondition for pop }
```

The proof outline is completely straightforward, as a postcondition of every command is directly the precondition for the next. Similarly, the postconditions for each terminating return produces an appropriate component of the ultimately desired postcondition for pop.

3.6 New rules

We require a few changes to the coarse-grained logic from the previous chapter.

First, instead of types of the form $\{P\}\ A\ \{Q\}@\Gamma$, where Γ was a context of resources, now we have

$$\{P\}\ A\ \{Q\}@C$$

where C is a state transition system of the form we described. That is, its states contain a_s, a_j and a_o components, a resource invariant I describes the state space, and transitions from a set T determine the atomic steps that the programs of the above type are allowed to make. The program will only every be composed in parallel with programs making same transitions; thus, when preempted, we know that the shared and other state can be changed only by taking arbitrarily many *transposed* transitions from C.

Second, the inference rules for atomic primitives such as lookup and mutation, are omitted from the logic per-se, as the atomics are now specified by the user. We do have a rule, however, which assigns a Hoare triple to a user-specified atomic action.

As we described, an atomic action α is semantically a relation between the input state, output state, and output result r. The input and output states are drawn from the state space I. Thus, when one wants to prove that a certain atomic action α can be ascribed a precondition P and a postcondition Q, one essentially has to prove an inclusion of relations: that the precondition P is included in the domain of α, and that the

range of α is included in Q. As discussed previously, we need to show that P and Q are stable, as per the semantic meaning of Hoare triples.

$$\frac{\vdash \forall s_1.\, P\ s_1 \wedge I\ s_1 \implies s_1 \in \mathsf{dom}(\alpha) \wedge \forall r\ s_2.\, \alpha\ s_1\ s_2\ r \implies Q\ r\ s_2 \quad P, Q \text{ stable wrt. } C}{\vdash \alpha : \{P\}\ A\ \{Q\}@C}$$

In the above rule, we denote by $\mathsf{dom}(\alpha)$, and call it the domain of the relation α, the following:

$$\mathsf{dom}(\alpha) = \{s \mid \exists s'\ r.\, (s, s', r) \in \alpha\}$$

The similar side conditions on stability show up in two other rules as well, namely the rule for monadic unit, and the framing rule, which now look as follows.

$$\frac{\vdash e : A \qquad P \text{ is stable wrt. } C}{\vdash \mathsf{return}\ e : \{P\}A\{r = e \wedge P\}@C}$$

$$\frac{c : \{P\}A\{Q\}@C \qquad R \text{ is stable wrt. } C}{c : \{P \circledast R\}A\{Q \circledast R\}@C}$$

In the coarse-grained setting, we had a very specific definition of stability, which was, for example, baked into the rule for return by means of insisting that $\mu_s = \mathsf{Own} \implies \alpha_o = a$. But, in the fine-grained setting, we have to prove the stability with respect to an arbitrary resource C.

Finally, we have a rule of consequence which allows us to assume the resource invariant I whenever we need it in the preconditions and postconditions, even if I was not explicitly provided.

$$\frac{\vdash c : \{P'\}A\{Q'\}C \qquad \forall s.\, P\ s \wedge I\ s \implies P'\ s \qquad \forall r\ s.\, Q'\ r\ s \wedge I\ s \implies Q\ r\ s}{\vdash c : \{P\}A\{Q\}}$$

Note 3.6.1 (Other rules, not discussed here).

In the FCSL papers, we have introduced two other inference rules.

The first rule, takes a command c verified under resource C, and extends the verification to a new resource obtained by composing C with another resource C'. Thus, such a rule generalizes to the fine-grained case the concept of context weakening from CSL, where instead of C we had a context of resources Γ. This precisely corresponds to extending the specifications of allocate and $:=$ to use not only σ_s, but also histories τ_s and τ_o.

The second rule, and the associated language primitive hide, adds a new resource C' to the existing resource C, runs a given command c, and then deallocates C'. Thus, it generalizes to the fine-grained case the construction resource l in c from CSL.

We do not consider these two constructions in the current notes, as they are still in a somewhat preliminary form.

We close the chapter with a discussion on the conditions that resource invariants, transitions, and atomic primitives have to satisfy. So far, we have treated them as arbitrary relations, but they are not arbitrary. In order to be able to prove sound the various inference rules of the system, these components have to satisfy numerous conditions. Whenever we define a resource invariant, or a transition, or an atomic action, these conditions have to be discharged (in the example for Treiber, we did not show proofs that these conditions hold, because they are a bit tedious, but, we did encode such proofs in Coq). The conditions are as follows. We mention here that the exact formulations of the definitions of transitions and actions is an ongoing research effort, and we expect that in the future this set of conditions will change (and hopefully, significantly reduce in size).

In the notation below, we use s and variants to range over states, and ρ and variants to range over transitions. We also assume that always $s = (a_s, a_j, a_o)$, and similarly for the variants, e.g. $s_1 = (a_{s1}, a_{j1}, a_{o1})$, and $s_2 = (a_{s2}, a_{j2}, a_{o2})$.

Conditions for invariants.

1. Subjective validity.

$$\forall s \in I.\, \mathsf{valid}(a_s \bullet a_o)$$

2. Heap validity. Denoting $\bar{s}$ the heap obtained after erasing auxiliary state from s:

$$\forall s \in I.\, \mathsf{valid}(\bar{s})$$

3. Type determinacy.
   If $s_1, s_2 \in I$, then a_{s1}, a_{s2}, a_{o1}, a_{o2} are drawn from the same PCM, and a_{j1}, a_{j2} are drawn from the same type.
4. Subjective stability.

$$(a_s, a_j, a_o \bullet x) \in I \iff (a_s \bullet x, a_j, a_o) \in I$$

Of the above conditions, the last one is the most significant. It says that forking and joining do not take the state of the resource, out of the state space, if it wasn't already out. Or, in other words, I cannot restrict the a_s and a_o components separately, but only their join $a_s \bullet a_o$. This is why in the coarse-grained setting, we let the invariant be a predicate over α that stood for the combination $a_s \bullet a_o$ when interpreted by the Owicki-Gries-O'Hearn protocol. Here, we parametrize the invariant over both a_s and a_o, but then, by the condition on subjective stability, insist that the invariant behaves as if it were parametrized by their join. Of course, we introduce the parametrization by both a_s and a_o because we need both for components when describing the transitions (e.g., the transition *push* increases τ_s, but also needs τ_o in order to compute the next unused timestamp).

Conditions for transitions.

1. Preservation of state invariants.

$$(s_1, s_2) \in \rho \implies s_1, s_2 \in I$$

2. Transitions cannot change the other.

$$(s_1, s_2) \in \rho \implies a_{o1} = a_{o2}$$

3. Locality.
   If $((a_{s1}, a_{j1}, a_{o1} \bullet x), (a_{s2}, a_{j2}, a_{o2} \bullet x)) \in \rho$, then $((a_{s1} \bullet x, a_{j1}, a_{o1}), (a_{s2} \bullet x, a_{j2}, a_{o2}) \in \rho$.

The first two conditions are obvious, but we comment on the last on. We give it a name *Locality*, because it essentially lifts to transitions ρ (hence, relations), the concept that we have already seen in the coarse-grained setting, where we applied it to auxiliary functions Φ. How so?

Well, let us imagine that ρ is actually a function, so it produces a unique triple (a_{s2}, a_{j2}, a_{o2}) for a given input (a_{s1}, a_{j1}, a_{o1}). Moreover, assume that ρ only operates on the first projection, and ignores the others, as this is what the auxiliary function Φ did in the previous chapter. Thus, we can write $\rho(a_{s1}) = a_{s2}$ instead of the more detailed $((a_{s1}, a_{j1}, a_{o1}), (a_{s2}, a_{j2}, a_{o2})) \in \rho$.

Then, the premise of the locality condition says exactly $a_{s2} = \rho(a_{s1})$, and the conclusion says that $\rho(a_{s1} \bullet x) = a_{s2} \bullet x = \rho(a_{s1}) \bullet x$, as in the functional definition of locality, taking $\alpha = a_{s2}$ and $\beta = x$.

The conditions that $\mathsf{valid}(a_{s2} \bullet x)$ and $\mathsf{valid}(\rho(a_{s1})$ that correspond to $\mathsf{valid}(\alpha \bullet \beta)$ and $\mathsf{valid}(\Phi(\alpha))$ from the functional definition are subsumed by the other conditions on invariants and transitions. Indeed, we know that $\mathsf{valid}(a_{s2} \bullet x)$ hold, because by preservation of state invariants $(a_{s2}, a_{j2}, a_{o2} \bullet x) \in I$, and thus by subjective validity $\mathsf{valid}(a_{s2} \bullet a_{o2} \bullet x)$. But then, it must also be $\mathsf{valid}(a_{s2} \bullet x)$, by preservation of validity. Similarly, we know that $\mathsf{valid}(\rho(a_{s1}))$ holds, because this equals $\mathsf{valid}(a_{s2})$, and this also follows from the above, by preservation of validity.

The significant of the locality transition is thus similar with the significance of the locality condition on functions. It is essentially important for the soundness of the framing rule, and more generally, parallel

composition, because it says that if we prove a program under the assumption that the program takes a transition over a certain state a_{s1}, then the same transition can be taken over a larger state.

Note 3.6.2 (One omitted condition).
In FCSL, there is one more condition, for footprint preservation, but we omit it here as it only makes sense in the presence of the infrastructure for composing resources, that we omitted as well.

Conditions for primitive actions.

1. Preservation of invariants.
   If $s \in \mathsf{dom}(\alpha)$ then $s \in I$.
2. Safety monotonicity.
   If $(a_s, a_j, a_o \bullet x) \in \mathsf{dom}(\alpha)$, then $(a_s \bullet x, a_j, a_o) \in \mathsf{dom}(\alpha)$.
3. Actions choose between transitions.
   If $(s_1, s_2, r) \in \alpha$ then there exists $\rho \in T$, such that $(s_1, s_2) \in \rho$.
4. Frameability.
   If $(a_s, a_j, a_o \bullet x) \in \mathsf{dom}(\alpha)$ and $((a_s \bullet x, a_j, a_o), s', r) \in \alpha$ then exists a'_s, a'_j and a'_o, such that $s' = (a'_s \bullet x, a'_j, a'_o)$, and $((a_s, a_j, a_o \bullet x), (a'_s, a'_j, a'_o \bullet x), r) \in \alpha$.
5. Functionality.
   If $s \in \mathsf{dom}(\alpha)$, then there exist unique s', r such that $(s, s', r) \in \alpha$.
6. Eraseability.
   If an action is executed twice in states s_1 and s_2 that erase to the same heap, then the two executions produce the same values, and same erasures of ending states.

   $$\forall s\ h\ t\ r\ s'\ h'\ t'\ r'.\ \mathsf{valid}(\overline{s} \cup h) \wedge \overline{s} \cup h = \overline{s'} \cup h' \wedge (s, t, v) \in \alpha \wedge (s', t', v') \in \alpha \implies v = v' \wedge \overline{t} \cup h = \overline{t'} \cup h'$$

The most interesting properties here are Safety monotonicity and Frameability. We explain them here, temporarily ignoring a_j, a_o and r in the explanation.

Safety monotonicity says the following. If we can run an action in a smaller a_s (i.e., the action has an output, given a_s), then we can run the action in an enlarged a_s as well. A similar property has been first defined for Abstract Separation Logic [4], but here we extend it with a_j and a_o, and the fact that when x is framed onto a_s, then it has to be taken out of a_o. There, the property was important for proving the soundness of the frame rule

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

Indeed, as discussed before, Hoare triples in separation logic imply fault avoidance. Thus, proving the frame rule sound will have to show that c is sound in a larger state satisfying $P * R$, if we are given, by the premiss, that it is sound in a smaller state satisfying P.

Frameability says if the action has been executed in a larger state $a_s \bullet x$, but is already safe for a smaller a_s, then we could run it in the smaller state, and the result a'_s is obtained by removing the frame x from the result obtained from running in the larger state. A similar property has also been first defined for Abstract Separation Logic [4], and is also essential for proving the soundness of the frame rule.

Indeed, let us assume that c is our atomic action, and that we executed it in a larger state satisfying $P * R$. We want to show that $Q * R$ holds of the result state. We also know that c is safe in a smaller state satisfying P, by the premiss. Then, frameability tells us that the ending state can be divided into two parts, the first satisfying Q by the premise, and the second being unchanged, thus satisfying R. Thus, the ending state in totality satisfies $Q * R$.

Notice that Frameability is in a sense inverse to Locality for transitions: where locality says that if we can move from running in smaller state to running in larger and obtained equal results (subject to adjusting for the difference in size), frameability says that we can move from running in a larger state, to running in a smaller, subject to the condition that the smaller state contains enough resources for us to run.

Exercise 3.6.1. Specify and verify the snapshot algorithm given bellow.

$$\begin{aligned}\mathsf{snapshot}() : A * A = {} & (vx, tx_1) \leftarrow !x;\\ & vy \leftarrow !y;\\ & (\_, tx_2) \leftarrow !x;\\ & \mathsf{if}\ tx_1 == tx_2\ \mathsf{then\ return}\ (vx, vy)\ \mathsf{else}\ \mathsf{snapshot}()\end{aligned}$$

Snapshot operates over the data structure consisting of two pointers x and y, which can be independently changed by concurrent threads. Snapshot returns the values of x and y, but makes sure that the returned values actually resided in the memory *together*, and have not been changed by interfering threads in the middle of the snapshoting. Snapshot recognizes such situations by keeping a timestamp with x, which is incremented upon every modification of x. It then sandwiches the read of y in between two reads of x. If the two obtained timestamps of x are equal, the read values of x and y resided in memory together, and can be returned.

Chapter 4

Non-Linearizable Structures and Structures with Sharing

We next explore the verification of several challenging programs in the formalism we introduced in the previous chapter. This will illustrate the salient points of the method, on examples that have been difficult to formally consider in the previous work.

4.1 Exchanger

The first example is the exchanger program, which we present further below. This program is interesting because it is *non-linearizable*. Traditionally, the work on concurrent algorithms have typically focused only on linearizable programs, and the non-linearizable ones were not considered useful, but the exchanger is one with practical use, witnessed by its inclusion into the `java.util.concurrent` library [7, 30].

In order to specify and verify this program, recent related work [9, 10] has thus proceeded to generalize linearizability to a new correctness criteria of Concurrency-Aware Linearizability (CAL), tailor-made for this example. We will show how to specify and verify this example, without recourse to generalizations of linearizability, using only the variant of separation logic that we introduced so far.

We next present the exchanger program, and discuss the challenges that it presents.

```
1  exchange (v : A) : option A =
2      p ← alloc v U;
3      b ← CAS (g, null, p);
4      if b == null then
5          sleep (50);
6          x ← CAS (p+1, U, R);
7          if x is M w then return (Some w)
8          else return None
9      else
10         dealloc p;
11         cur ← read g;
12         if cur == null then return None
13         else
14             x ← CAS (cur+1, U, M v);
15             CAS (g, cur, null);
16             if x is U then
17                 w ← read cur;
18                 return (Some w)
19             else return None
```

The main goal of the exchanger program, and of the associated data structure is, as the name suggests,

to exchange values. More precisely, exchange takes an input value $v : A$ and tries to find another concurrent instance of exchange, say one which took an input value $w : A$. The two instances will then proceed to communicate in order to exchange their respective values. If successful, the first instance will return Some w as its result, and the second instance will return Some v as its result.

The exchanger programs operate over a shared data structure which provides the book-keeping necessary for the exchanges to be advertised and then take place.

In particular, the main component of the shared state consists of a global pointer g which points to the last exchange offer made. An offer consists of two consecutive locations, storing v:A and vh : hole A, respectively. Here hole is defined as:

$$\mathsf{hole}\ A = \mathsf{U} \mid \mathsf{R} \mid \mathsf{M}\ \mathsf{of}\ A$$

The v component of the offer (stored in p) says that it is the value v that the offer advertised for the exchange. The vh component (stored in $p + 1$) describes the status of the offer.

1. If $vh = \mathsf{U}$, then the offer of v is *unmatched*, that is, no concurrent thread has agreed to exchange with it. Every offer initially starts unmatched.
2. If $vh = \mathsf{R}$, then v is *retired*. The offering thread waited for a match, but none came, thus the thread retires its offer, and returns None to signal that no matching was achieved.
3. If $vh = \mathsf{M}\ w$, then v was *matched* with w by some other thread.

Exchanger operates on the data structure as follows. It first allocates a new offer p in its private heap (line 2). Then it tries to offer p to other threads, by writing p into g (line 3). Since g is the pointer to the latest offer, and is global, i.e., known to every thread, this makes p "public"; other threads can notice p, and potentially match it. The en-linking is done via a "compare-and-swap" version of CAS. That is, the CAS here is a version that returns the read value, rather than a version that returns a boolean that we used in Treiber's example. This version of CAS will streamline the presentation in the situations when we are CAS-ing over values of type hole. Since hole has three different kinds of values (U, R or M w), returning the value is more informative than returning a boolean, as it will also tell us *why* the CAS failed, if it did.

If CAS succeeds, exchanger waits a bit (line 5, which we will turn into a skip when we carry out our proofs), then checks if the offer has been matched by some w (lines 6, 7). If so, Some w is returned (line 7). Otherwise, the offer is retired by storing R into its hole (line 6). Retired offers remain allocated, just like popped nodes remain allocated in Treiber's stack, in order to avoid the ABA problem [11, 35].

If CAS fails to en-link p into g in line 3, the exchanger deallocates the offer p (line 10), and instead tries to match the offer *cur* that is current in g. If no offer is current, perhaps because another thread managed to match the offer that made the CAS in line 3 fail, the exchanger returns None (line 12). Otherwise, the exchanger tries to make a match, by changing the hole of *cur* into M v (line 14). If successful (line 16), it reads the value w stored in *cur* that was initially proposed for matching, and returns it. In any case, it unlinks *cur* from g (line 15) to make space for other offers.

What is challenging about specifying the exchanger? If one wants to use linearizability, which is the standard correctness criterion for concurrent data structures, the problem is in the following. As described in the previous chapter, linearizability requires showing that an execution of a number of concurrent threads can be sequentialized. But, with exchange, what we need is not sequentiality, but the ability to say that some threads—namely those that participated in an exchange—executed *simultaneously*. In other words, values were exchanged atomically, without interference from other threads. Linearizability was simply not designed with such a property in mind.

On the other hand, in our setting of Hoare logic with subjective auxiliary state, this will be a very simple property to state. In particular, we will use subjective auxiliary state of histories to record the value that each thread exchanged, together with the timestamp at which the exchange occurred. We will engineer the manipulation of auxiliary state so that the two ends of an exchange appear at consecutive timestamps t and $t + 1$. This can be *logically interpreted* as simultaneous execution of two halves of the same events, since no other event can have a timestamp in between t and $t+1$ (timestamps are natural numbers). For example, the timestamps $1 \Mapsto (v_1, w_1)$ and $2 \Mapsto (w_1, v_1)$ will encode the end-points of the first exchange; the timestamps $3 \Mapsto (v_2, w_2)$ and $4 \Mapsto (w_2, v_2)$ will encode the end-points of the second exchange, etc., the timestamps t and $t + 1$, for odd t, will encode the end-points of the $\frac{t+1}{2}$-th exchange. We will easily impose this constraint on histories by requiring it in the resource invariant of the exchange resource. However, when working with

linearizability, there is no obvious way to impose the constraint, hence the need for a different correctness criterion such as CAL.

4.1.1 Auxiliary state and exchanger specification

In order to engineer the auxiliary state so that the events corresponding to the two ends of an exchange appear with consecutive timestamps, we require the following components, which will include heaps and histories, but also a helper component which tracks the offers created by a thread. Having a witness of ownership for an offer in its private auxiliary state, will give the thread a permission to manipulate that offer (i.e., retire it, or collect the value with which the offer is matched).

In particular, we have the following: (1) thread-private heap σ_s of the thread, and of the environment σ_o, (2) a time-stamped history of values τ_s that the thread exchanged so far, and dually τ_o for the environment, and (3) the set of outstanding offers π_s created by the thread, and by the environment π_o. These are thread-private components (i.e., projections from a_s or a_o). We also need the following joint components (projections from a_j): the heap σ_j of the structure, and a map μ of "matched offers". The heap σ_j is further subdivided to contain the pointer g, whose contents we name γ, and the rest of the heap, which we name χ.

The idea is that we will decorate the exchanger with auxiliary code, so that it manipulates the above auxiliary state as follows.

- First, σ_j is a heap that serves as the "staging" area for the offers. It includes the global pointer g. Whenever a thread wants to make an offer, it allocates a pointer p in σ_s, and then tries to move p from σ_s into σ_j, simultaneously linking g to p. This is very similar to the situation we had with Treiber's stack.
- Second, π_s and π_o are sets of offers (hence, sets of pointers) that determine offer ownership. A thread that has the offer $p \in \pi_s$ is the one that created it, and thus has the *sole* right to retire p, or to collect the value that p was matched with. Upon collection or retirement, p is removed from π_s.
- Third, τ_s and τ_o are histories, each mapping a time-stamp to a pair of exchanged values. A singleton history $t \Mapsto (v, w)$ symbolizes that a thread having this singleton as a sub-component of τ_s, has exchanged v for w at time t. As mentioned above, the most important invariant of the exchanger is that each such singleton is matched by a "symmetric" one to capture the other end of the exchange. We emphasize that if τ_s contains a singleton for one end of the exchange, then it does not follow that the singleton for the other end must be in τ_o. For example, in the parallel composition

  $$\mathsf{exchange}(v) \parallel \mathsf{exchange}(w)$$

  each child subthread will hold one end of the exchange in its τ_s, and the other in τ_o, but the parent thread will hold both ends in τ_s.
- Fourth, μ is a map storing the offers that were matched, but not yet collected. Thus, $\mathsf{dom}\ \mu = \pi_s \cup \pi_o$. A singleton entry in μ has the form $p \Mapsto (t, v, w)$ and denotes that offer p, initially storing v, was matched at time t with w. A singleton entry is entered into μ when a thread on the one end of matching, matches v with w. Such a thread also places the *twin* entry $\bar{t} \Mapsto (w, v)$, with inverted order of v and w, into its own private history τ_s, where:

  $$\bar{t} = \begin{cases} t+1 & \text{if } t \text{ is odd} \\ t-1 & \text{if } t > 0 \text{ and } t \text{ is even} \end{cases}$$

  For technical reasons, 0 is not a valid time-stamp, and has no distinct twin. The technical reason goes back to the definition of histories in Treiber's stack. There, we used the timestamp 0 to encode the initial values for the stack. In the case of the exchanger, we do not require such an initial value, so 0 is unused.

  The pending entry for p resides in μ until the thread that created the offer p decides to "collect" it. It removes p from μ, and simultaneously adds the entry $t \Mapsto (v, w)$ into its own τ_s, thereby logically completing the exchange. Since twin time-stamps are consecutive integers, a history cannot contain entries *between* twins. Thus, two twin entries in the combined history including τ_s, τ_o and μ, jointly represent a single exchange, as if it occurred *atomically*. More formally, consider

  $$T = \tau_s \cup \tau_o \cup \|\mu\|.$$

Then, the exchanger's main invariant is that T always contains matching twin entries:

$$t \Mapsto (v, w) \sqsubseteq T \iff \bar{t} \Mapsto (w, v) \sqsubseteq T \tag{1}$$

Here $\|\mu\|$ is the collection of all the entries in μ. That is,

$$\begin{array}{rcl} \|\mathsf{empty}\| & = & \mathsf{empty} \\ \|p \Mapsto (t, v, w) \cup \mu'\| & = & t \Mapsto (v, w) \cup \|\mu'\| \end{array}$$

With these components introduced and informally explained, we can now present the desired Hoare spec for exchange.

$$\begin{array}{l} \mathsf{exchange}(v) : [\tau]. \ \{\sigma_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\|\} \\ \qquad\qquad \{\sigma_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \\ \qquad\qquad\ \mathsf{if}\ r\ \mathsf{is\ Some}\ w\ \mathsf{then}\ \exists t.\, \tau_s = t \Mapsto (v, w) \wedge \mathsf{last}(\tau) < t, \bar{t}\ \mathsf{else}\ \tau_s = \mathsf{empty}\} \end{array} \tag{2}$$

The precondition says that the exchanger starts with the empty private heap σ_s, empty set of offers π_s and empty history τ_s; hence by framing, it can start with any value for these components. The logical variable τ names the initial history of other threads, which, in this case, includes the finished, as well as pending events, i.e.:

$$\tau \sqsubseteq \tau_o \cup \|\mu\|.$$

In the postcondition, the self heap σ_s and the set of offers π_s didn't change. Hence, if `exchange` made an offer during its execution, it also collected or retired it by the end. The history τ is still a subset of the ending value for $\tau_o \cup \|\mu\|$, signifying that the environment history only grows by interference.

If the exchange fails (*i.e.*, r is None), then τ_s remains empty. If it succeeds (either in line 7 or in line 16 in the code for exchanger), i.e., if the result r is Some w, then there exists a time-stamp t, such that self-history τ_s contains the entry $t \Mapsto (v, w)$, symbolizing that v and w were exchanged at time t.

Importantly, the postcondition implies, by invariant (1), that in the success case, the twin entry $\bar{t} \Mapsto (w, v)$ must belong to $\tau_o \cup \|\mu\|$, *i.e.*, *another* thread matched the exchange. Moreover, the exchange occurred *after* the call to exchange: whichever τ we chose in the pre-state, both t and $\bar{t}$ are larger than the last time-stamp in τ.

4.1.2 Resource invariant, transitions, atomics

Resource invariant. The states in the exchanger state-space must satisfy other invariants in addition to (1). These properties arise from our description of how the exchanger behaves on decorated state. As we introduced in Section 1, we abbreviate with $p \Mapsto (x; y)$ the heap $p \Mapsto x \cup p{+}1 \Mapsto y$.

We list the properties (on top of 1) that make up the resource invariants in an itemized list below.

(i) σ_j contains a pointer g (storing γ), with a number of offers $p \Mapsto (v; x)$, and g points to either null or to some offer in σ_j.

$$(\sigma_j = g \Mapsto \gamma \cup \chi) \wedge (\gamma \in \{\mathsf{null}\} \cup \mathsf{dom}(\chi)) \wedge \forall p \in \mathsf{dom}(\chi).\, \exists v : A.\, w : \mathsf{hole}\ A.\, h : \mathsf{heap}.\, \chi = p \Mapsto (v; w) \cup h$$

(ii) τ_s, τ_o and $\|\mu\|$ contain only disjoint time-stamps. Similarly, π_s is disjoint from π_o.

$$\mathsf{valid}(\tau_s \cup \tau_o \cup \|\mu\|) \wedge \mathsf{valid}(\pi_s \cup \pi_o)$$

(iii) All offers in μ are matched, and are always owned by some thread:

$$(\exists t.\, p \Mapsto (t, v, w) \sqsubseteq \mu) \iff p \in \pi_s \cup \pi_o \wedge p \Mapsto (v; \mathsf{M}\ w) \sqsubseteq \chi.$$

(iv) There is at most one unmatched offer; it is the one linked from g. It is owned by someone.

$$p \Mapsto (v; \mathsf{U}) \sqsubseteq \chi \implies \gamma = p \wedge p \in \pi_s \cup \pi_o.$$

(v) Retired offers aren't owned by anyone.

$$p \Mapsto (v; \mathsf{R}) \sqsubseteq \chi \implies p \notin \pi_s \cup \pi_o.$$

(vi) The outstanding offers are included in the joint heap.

$$p \in \pi_s \cup \pi_o \implies p \in \mathsf{dom}(\chi).$$

(vii) The combined history $T = \tau_s \cup \tau_o \cup \|\mu\|$ is gap-less: if it contains a time-stamp t, it also contains all the smaller time-stamps (sans 0).

$$\forall t.\, t \leqslant \mathsf{last}(T) \implies t > 0 \implies t \in \mathsf{dom}(T).$$

This last requirement is similar to what we had for Treiber, although, notice that we do not require continuity on histories in the case of the exchanger. This illustrates that different examples may require different notions of histories, and there is no one uniform definition or property (beyond PCM-ness), that should be adopted for all situations.

Transitions. We require the following transitions, which bound the modifications on the auxiliary state, and formalize the intuition on what auxiliary state does that we outlined before.

- *install*(p) takes a pointer p, and moves the (unmatched) offer represented as $p \Mapsto (v; \mathsf{U})$, from the private heap, into shared χ, thus installing the offer into the data structure. The transition requires that no offer is current (i.e., $\gamma = \mathsf{null}$). It adds p to π_s to record the ownership of p by the current thread, and sets p as the current offer ($\gamma = p$). More succinctly:

$$\begin{aligned} \mathit{install}(p) = [v, h_1, \pi, h_2].\, (\sigma_s = p \Mapsto (v; \mathsf{U}) \cup h_1 \wedge \pi_s = \pi \wedge \gamma = \mathsf{null} \wedge \chi = h_2) \rightsquigarrow \\ (\sigma_s = h_1 \wedge \pi_s = \pi \cup \{p\} \wedge \gamma = p \wedge \chi = p \Mapsto (v; \mathsf{U}) \cup h_2) \end{aligned}$$

- *match*(p, w) encodes matching of the offer p (and storing v) with w. It changes the hole of the offer from U to $\mathsf{M}\ w$, it adds the item $t \Mapsto (w, v)$ to the self history, and it also pushes onto μ an entry an item of the form $p \Mapsto (t+1, v, w)$. This is a pending entry that will be eventually collected by the thread that proposed v for matching, and moved into *that thread's* self history, as an event at timestamp $t+1$.

$$\begin{aligned} \mathit{match}(p, w) = [v, h, \tau, m].\, (\chi = p \Mapsto (v; \mathsf{U}) \cup h \wedge \tau_s = \tau \wedge \mu = m) \rightsquigarrow \\ (\chi = p \Mapsto (v; \mathsf{M}\ w) \cup h \wedge \tau_s = \tau \cup t \Mapsto (w, v) \wedge \mu = m \cup p \Mapsto (t+1, v, w)) \end{aligned}$$

where t is the smallest unused timestamp in the global history, i.e., $t = \mathsf{fresh}(\tau_s \cup \tau_o \cup \|\mu\|) = \mathsf{last}(\tau_s \cup \tau_o \cup \|\mu\|) + 1$.
Notice that if the hole of p stores U, by the property (iv) above, it must be $\gamma = p$, i.e., p is the current offer linked to by g. Also, t must be an odd number, because the size of the global history $\tau_s \cup \tau_o \cup \|\mu\|$ is even, by property (1).

- *retire*(p) takes an unmatched offer of the form $p \Mapsto (v; \mathsf{U})$ and changes the hole to R. The transition can only be taken by the thread owning p, i.e., for which $p \in \pi_s$.

$$\begin{aligned} \mathit{retire}(p) = [v, h, \pi].\, (\chi = p \Mapsto (v; \mathsf{U}) \cup h \wedge \pi_s = \{p\} \cup \pi) \rightsquigarrow \\ (\chi = p \Mapsto (v; \mathsf{R}) \cup h \wedge \pi_s = \pi) \end{aligned}$$

- *collect*(p, w) is carried out by a thread whose offer has been matched. It entails picking out an entry of the form $p \Mapsto (t, v, w)$ from the list of pending offers, and moving it to self history. The transition can only be taken by a thread that offered p. Thus, an entry left in μ by a thread which achieved the matching, is collected and moved to the self history of a thread that originally made the offer.

$$\begin{aligned} \mathit{collect}(p, w) = [t, v, m, \pi, \tau].\, (\mu = p \Mapsto (t, v, w) \cup m \wedge \pi_s = \{p\} \cup \pi \wedge \tau_s = \tau) \rightsquigarrow \\ (\mu = m \wedge \pi_s = \pi \wedge \tau_s = \tau \cup t \Mapsto (v, w)) \end{aligned}$$

If this transition applies to an offer p, then, by property (iii) above, one can show that $p \Mapsto (v, \mathsf{M}\ w) \sqsubseteq \chi$.

- *unlink*(p) unlinks the offer p which is required to be the current offer (i.e., the one en-linked by g), and is required to be matched or retired. Unmatched offers can't be unlinked, so that we preserve the property (iv).

$$\begin{aligned} \mathit{unlink}(p) = [v, x, h].\, (\gamma = p \wedge \chi = p \Mapsto (v; x) \cup h \wedge x \neq \mathsf{U}) \rightsquigarrow \\ (\gamma = \mathsf{null} \wedge \chi = p \Mapsto (v; x) \cup h) \end{aligned}$$

Atomic actions. The atomic actions are the following.

1. $\mathsf{try\_install}(p)$ annotates with auxiliary code the $\mathsf{CAS}(g, \mathsf{null}, p)$ command from line 3. If the CAS succeeds, the $install(p)$ transition is taken. If the CAS fails, the $idle$ transition is taken, but we know that the contents of g is non-null.

$$\mathsf{try\_install}(p) = \mathsf{if}\ r = \mathsf{null}\ \mathsf{then}\ install(p)\ \mathsf{else}\ \gamma = r \rightsquigarrow \gamma = r$$

2. $\mathsf{retire\_collect}(p)$ annotates with auxiliary code the $\mathsf{CAS}(p+1, \mathsf{U}, \mathsf{R})$ command from line 6. It looks into the hole of p, and if it finds U, it takes the $retire(p)$ transition. If it finds $\mathsf{M}\ \ w$, it takes the $collect(w)$ transition. If it finds R, nothing happens.

$$\begin{aligned}\mathsf{retire\_collect}(p) = \mathsf{match}\ r\ \mathsf{with}\\ \mathsf{U} &\Rightarrow retire(p)\\ \mid \mathsf{M}\ w &\Rightarrow collect(p, w)\\ \mid \mathsf{R} &\Rightarrow idle\ (\exists v.\, p \Mapsto (v; \mathsf{R}) \in \chi)\end{aligned}$$

   We will only run this action in a situation when we have the ownership of p; hence, the third case will never appear, as retired offers are not owned, by the resource invariant. Nevertheless, we specify that in the third case corresponds to the input heap in which the hole of p is R.
   It is not difficult to see that the above action erases to $\mathsf{CAS}(p+1, \mathsf{U}, \mathsf{R})$. Indeed, if $r = \mathsf{U}$, then the appropriate transition is $retire(p)$, which changes $p+1$ from U to R, and all the other of its changes are on the auxiliary parts of the state. If $r = \mathsf{M}\ w$, then the transition is $\mathsf{collect}(p, w)$, which applies only when $p+1$ stores $\mathsf{M}\ w$, and then only modifies auxiliary state, but remains identity on heaps. Finally, if $r = \mathsf{R}$, then the specified transition applies only when $p+1$ stores R, but, again, nothing is changed in the heap or the auxiliary state.
3. $\mathsf{try\_match}(cur, v)$, annotates with auxiliary code the $\mathsf{CAS}(cur+1, \mathsf{U}, \mathsf{M}\ v)$ command from line 14. Cur is the offer we're trying to match, and v is the value to match it with. The commands looks into the hole of cur, and if it finds U, it takes the $match(cur, v)$ transition. Otherwise, it takes the idle transition, but we know that the hole of cur stores r.

$$\begin{aligned}\mathsf{try\_match}(cur, v) = \mathsf{if}\ r = \mathsf{U}\ \mathsf{then}\ match(cur, v)\\ \mathsf{else}\ idle\ (\exists w.\, cur \Mapsto (w; r) \sqsubseteq \chi)\end{aligned}$$

4. $\mathsf{unlink\_offer}(cur)$ annotates with auxiliary code the $\mathsf{CAS}(g, cur, \mathsf{null})$ command from line 15. It invokes the $unlink(cur)$ transition and returns $r = cur$, or it invokes the idle transition, with $r \neq cur$ being the contents of g (hence, invoked by the label γ).

$$\mathsf{unlink\_offer}(cur) = \mathsf{if}\ r = cur\ \mathsf{then}\ unlink(cur)\ \mathsf{else}\ idle\ (\gamma = r)$$

   We will only invoke this action when we know that the offer cur is either matched or retired. The result will be unlinking cur from g, if cur was not already unlinked by some other thread.
5. $\mathsf{read\_current}$ annotates with auxiliary code the $\mathsf{read}\ g$ command from line 11. It reads the pointer stored in g (i.e., γ in terms of auxiliary state), and corresponds to taking the idle transition.

$$\mathsf{read\_current} = idle\ (\gamma = r)$$

6. $\mathsf{read\_node}(cur)$ annotates with auxiliary code the $\mathsf{read}\ cur$ command from line 16. It just takes an idle transition, but we need to know that cur is an offer in the heap.

$$\mathsf{read\_node}(cur) = idle\ (\exists w.\, cur \Mapsto (r; w) \sqsubseteq \chi)$$

The code for the exchanger is now modified, to account for the annotations, as follows.

```
1  exchange (v : A) : option A =
2    p ← alloc v U;
3    b ← try_install(p);
4    if b == null then // installation successful
5      sleep (50);
6      x ← retire_collect(p);
7      if x is M w then return (Some w) // successful match, we collected
8      else return None // no match, we retired
9    else // couldn't install, try match other's offer
10     dealloc p;
11     cur ← read_current;
12     if cur == null then return None // no offer available to match
13     else
14       x ← try_match(cur, v);
15       unlink_offer(cur);
16       if x is U then
17         w ← read_node cur;
18         return (Some w) // match successful
19       else return None // match failed
```

Stability. We next present the stable specifications for the actions.

1.

$$\begin{aligned}\mathsf{try\_install}(p) : [v,\tau].\ &\{\sigma_s = p \Mapsto (v;\mathsf{U}) \land \pi_s = \mathsf{empty} \land \tau_s = \mathsf{empty} \land \tau \sqsubseteq \tau_o \cup \|\mu\|\}\\ &\{\tau_s = \mathsf{empty} \land \tau \sqsubseteq \tau_o \cup \|\mu\| \land\\ &\ \mathsf{if}\ r = \mathsf{null}\ \mathsf{then}\ \sigma_s = \mathsf{empty} \land \pi_s = \{p\} \land \exists x.\, p \Mapsto (v;x) \sqsubseteq \chi \land \mathsf{bounded}\ x\ \mathsf{last}(\tau)\ \mu\\ &\ \mathsf{else}\ \sigma_s = p \Mapsto (v;\mathsf{U}) \land \pi_s = \mathsf{empty}\}\end{aligned}$$

This specification says several thing. The easy part is what one would expect, namely, that, if the call was successful, then the offer p, which is initially unmatched, is installed into the shared state ($p \Mapsto (v;x) \sqsubseteq \chi$), and is simultaneously removed from the private heap $\sigma_s = \mathsf{empty}$. The pointer p is added to π_s to indicate that the ownership of p to the *self* thread. We did not change the history $\tau_s = \mathsf{empty}$, and we use the logical variable τ to express the monotonicity of $\tau_o \cup \|\mu\|$, in a stable way, as customary.

Most conjuncts in the specification invoke only self variables, and are hence trivially stable. The only non-trivial property is the existential quantification: $\exists x.\, p \Mapsto (v;x) \sqsubseteq \chi \land \mathsf{bounded}\ x\ \mathsf{last}(\tau)\ \mu$. We discuss it next.

The first conjunct says that the offer p has been moved into the shared state χ. However, we do not say that the hole x remains U, because x can change by a successful matching from some other thread. Nevertheless, once in χ, p remains there, and its value v is unchanged, since we do not have transition to change v or remove p. Thus, this conjunct is stable.

The predicate $\mathsf{bounded}\ x\ k\ m$ is defined as follows.

$$\forall v\ w\ t.\, x = \mathsf{M}\ w \land p \Mapsto (t,v,w) \sqsubseteq m \implies k < \bar{t} < t$$

When instantiated as $\mathsf{bounded}\ x\ \mathsf{last}(\tau)\ \mu$ it captures the following property: If at some point in time, the offer p is matched ($x = \mathsf{M}\ w$), and the offer is not collected ($p \Mapsto (t,v,w) \sqsubseteq \mu$), and the timestamp at which p is matched is t, as can be read off from μ, then both t and $\bar{t}$ are larger than the last timestamp in τ. In other words, the matching, if it occurs, occurs strictly after try_install was invoked. Or put another way, the matching of an offer can only happen after the offer was installed.

This property is stable because x can become $\mathsf{M}\ w$ only by taking the *match* transition. This transition generates the timestamp $l = \mathsf{last}(\tau_o \cup \|\mu\|) + 1$, to ascribe to one end of the offer, and uses $t = l + 1$ for the other end of the offer, which is placed into μ for later collection by the offering thread. Since l must be odd (as explained in the case of *match* transition, because the size of $\tau_o \cup \|\mu\|$ is even), it

must be: $\bar{t} = \overline{l+1} = l < l+1 = t$. Clearly, both $\bar{t}$ and t are larger than any timestamp in $\tau_o \cup \|\mu\|$, and hence are larger than any timestamp in $\tau \sqsubseteq \tau_o \cup \|\mu\|$.

2.
$$\begin{aligned}\mathsf{retire\_collect}(p) : [v, \tau].\ &\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \{p\} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \\ &\quad \exists x.\, p \Mapsto (v, x) \sqsubseteq \chi \wedge \mathsf{bounded}\ x\ \mathsf{last}(\tau)\ \mu\} \\ &\{\sigma_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \\ &\quad \mathsf{if}\ r\ \mathsf{is}\ \mathsf{M}\ w\ \mathsf{then}\ \exists t.\, \tau_s = t \Mapsto (v, w) \wedge \mathsf{last}(\tau) < \bar{t} < t \\ &\quad \mathsf{else}\ \tau_s = \mathsf{empty}\}\end{aligned}$$

We start with $\pi_s = \{p\}$, which means that the thread invoking this action owns the offer p that it wants to either collect, or retire, depending on whether the offer was matched or not. We also know that the offer is in the shared state χ and has a stable lower bound $\mathsf{last}(\tau)$ on its time-stamp, as discussed in the case of try_install. If the offer was matched, then we collect it, i.e., the event $t \Mapsto (v, w)$ is moved into the self history, and π_s is emptied. We also know that $\mathsf{last}(\tau) < \bar{t} < t$, from the lower bound $\mathsf{last}(\tau)$ in the precondition. If the offer was not equal to $\mathsf{M}\ w$, then it must have been U. It couldn't have been R, since a retired offer p cannot appear in π_s by property (v). We retire the offer, which is shown by removing p from π_s.

3.
$$\begin{aligned}\mathsf{try\_match}(cur, v) : [w, \tau].\ &\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \exists x.\, cur \Mapsto (w, x) \sqsubseteq \chi\} \\ &\{\sigma_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \exists x.\, cur \Mapsto (w, x) \sqsubseteq \chi \wedge x \neq \mathsf{U} \wedge \\ &\quad \mathsf{if}\ r = \mathsf{U}\ \mathsf{then}\ \exists t.\, \tau_s = t \Mapsto (v, w) \wedge \mathsf{last}(\tau) < t < \bar{t}\ \mathsf{else}\ \tau_s = \mathsf{empty}\}\end{aligned}$$

We start with the knowledge that cur is an offer in the shared heap. If the action returns U, then the matching was successful, hence a new item $t \Mapsto (v, w)$ is placed in our history. We also know that the hole x of cur is filled with $\mathsf{M}\ v$, but, it turns out we don't need to track such a detail; it suffices to know that $x \neq \mathsf{U}$. The timestamp t is chosen as per the *match* transition, to be one larger than the last timestamp in $\tau_o \cup \|\mu\|$, hence $\mathsf{last}(\tau) < t$. Also, t is odd, and hence also $t < \bar{t}$. Notice how $t < \bar{t}$ indicates that the timestamp t is a timestamp generated for the matching thread, whereas the other inequality $\bar{t} < t$ indicates that t is the timestamp to be collected by the offering thread.
On the other hand, if the failed match, the self history τ_s remains unchanged, but we also know that the hole $x \neq \mathsf{U}$, because, had x been U, the match would have succeeded.
Most of the conjuncts are trivially stable as they talk about self variables, or are otherwise pure. There is only one property whose stability needs comment, as the property concerns shared state:

$$\exists x.\, cur \Mapsto (w; x) \sqsubseteq \chi \wedge x \neq \mathsf{U}$$

This property is stable, because an offer cur in the shared heap is never removed, and its value w is never changed, by any thread. The hole x may change, but only if $x = \mathsf{U}$. If the value of the hole is $\mathsf{M}\ w$ or R, no transition exists that modifies it.

4.
$$\begin{aligned}\mathsf{unlink\_offer}(cur) : [\tau, w].\ &\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \\ &\quad \exists x.\, cur \Mapsto (v; x) \sqsubseteq \chi \wedge x \neq \mathsf{U}\} \\ &\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \exists x.\, cur \Mapsto (v; x) \sqsubseteq \chi\}\end{aligned}$$

We start with an offer cur in the shared heap, which is either matched or retired. The action unlinks cur from g, though we do not need to specify that in the postcondition. As we have seen in the case of readSentinel in the Treiber's stack example, unlinking cur from g is only important for liveness, but not for partial correctness of the exchanger.
We do need to prove that $x \neq \mathsf{U}$ before we invoke this action, because the action has to preserve the property (iv) that only the currently linked offer is unmatched. But, as it turns out, we do not need to track the value of x in the postcondition, so we drop the property $x \neq \mathsf{U}$.

5.
$$\begin{aligned}\mathsf{read\_current} : [\tau].\ &\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\|\} \\ &\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \\ &\quad r = \mathsf{null} \vee \exists v\ x.\, r \Mapsto (v, x) \sqsubseteq \chi\}\end{aligned}$$

This action reads the value of g. We either obtain $r = \mathsf{null}$, or otherwise return a pointer to a valid offer. We do not say that the offer is the top one, because it does not have to be, as an interfering thread may unlink that offer after read_current terminates.

6.

$$\mathsf{read\_node}(cur) : [\tau, v].\ \{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \exists x.\, cur \mapsto (v, x) \sqsubseteq \chi\}$$
$$\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \exists x.\, cur \mapsto (v, x) \sqsubseteq \chi \wedge r = v\}$$

Finally, when we read the offer passed as a pointer cur, we obtain the value of the offer associated with cur.

4.1.3 Proof outline

$\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\|\}$
  $p \leftarrow \mathsf{alloc}\ (v; \mathsf{U})$;
$\{\sigma_s = p \mapsto (v; \mathsf{U}) \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\|\}$
  $b \leftarrow \mathsf{try\_install}(p)$;
$\{\tau_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge$
 if $b = \mathsf{null}$ then $\sigma_s = \mathsf{empty} \wedge \pi_s = \{p\} \wedge \exists x.\, p \mapsto (v; x) \sqsubseteq \chi \wedge \mathsf{bounded}\ x\ \mathsf{last}(\tau)\ \mu$
 else $\sigma_s = p \mapsto (v; \mathsf{U}) \wedge \pi_s = \mathsf{empty}\}$
  if $b == \mathsf{null}$ then
    sleep (50);
$\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \{p\} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge$
 $\exists x.\, p \mapsto (v; x) \sqsubseteq \chi \wedge \mathsf{bounded}\ x\ \mathsf{last}(\tau)\ \mu\}$
    $x \leftarrow \mathsf{retire\_collect}(p)$;
$\{\sigma_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge$
 if x is M w then $\exists t.\, \tau_s = t \mapsto (v; w) \wedge \mathsf{last}(\tau) < \bar{t} < t$ else $\tau_s = \mathsf{empty}\}$
    if x is M w then return (Some w)
    else return None
$\{\sigma_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge$
 if r is Some w then $\exists t.\, \tau_s = t \mapsto (v; w) \wedge \mathsf{last}(\tau) < t, \bar{t}$ else $\tau_s = \mathsf{empty}\}$
  else
$\{\sigma_s = p \mapsto (v; \mathsf{U}) \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\|\}$
    dealloc p;
$\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\|\}$
    $cur \leftarrow \mathsf{read\_current}$;
$\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\|,$
 $cur = \mathsf{null} \vee \exists w\ x.\, cur \mapsto (w; x) \subseteq \chi\}$
    if $cur == \mathsf{null}$ then return None
  $\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge r = \mathsf{None}\}$
    else
$\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \exists w\ y.\, cur \mapsto (w; y) \sqsubseteq \chi\}$
// introduce w into scope
$\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \exists y.\, cur \mapsto (w; y) \sqsubseteq \chi\}$
      $x \leftarrow \mathsf{try\_match}(cur, v)$;
$\{\sigma_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o \cup \|\mu\| \wedge \exists y.\, cur \mapsto (w; y) \subseteq \chi \wedge y \neq \mathsf{U} \wedge$
 if $x = \mathsf{U}$ then $\exists t.\, \tau_s = t \mapsto (v, w) \wedge \mathsf{last}(\tau) < t < \bar{t}$ else $\tau_s = \mathsf{empty}\}$
// introduce T and open its scope, and frame
$\{(\sigma_s = \mathsf{empty} \wedge \tau_s = T \wedge \pi_s = \mathsf{empty} \wedge T \cup \tau \sqsubseteq T \cup \tau_o \cup \|\mu\| \wedge \exists y.\, cur \mapsto (w; y) \subseteq \chi \wedge y \neq \mathsf{U})$
 $\circledast (\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge$
   if $x = \mathsf{U}$ then $\exists t.\, T = t \mapsto (v, w) \wedge \mathsf{last}(\tau) < t < \bar{t}$ else $T = \mathsf{empty})\}$
      $\mathsf{unlink\_offer}(cur)$; // choose $T \cup \tau$ for the logical variable τ

```
40  {(σ_s = empty ∧ τ_s = T ∧ π_s = empty ∧ T ∪ τ ⊑ T ∪ τ_o ∪ ‖μ‖ ∧ ∃y. cur ↦ (w; y) ⊆ χ)
41   ⊛ (σ_s = empty ∧ τ_s = empty ∧ π_s = empty ∧
42       if x = U then ∃t. T = t ↦ (v, w) ∧ last(τ) < t < t̄ else T = empty)}
43  // eliminate T, simplify ⊛
44  {σ_s = empty ∧ π_s = empty ∧ τ ⊑ τ_o ∪ ‖μ‖ ∧ ∃y. cur ↦ (w; y) ⊆ χ ∧
45   if x = U then ∃t. τ_s = t ↦ (v, w) ∧ last(τ) < t < t̄ else τ_s = empty}
46            if  x == U  then
47           {σ_s = empty ∧ π_s = empty ∧ τ ⊑ τ_o ∪ ‖μ‖ ∧ ∃y. cur ↦ (w; y) ⊆ χ ∧
48            ∃t. τ_s = t ↦ (v, w) ∧ last(τ) < t < t̄}
49           // introduce T and open its scope, and frame
50           {(σ_s = empty ∧ τ_s = T ∧ π_s = empty ∧ T ∪ τ ⊑ T ∪ τ_o ∪ ‖μ‖ ∧ ∃y. cur ↦ (w; y) ⊆ χ)
51            ⊛ (σ_s = empty ∧ τ_s = empty ∧ π_s = empty ∧ ∃t. T = t ↦ (v, w) ∧ last(τ) < t < t̄)}
52              w ← read_node(cur);
53           {(σ_s = empty ∧ τ_s = T ∧ π_s = empty ∧ T ∪ τ ⊑ T ∪ τ_o ∪ ‖μ‖ ∧ ∃y. cur ↦ (w; y) ⊆ χ)
54            ⊛ (σ_s = empty ∧ τ_s = empty ∧ π_s = empty ∧ ∃t. T = t ↦ (v, w) ∧ last(τ) < t < t̄)}
55           // eliminate T, simplify ⊛, remove ∃y. cur ↦ (w; y) ⊑ χ by weakening
56           {σ_s = empty ∧ π_s = empty ∧ τ ⊑ τ_o ∪ ‖μ‖ ∧ ∃t. τ_s = t ↦ (v, w) ∧ last(τ) < t < t̄)}
57              return (Some w)
58           {σ_s = empty ∧ π_s = empty ∧ τ ⊑ τ_o ∪ ‖μ‖ ∧
59            ∃t.τ_s = t ↦ (v, w) ∧ last(τ) < t, t̄ ∧ r = Some w}
60         else
61           {σ_s = empty ∧ τ_s = empty ∧ π_s = empty ∧ τ ⊑ τ_o ∪ ‖μ‖}
62              return (None)
63           {σ_s = empty ∧ τ_s = empty ∧ π_s = empty ∧ τ ⊑ τ_o ∪ ‖μ‖ ∧ r = None}
64  {σ_s = empty ∧ π_s = empty ∧ τ ⊑ τ_o ∪ ‖μ‖ ∧
65   if r is Some w then ∃t. τ_s = t ↦ (v, w) ∧ last(τ) < t, t̄ else τ_s = empty}
```

The proof outline is now largely self-explanatory, as the assertions merely follow straightforwardly from the specifications of the primitive actions.

The only somewhat non-trivial situation arises in line 35, and later also in line 49. It has to do with the fact that the actions unlink_offer(cur) (in line 39, and read_node(cur) in line 52, have to be invoked not with the specifications that we give for them previously, but with framed versions.

Indeed, both these actions are specified using $\tau_s = \mathsf{empty}$, since neither of them changes the history, but are invoked in the program in situations when the history may have already been enlarged with a timestamp. Thus, we frame the specs for the actions as shown below, by introducing a new logical variable T, to stand for the initial value of τ_s. As explained in Chapter 2, that means we also need to replace the occurrences of τ_o with $T \cup \tau_o$, to capture that the frame T is taken from the environment.

$$\mathsf{unlink\_offer}(cur) : [\tau, w, T].\ \{\sigma_s = \mathsf{empty} \wedge \tau_s = T \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq T \cup \tau_o \cup \|\mu\| \wedge \exists x.\, cur \mapsto (v; x) \sqsubseteq \chi \wedge x \neq \mathsf{U}\}$$
$$\{\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq T \cup \tau_o \cup \|\mu\| \wedge \exists x.\, cur \mapsto (v; x) \sqsubseteq \chi\}$$

$$\mathsf{read\_node}(cur) : [\tau, v, T].\ \{\sigma_s = \mathsf{empty} \wedge \tau_s = T \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq T \cup \tau_o \cup \|\mu\| \wedge \exists x.\, cur \mapsto (v, x) \sqsubseteq \chi\}$$
$$\{\sigma_s = \mathsf{empty} \wedge \tau_s = T \wedge \pi_s = \mathsf{empty} \wedge \tau \sqsubseteq T \cup \tau_o \cup \|\mu\| \wedge \exists x.\, cur \mapsto (v, x) \sqsubseteq \chi \wedge r = v\}$$

In the proof outline, we need to further frame these specifications, in order to propagate the relations between the various functional variables, as already explained in Chapter 1, Exercise 1. Thus, in both cases, we frame with respect to a predicate of the form

$$\sigma_s = \mathsf{empty} \wedge \tau_s = \mathsf{empty} \wedge \pi_s = \mathsf{empty} \wedge R$$

where R is pure; that is, R only constrains variables, but no labeled projections from the state (e.g., no χ, τ_o, etc.). Such framing corresponds to simply conjoining R to both the precondition and the postcondition of the actions above.

We also note that in both lines 35 and 49, we have a conjunct of the form

$$T \cup \tau \sqsubseteq T \cup \tau_o \cup \|\mu\|$$

but the preconditions of the framed versions of unlink_offer and read_node both have

$$\tau \subseteq T \cup \tau_o \cup \|\mu\|$$

But, notice that τ is a local variable in both these specifications, and should not be confused with τ from the proof outline. Thus, we instantiate the τ that is local to the specs, with $T \cup \tau$, where T and τ are the variables that are in scope in the proof outline.

We also note that $T \cup \tau \sqsubseteq T \cup \tau_o \cup \|\mu\|$ is equivalent to $\tau \sqsubseteq \tau_o \cup \|\mu\|$ because histories are cancellative with respect to $\cup$, as long as T that we are canceling does not overlap with the other histories involved.

Exercise 4.1.1. Consider the following client program for the exchanger, which takes a value v and loops until it exchanges it.

```
exchange' (v : A) : A =
  w' ← exchange v;
  if w' is Some w then return w else exchange' v
```

Next, `exchange'` is iterated over a sequence, exchanging each element in order, appending the received matches to an accumulator.

```
ex_seq (vs, ac : list A) : list A =
  if vs is v::vs' then
    w ← exchange' v;  ex_seq (vs', snoc ac w)
  else return ac
```

Consider what is required in order to show that:

$$e = \texttt{ex\_seq}\ (vs_1, \texttt{nil}) \parallel \texttt{ex\_seq}\ (vs_2, \texttt{nil})$$

exchanges vs_1 and vs_2, *i.e.*, returns the pair (vs_2, vs_1). This is a valid property, if e runs without interference, so that the two threads in e have no choice but to exchange the values between themselves. Can you express a generalization of this property in terms of τ_o, so that if τ_o = empty, we can derive that the lists are exchanged?

4.2 Spanning tree

The second example that we present in this section is a procedure, which we call span, which takes a *binary directed graph* laid out in the shared state as a linked structure, and concurrently traverses the graph, modifying it in-place by marking the nodes and pruning the edges, until ultimately, the graph is transformed into its spanning three. That is, all the nodes are marked, and the remaining edges form a tree.

Verifying graph algorithms in separation logic has traditionally been considered difficult, even in the sequential case, let alone the concurrent one. Separation logic was considered to excel when dealing with data structures such as linked lists and trees, where there is not much *sharing*, i.e., where there is typically only a single pointer going into every node, and the exceptional cases are rare, and can be described systematically. Such data structures can be divided into disjoint heaps, each of which described by a separation logic formula, and the links between the disjoint pieces provided separately.

With graphs, one cannot do this, as the nodes of a graph can be connected in arbitrary ways. While one can always divide the graph into disjoint sets of nodes, no division is guaranteed to offer a particularly easy, natural, or uniform way to describe the edges that connect the divided pieces. This has made it difficult to verify graph programs in separation logic. Specifically, we can try to use CSL in the customary idiom, and decide to split the heap of the graph so that concurrent subthreads operate over disjoint subheaps. But

then, as each thread merely follows the nodes reachable from the pointer it is given, it will quickly escape the heap ascribed to it, and will need to encroach onto other thread's heap.

In this section, we illustrate how subjective auxiliary state helps with this. The idea is to *not* split the heap of the graph when forking threads. The full graph remains in the shared state. What the threads split on, is the the value of an auxiliary field that keeps the set of nodes that each thread has marked. The specifications and proofs shown here are a simplified versions of ones shown in [31].

We first present the code of the span procedure, and discuss the intuition.

```
1  span (x : ptr) : bool =
2      if x == null then return false;
3      else
4          b ← CAS(x.m, 0, 1);
5          if b then
6              (r_l, r_r) ← (span(x.l) || span(x.r));
7              if ¬r_l then x.l := null;
8              if ¬r_r then x.r := null;
9              return true;
10         else return false;
```

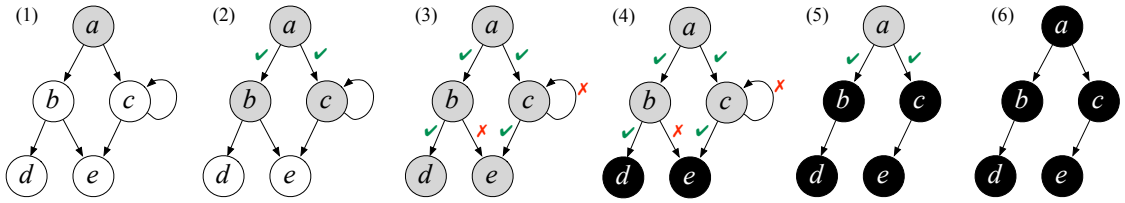
The graph is implemented as a memory region where each pointer p represents a node of the graph, and points to a triple of values, laid out in successive memory locations, i.e.

$$p \Mapsto (m; l; r)$$

The triple's first component is a boolean m indicating whether the node is *marked*; the second and third components are pointers to the node's *left* and *right* successors, or `null` if a successor does not exist. Thus, in the above program, we could have written x instead of $x.m$, $x+1$ instead of $x.l$ and $x+2$ instead of $x.r$, but we chose the more symbolic notation for clarity. Also, note that null cannot appear in the heap (by definition of heaps), hence null is not a valid node in any graph.

The recursive procedure span takes an argument x, which is pointer to a node in the graph, and constructs the spanning tree rooted in x by traversing the graph and removing redundant edges. If x is null, span returns false (line 2). Otherwise, it tries to mark the node by invoking the compare-and-set (CAS) operation (line 4). If CAS fails, then x was previously marked, i.e., included in the spanning tree by another call to span, so no need to continue (line 10). If CAS succeeds, two new parallel threads are spawned (line 6): span is called recursively on the left ($x.l$) and right ($x.r$) successors of x, returning respectively the booleans r_l and r_r upon termination. When r_l is false, $x.l$ has already been marked, i.e., span has already found a spanning sub-tree that includes $x.l$ but doesn't traverse the edge from x to $x.l$. That edge is superfluous, and thus pruned by nullifying $x.l$ (line 7). The behavior is dual for $x.r$.

The following figure illustrates a possible execution of span.



Referring to the figure, a node is colored gray right after a corresponding thread successfully marks it (line 4 of span code). It is colored black right before the thread returns true (line 9). A black sub-tree is *logically ascribed* to a thread that marked its root. ✔ indicates a child thread exploring an edge and succeeding in marking its target node; ✗ indicates a thread that failed to do so. In (1), the main thread marks a and forks two children; (2) the children succeed in marking b and c; (3) only one thread succeeds in marking e; (4) the processing of d and e is done; (5) the redundant edges $b \to e$ and $c \to c$ are removed by the corresponding threads; (6) the initial thread joins its children and terminates.

Why does span compute a *tree*? Assume that (a) the graph initially represented in memory is *connected*, and that (b) it is modified only by recursive calls to span, with no external interference. To see that span obtains a tree, consider four cases, according to the values of r_l and r_r. If $r_l = r_r = \mathsf{true}$, then the calls to span have, by recursive assumption, computed trees from sub-graphs rooted at $x.l$ and $x.r$ to trees. These trees have disjoint nodes, and there are no edges connecting them. As we will show in Section 4.2.1, this will follow from a property that each tree is *maximal* wrt. the resulting *final* graph's topology (*i.e.*, the tree cannot be extended with additional nodes). Lines 7 and 8 of span preserve the edges from x to $x.l$ and $x.r$; thus x becomes a root of a tree with sub-trees pointed to by $x.l$ and $x.r$ (case (6) in the above figure). If $r_l = \mathsf{true}$ and $r_r = \mathsf{false}$, the recursive call has computed a tree from the sub-graph rooted at $x.l$, but $x.r$ has been found marked. The edge to $x.r$ is removed in line 8, but that to $x.l$ is preserved in line 7; x becomes a root of a tree with left sub-tree rooted in $x.l$, and right sub-tree empty (case (5) in the figure). The case $r_l = \mathsf{false}$, $r_r = \mathsf{true}$ is dual. The case $r_l = r_r = \mathsf{false}$ results in a sub-tree containing just the root x (case (4) in the figure).

Why does span construct a *spanning* tree? Consider the *front* of the constructed tree in the initial graph. These are the nodes immediately reachable *in the initial graph*, from the nodes of the constructed tree. For example, in case (5) of the figure above, the front of b are nodes d, e, as d and e are children of b in the initial graph from case (1). We will show in Section 4.2.1 that, at the end of span, the front of the constructed tree must contain only marked nodes, as otherwise span would have proceeded with node marking, instead of terminating. Thus, the front of the tree constructed by the top-level call to span must actually be included in the very same tree. Otherwise, there exists a node which is marked but not in the tree. Therefore, this node must have been marked by another thread, thus contradicting assumption (b). Since, by (a), the initial graph is connected, a tree which contains its front must contain all the nodes, and thus be a spanning one.

4.2.1 Auxiliary state and span specification

The subjective components (i.e., a_s and a_j) of the resource for the spanning tree are finite sets of pointers, representing nodes in the shared graph, that the *self* and *other* threads *marked*, respectively. To be consistent with the rest of the material, we use Greek names μ_s and μ_o for these values.

The sets are obviously PCMs under the operation of disjoint union, with the empty set as the unit. The component a_j consists of a heap σ_j storing the graph physically, and an auxiliary component γ, storing a mathematical equivalent of σ_j.

More precisely, γ is a triple $(\mathsf{nodes}, \mathsf{left}, \mathsf{right})$, where nodes is the set of nodes of the graph (i.e., pointers), and $\mathsf{left}, \mathsf{right} : \mathsf{ptr} \to \mathsf{ptr}$ are functions supplying the left and the right child of the input pointer (or null if the input pointer is not a node in the graph). We name the projections out of γ, and write $\mathsf{nodes}\ \gamma$, $\mathsf{left}\ \gamma$ and $\mathsf{right}\ \gamma$ when we want to obtain the node set, and the edge functions for the graph γ. As an abbreviation, we also write $\mathsf{ch}\ g\ x$ for the pair $(\mathsf{left}\ g\ x, \mathsf{right}\ g\ x)$. We only consider triples γ where $\mathsf{null} \notin \mathsf{nodes}\ g$, and where for every $x \in \mathsf{nodes}\ g\ x$, the pointers $\mathsf{left}\ g\ x$ and $\mathsf{right}\ g\ x$ are either null or themselves in $\mathsf{nodes}\ g\ x$.

We will impose by the resource invariant that the components γ and σ_j are kept synchronized, i.e., γ is always the mathematical representation of the graph in σ_j. That way, our specifications will not have to manipulate the heap σ_j to extract the required components, but will directly access the required information about each node.

Now we give the desired specification for span and discuss it. We also abbreviate by μ the sum $\mu_s \cup \mu_o$.

$$\begin{aligned}\mathsf{span}(x) : [g].\ &\{\gamma \leqslant_\mu g \wedge \mu_s = \mathsf{empty} \wedge x \in \{\mathsf{null}\} \cup \mathsf{nodes}\ g\} \\ &\{\gamma \leqslant_\mu g \wedge \mathsf{if}\ r\ \mathsf{then}\ \mathsf{subspan}_{\gamma,g,\mu}\ x\ \mu_s\ \mathsf{else}\ \mu_s = \mathsf{empty} \wedge x \in \{\mathsf{null}\} \cup \mu\}\end{aligned}$$

We define the helper predicates $g_1 \leqslant_m g_2$ and $\mathsf{subspan}_{g_1,g_2,m}\ x\ t$ next. The predicate $g_1 \leqslant_m g_2$ says that the graph g_1 is obtained from g_2 by pruning (some) edges out of nodes from the set m. Thus, the sets of nodes of g_1 and g_2 are equal. The child pointers of a node $x \notin m$ are the same in both graphs, but they can be set to null (hence, the edge is pruned), in the graph g_1, for a node $x \in m$.

$$\begin{aligned}g_1 \leqslant_m g_2 \quad \widehat{=} \quad &\mathsf{nodes}\ g_1 = \mathsf{nodes}\ g_2 \wedge (\forall x \notin m.\, \mathsf{ch}\ g_1\ x = \mathsf{ch}\ g_2\ x) \wedge \\ &\forall x \in m.\, \mathsf{left}\ g_1\ x \in \{\mathsf{null}, \mathsf{left}\ g_2\ x\} \wedge \mathsf{right}\ g_1\ x \in \{\mathsf{null}, \mathsf{right}\ g_2\ x\}\end{aligned}$$

In the specification of span, the predicate is used in the form $\gamma \leqslant_\mu g$, where γ is the current graph stored in the state, g is an input logical variable, and μ is the set of marked nodes. In particular, notice that if we

start the execution of span in the setting where all the nodes are unmarked, i.e. $\mu = \mathsf{empty}$, then γ and g must have the same edges, and hence, must be equal graphs. Thus, the meaning of the logical variable g is to stand for the initial graph, in the top-level call to span.

In the predicate $\mathsf{subspan}_{g_1,g_2,m}\ x\ t$, g_1 and g_2 are graphs as above, m and t are sets of nodes, and x is a node. The predicate states that x is non-null, that t is a sub-tree in the graph g_1, with x as a root. Moreover, t is a maximal such sub-tree in g_1, i.e., it cannot be extended with further nodes. In other words, the front of the tree t, that is, the set of nodes reachable from t in one step by edges of g_1, is included in t. Finally, when the nodes t are considered as a sub-graph of g_2, the their front in g_2 is bounded from above by m:

$$\mathsf{subspan}_{g_1,g_2,m}\ x\ t\ \widehat{=}\ x \neq \mathsf{null} \wedge \mathsf{subtree}\ g_1\ x\ t \wedge \mathsf{front}\ g_1\ t \sqsubseteq t \wedge \mathsf{front}\ g_2\ t \sqsubseteq m.$$

Here, as described:

$$\mathsf{front}\ g\ t\ \widehat{=}\ \{\mathsf{left}\ g\ x \mid x \in t\} \cup \{\mathsf{right}\ g\ x \mid x \in t\} - \mathsf{null}.$$

On the other hand, t is a subtree rooted at x in a graph g, if x is a node in t ($x \in t$), and every other node in t is reachable from x by means of a unique path:

$$\mathsf{subtree}\ g\ x\ t\ \ \widehat{=}\ x \in t \wedge \forall y \in t.\, \exists!p.\, \mathsf{path}\ p\ g\ x\ y\ t.$$

We do not formally define the predicate path $p\ g\ x\ y\ t$ here, but just say that it holds iff p is a list of nodes, starting in x, ending in y, containing only elements from t, and each consecutive pair of nodes n_1 and n_2 in p, represents an edge in g (i.e., $n_2 \in \{\mathsf{left}\ g\ n_1, \mathsf{right}\ g\ n_1\}$).

In the specification of span, the predicate is used in the form $\mathsf{subspan}_{\gamma,g,\mu}\ x\ t$, where γ is the current graph stored in the state, g is the input graph, μ is the set of marked nodes, and $t = \mu_s$ are the nodes marked by the self thread. Thus, abbreviating $t = \mu_s$, we have $x \neq \mathsf{null}$, $\mathsf{subtree}\ \gamma\ x\ t$, $\mathsf{front}\ \gamma\ t \sqsubseteq t$, and $\mathsf{front}\ g\ t \sqsubseteq \mu$.

The conjunct $x \neq \mathsf{null}$ tells us that if span is called a pointer which is a node in the initial graph (hence, is not null), then span must return $r = \mathsf{true}$ (subject to returning at all, as usual in partial correctness).

The conjunct $\mathsf{subtree}\ \gamma\ x\ t$ tells us that the call to span marked a set of nodes t, which is a tree, rooted in x in the ending graph.

The conjunct $\mathsf{front}\ \gamma\ t \sqsubseteq t$ tells us that this is a maximal such tree, which cannot be extended further in γ. This property is not important for the final result, but it is important in the nested recursive calls. In the span procedure, we make two concurrent recursive concurrent calls (line 6 of the code), which themselves obtain two sub-trees of the initial graph. We need to know that these sub-trees are maximal in the above sense, i.e., there are no edges connecting a node of one tree with a node of another, so that we can combine these two trees into a larger one with a root x. If the trees were not maximal, the combination would not have been a tree.

The conjunct $\mathsf{front}\ g\ t\ \sqsubseteq \mu$ says that all the nodes reachable from t in the original graph are marked. Thus, in the top-level call, where there is no interference, i.e., no other threads are running concurrently with span to mark nodes, we will know that $\mu_o = \mathsf{empty}$, and thus $\mu = \mu_s \cup \mu_o = \mu_s$. But then, $\mu = \mu_s = t$, i.e. $\mathsf{front}\ g\ t\ \sqsubseteq t$. Under the assumption that g a connected graph, rooted in node x (i.e., from x we can reach any other node in g), the last property means that t contains all of g's nodes, i.e., it is *spanning*. Indeed, if t is not spanning, then there exists a node $n \notin t$, and a path from x to n has to leave t at some other node m. But then $m \in \mathsf{front}\ g\ t$ and $m \notin t$, which is a contradiction with $\mathsf{front}\ g\ t\ \sqsubseteq t$.

Note 4.2.1 (On deriving the spec for the top-level call).

As we shall see, the above specification is appropriate as a loop invariant for deriving the correctness of span, including the recursive sub-calls. But, what we ultimately want is a somewhat weaker specification that applies only to the top-level call to span, when we know that the starting graph is connected from x, and we want to know that the ending graph is a spanning tree. That is, informally, we want something like:

$$\begin{array}{l}\mathsf{span\text{-}top} : [g].\ \{\gamma \leqslant_\mu g \wedge \mu = \mathsf{empty} \wedge \mathsf{connected}\ x\ \gamma \wedge x \in \mathsf{nodes}\ g\} \\ \qquad\qquad\qquad\quad \{\gamma \leqslant_\mu g \wedge \mathsf{subtree}\ g\ x\ \gamma\}\end{array}$$

In order to derive this specification, FCSL provides the construction hide, which we omitted in these notes, but mentioned in Note 1. For a description of how to derive a formal version of this ultimately desired specification, we refer the interested reader to [31].

Finally, if span fails, i.e., the call returns $r = \mathsf{false}$, it is important to know that we constructed no tree ($t = \mathsf{empty}$), and that we failed either because we invoked span with $x = \mathsf{null}$, or because x was already marked by some other concurrent thread, and hence span couldn't proceed. As before in the case of the conjunct $\mathsf{front}\ \gamma\ t \sqsubseteq t$, this knowledge is not important for the top-level call, but it is necessary for bootstrapping the recursion. In particular, in line 6 of the span code, we have:

$$\mathsf{span}(x_l) \parallel \mathsf{span}(x_r).$$

Let us suppose that the call to x_l fails (and thus, x_l is subsequently pruned), and the call to x_r returns a tree $\mu_s = t_r$ (the other combinations of failing and succeeding are similar). Then we need to know that x_l is marked, assuming it weren't null in the original graph, for the following reasons. The tree that the enclosing call to span returns, will be constructed with x as a root, with empty left sub-tree, and t_r as the right sub-tree. The set of nodes of that tree is $t = \{x\} \cup t_r$. Even if x has no left child in γ, it does have a left child x_l in g. Thus, the front of t in g includes x_l, and we need to know that x_l is marked in order to prove the conjunct $\mathsf{front}\ g\ t \sqsubseteq t$ that features as a conjunct in the predicate $\mathsf{subspan}_{\gamma,g,\mu}\ x\ t$ from the postcondition of span.

4.2.2 Resource invariant, transitions, atomics

Resource invariant. In addition to the conventional properties (heap validity, subjective validity), the resource invariant's task in span is to relate σ_j with γ, and with μ.

In particular, we have that σ_j encodes the graph γ, with the marking given by μ; these quantities are all defined elements of their respective PCMs. Furthermore μ does not record as marked a pointer which is not actually in the graph. That is:

$$\begin{aligned} I \ \widehat{=}\ & \mathsf{valid}\ \sigma_j \wedge \mathsf{valid}\ \mu \wedge \mathsf{dom}\ \sigma_j = \mathsf{nodes}\ \gamma \wedge \mu \sqsubseteq \mathsf{nodes}\ \gamma \wedge \\ & \forall x \in \mathsf{dom}\ \sigma_j.\, x \Mapsto (x \in \mu, \mathsf{left}\ \gamma\ x, \mathsf{right}\ \gamma\ x) \sqsubseteq \sigma_j \end{aligned}$$

Transitions. The resource contains three transitions: $mark(x)$ for marking the node x, and $prune\_left(x)$, for pruning $x.l$, and symmetrically for $prune\_right(x)$.

$Mark(x)$ transition expects that the mark-bit of x is false; it switches it to true, and puts x into μ_s.

$$\begin{aligned} mark(x)\ \widehat{=}\ [m, x_l, x_r, h].\, & (\mu_s = m \wedge \sigma_j = x \Mapsto (\mathsf{false}, x_l, x_r) \cup h) \rightsquigarrow \\ & (\mu_s = \{x\} \cup m \wedge \sigma_j = x \Mapsto (\mathsf{true}, x_l, x_r) \cup h). \end{aligned}$$

$prune\_left(x)$ transition prunes the left child of x, by writing null into the left pointer of x in σ_j, and consequently modifying the mathematical representation γ of σ_j. It requires that x is marked by the self thread, so that the thread has the permission to prune the edges coming out of x. Symmetrically, $prune\_right(x)$ does the same for the right child.

$$\begin{aligned} & prune\_left(x)\ \widehat{=} \\ & \quad [m, x_l, x_r, h, g].\, (\mu_s = \{x\} \cup m \wedge \sigma_j = x \Mapsto (\mathsf{true}, x_l, x_r) \cup h \wedge \gamma = g) \rightsquigarrow \\ & \qquad\qquad (\mu_s = \{x\} \cup m \wedge \sigma_j = x \Mapsto (\mathsf{true}, \mathsf{null}, x_r) \cup h \wedge \gamma = (\mathsf{nodes}\ g, (\mathsf{left}\ g)[x \mapsto \mathsf{null}], \mathsf{right}\ g) \end{aligned}$$

$$\begin{aligned} & prune\_right(x)\ \widehat{=} \\ & \quad [m, x_l, x_r, h, g].\, (\mu_s = \{x\} \cup m \wedge \sigma_j = x \Mapsto (\mathsf{true}, x_l, x_r) \cup h \wedge \gamma = g) \rightsquigarrow \\ & \qquad\qquad (\mu_s = \{x\} \cup m \wedge \sigma_j = x \Mapsto (\mathsf{true}, x_l, \mathsf{null}) \cup h \wedge \gamma = (\mathsf{nodes}\ g, \mathsf{left}\ g, (\mathsf{right}\ g)[x \mapsto \mathsf{null}]) \end{aligned}$$

Atomic actions.

1. $\mathsf{try\_mark}(x)$ annotates with auxiliary code the CAS command from line 4 of span code. if CAS succeeds, the $mark(x)$ transition is taken. If the CAS fails, the *idle* transition is taken, but we know that the x node was marked.

$$\mathsf{try\_mark}(x) \;\widehat{=}\; \mathsf{if}\ r\ \mathsf{then}\ mark(x)\ \mathsf{else}\ idle\ \ (x \in \mu_s \cup \mu_o)$$

It is easy to see that when auxiliary state is erased, $\mathsf{try\_mark}(x)$ behaves, on the underlying heap, like a CAS which modifies $x.m$.

2. $\mathsf{prune\_left}(x)$ annotates with auxiliary code the $x.l := \mathsf{null}$ command from line 7. It does not return any values (i.e., its result is of unit type), and it simply takes the $prune\_left(x)$ transition. Symmetrically for prune_right.

$$\begin{array}{rcl} \mathsf{prune\_left}(x) & \widehat{=} & prune\_left(x) \\ \mathsf{prune\_right}(x) & \widehat{=} & prune\_right(x) \end{array}$$

It is easy to see that after erasure the commands correspond to modifying one pointer, $x.l$ in the case of $\mathsf{prune\_left}(x)$ and $x.r$ in the case of $\mathsf{prune\_right}(x)$.

3. We also require two atomic actions for reading the left and right child, respectively, of a given node x. These are both defined to take the $idle$ transition, but constrain the return result r to be the appropriate pointer.

$$\begin{array}{rcl} \mathsf{read\_left}(x) & \widehat{=} & idle(r = \mathsf{left}\ \gamma\ x) \\ \mathsf{read\_right}(x) & \widehat{=} & idle(r = \mathsf{right}\ \gamma\ x) \end{array}$$

It is easy to see that after erasure, the commands correspond to reading one pointer, $x.l$ and $x.r$, respectively.

The annotation of span with auxiliary code then looks as follows.

```
1  span (x : ptr) : bool =
2      if x == null then return false;
3      else
4          b ← try_mark(x);
5          if b then
6              x_l ← read_left(x);
7              x_r ← read_right(x);
8              (r_l, r_r) ← (span(x_l) || span(x_r));
9              if ¬r_l then prune_left(x);
10             if ¬r_r then prune_right(x);
11             return true;
12         else return false;
```

Stability.

1. $$\begin{array}{l} \mathsf{try\_mark}(x) : [g].\ \{\gamma \leqslant_\mu g \wedge \mu_s = \mathsf{empty} \wedge x \in \mathsf{nodes}\ g\} \\ \qquad\qquad \{\gamma \leqslant_\mu g \wedge \mathsf{if}\ r\ \mathsf{then}\ \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x\ \mathsf{else}\ \mu_s = \mathsf{empty} \wedge x \in \mu\} \end{array}$$

The postcondition says that no matter the return result of try_mark, we know that x is marked in the end. This may be because try_mark succeeded, or because try_mark failed, as x was already marked by someone else. In the success case, however, the set of *self*-marked nodes μ_s is extended by x, whereas it remains unchanged in the failure case, as per the definition of the try_mark action, which is a combination of the $mark$ and $idle$ transitions.
Additionally, in the success case, we expose that the children nodes of x remain unchanged

$$\mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x.$$

In the precondition, this property is subsumed by $\gamma \leqslant_\mu g$, since if we managed to mark x, then x must have been unmarked to start with, and $\gamma \leqslant_\mu g$ says that γ and g agree on the edges out of unmarked nodes. As x is marked in the postcondition, the predicate $\gamma \leqslant_\mu g$ in the postcondition does not say anymore that edges out of x are preserved, and is thus weaker than the identical predicate in the precondition. To recover the strength, we expose the missing property explicitly in the then branch of the conditional.

Because our resource only possesses transitions that change edges by pruning them out of *self*-marked nodes, the *other* threads cannot change the edges out of *unmarked* nodes, and out of nodes owned by the *self* thread. In particular, this means that the properties $\gamma \leqslant_\mu g$, and $\mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = -$, are stable. Also, because our resource only possesses transitions that increase the set of marked nodes, the property $x \in \mu$ is stable.

2.
$$\begin{aligned}\mathsf{prune\_left}(x) \quad &: \quad [g, x_l, x_r].\, \{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = (x_l, x_r)\} \\ &\qquad\qquad \{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = (\mathsf{null}, x_r)\} \\ \mathsf{prune\_right}(x) \quad &: \quad [g, x_l, x_r].\, \{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = (x_l, x_r)\} \\ &\qquad\qquad \{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = (x_l, \mathsf{null})\}\end{aligned}$$
The specifications immediately follow from the definition of the underlying actions. The properties involved in the specifications are all stable, as discussed above in the case of try_mark.

3.
$$\begin{aligned}\mathsf{read\_left}(x) \quad &: \quad [g, x_l, x_r].\, \{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = (x_l, x_r)\} \\ &\qquad\qquad \{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = (x_l, x_r) \wedge r = x_l\} \\ \mathsf{read\_right}(x) \quad &: \quad [g, x_l, x_r].\, \{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = (x_l, x_r)\} \\ &\qquad\qquad \{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = (x_l, x_r) \wedge r = x_r\}\end{aligned}$$
The specifications again immediately follow from the definition of the actions, which are all idle. One interesting point is that we require $x \in \mu_s$, even though the action definitions do not require it, hence the actions can be used even without ownership of x. However, without this property, we cannot specify the value of the return result r in a stable manner, because other threads may be able to prune x_l or x_r.

4. We also note at this point that the spec for span is stable. The only non-trivial case is the predicate $\mathsf{subspan}_{\gamma,g,\mu}\ x\ t$, whose stability we consider under the condition that $t \sqsubseteq \mu_s$. This predicate is a conjunction of three predicates: $\mathsf{subtree}\ \gamma\ x\ t$, $\mathsf{front}\ \gamma\ t \sqsubseteq t$, and $\mathsf{front}\ g\ t \sqsubseteq \mu$, each of which is stable under the condition $t \sqsubseteq \mu_s$. Let us see why.
Under interference of other threads, g, t and x are fixed, μ may increase, and some edges in γ may get pruned. However, no edges can be pruned that originate in nodes from μ_s, as *other* threads cannot modify nodes marked by the *self* thread. Thus, in particular, no edges can be pruned that originate in nodes from t.
Under such transformation, it is clear that $\mathsf{subtree}\ \gamma\ x\ t$ preserves its validity. While interference can prune some edges of γ, it cannot prune those out of nodes in t. If nodes in t form a tree in γ, the tree remains untouched by interference. For similar reasons $\mathsf{front}\ \gamma\ t$ remains fixed by interference, and hence $\mathsf{front}\ \gamma\ t \sqsubseteq t$ preserves its validity.
Finally, the fact that $\mathsf{front}\ g\ t \sqsubseteq \mu$ preserve its validity under interference is even simpler, as $\mathsf{front}\ g\ t$ remains fixed, but μ may only increase.

4.2.3 Proof outline

The proof outline in Figure 4.1 is for the most part very straightforward, as it simply follows the specifications of the atomic actions. The interesting points of the proof outline are the two concurrent recursive calls to span in line 15, and the construction of the output tree in line 25.

For the recursive calls, we split the value of $\mu_s = \{x\}$ between the threads, as follows: the threads each receive $\mu_s = \mathsf{empty}$, and $\{x\}$ is given to the frame predicate

$$\mu_s = \{x\} \wedge (x_l, x_r) = \mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x.$$

The frame predicate is required so that we can transfer the equations about x_l and x_r from line 14 to line 16. By giving $\{x\}$ to the μ_s of the frame predicate, we make the equations stable. We have used exactly the same trick in the proof outline for the exchanger, so this move should not be surprising.

Notice that for the recursive calls we also derive $x_l, x_r \in \{\mathsf{null}\} \cup \mathsf{nodes}\ g$, which is not something that we have listed in line 14 preceding the recursive calls. This property, however, is provable from the resource invariant and the facts that $x_l = \mathsf{left}\ \gamma\ x$ and $x_r = \mathsf{right}\ \gamma\ x$. Indeed, the resource invariant explicitly states that child nodes out of any node in the graph, must also be nodes in the graph, or null. Here, we know that

1 $\{\gamma \leqslant_\mu g \wedge \mu_s = \mathsf{empty} \wedge x \in \{\mathsf{null}\} \cup \mathsf{nodes}\ g\}$
2 if $x == \mathsf{null}$ then return false;
3 $\{\gamma \leqslant_\mu g \wedge \mu_s = \mathsf{empty} \wedge x \in \{\mathsf{null}\} \cup \mu \wedge r = \mathsf{false}\}$
4 else
5 $\{\gamma \leqslant_\mu g \wedge \mu_s = \mathsf{empty} \wedge x \in \mathsf{nodes}\ g\}$
6 $b \leftarrow \mathsf{try\_mark}(x)$;
7 $\{\gamma \leqslant_\mu g \wedge \mathsf{if}\ b\ \mathsf{then}\ \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x\ \mathsf{else}\ \mu_s = \mathsf{empty} \wedge x \in \mu\}$
9 if b then
10 $\{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x\}$
11 $x_l \leftarrow \mathsf{read\_left}(x)$;
12 $\{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge \mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x \wedge x_l = \mathsf{left}\ \gamma\ x\}$
13 $x_r \leftarrow \mathsf{read\_right}(x)$;
14 $\{\gamma \leqslant_\mu g \wedge \mu_s = \{x\} \wedge (x_l, x_r) = \mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x\}$
15 $(r_l, r_r) \leftarrow$ $\{\gamma \leqslant_\mu g \wedge \mu_s = \mathsf{empty} \wedge x_l \in \{\mathsf{null}\} \cup \mathsf{nodes}\ g\}$ $(\mathsf{span}(x_l)$ $\{\gamma \leqslant_\mu g \wedge \exists t_l.\, \mu_s = t_l \wedge$ if r_l then $\mathsf{subspan}_{\gamma,g,\mu}\ x_l\ t_l$ else $t_l = \mathsf{empty} \wedge x_l \in \{\mathsf{null}\} \cup \mu\}$ $\parallel$ $\{\gamma \leqslant_\mu g \wedge \mu_s = \mathsf{empty} \wedge x_r \in \{\mathsf{null}\} \cup \mathsf{nodes}\ g\}$ $\mathsf{span}(x_r))$; $\{\gamma \leqslant_\mu g \wedge \exists t_r.\, \mu_s = t_r \wedge$ if r_r then $\mathsf{subspan}_{\gamma,g,\mu}\ x_r\ t_r$ else $t_r = \mathsf{empty} \wedge x_r \in \{\mathsf{null}\} \cup \mu\}$ $\circledast$ $(\mu_s = \{x\} \wedge (x_l, x_r) = \mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x)$ $\circledast$ $(\mu_s = \{x\} \wedge (x_l, x_r) = \mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x)$
16 $\{\gamma \leqslant_\mu g \wedge (x_l, x_r) = \mathsf{ch}\ \gamma\ x = \mathsf{ch}\ g\ x \wedge \exists t_l\ t_r.\, \mu_s = \{x\} \cup t_l \cup t_r \wedge$
17 (if r_l then $\mathsf{subspan}_{\gamma,g,\mu}\ x_l\ t_l$ else $t_l = \mathsf{empty} \wedge x_l \in \{\mathsf{null}\} \cup \mu$) $\wedge$
18 (if r_r then $\mathsf{subspan}_{\gamma,g,\mu}\ x_r\ t_r$ else $t_r = \mathsf{empty} \wedge x_r \in \{\mathsf{null}\} \cup \mu$)$\}$
19 if $\neg r_l$ then $\mathsf{prune\_left}(x)$;
20 if $\neg r_r$ then $\mathsf{prune\_right}(x)$;
21 $\{\gamma \leqslant_\mu g \wedge (x_l, x_r) = \mathsf{ch}\ g\ x \wedge \exists t_l\ t_r.\, \mu_s = \{x\} \cup t_l \cup t_r \wedge$
22 (if r_l then $\mathsf{subspan}_{\gamma,g,\mu}\ x_l\ t_l$ else $t_l = \mathsf{empty} \wedge x_l \in \{\mathsf{null}\} \cup \mu$) $\wedge$
23 (if r_r then $\mathsf{subspan}_{\gamma,g,\mu}\ x_r\ t_r$ else $t_r = \mathsf{empty} \wedge x_r \in \{\mathsf{null}\} \cup \mu$) $\wedge$
24 $\mathsf{ch}\ \gamma\ x = (\mathsf{if}\ r_l\ \mathsf{then}\ x_l\ \mathsf{else}\ \mathsf{null}, \mathsf{if}\ r_r\ \mathsf{then}\ x_r\ \mathsf{else}\ \mathsf{null})\}$
25 $\{\gamma \leqslant_\mu g \wedge \exists t (= \{x\} \cup t_l \cup t_r).\, \mu_s = t \wedge \mathsf{subspan}_{\gamma,g,\mu}\ x\ t\}$
26 $\{\gamma \leqslant_\mu g \wedge \mathsf{subspan}_{\gamma,g,\mu}\ x\ \mu_s\}$
27 return true;
28 else return false;
29 $\{\gamma \leqslant_\mu g \wedge \mathsf{if}\ r\ \mathsf{then}\ \mathsf{subspan}_{\gamma,g,\mu}\ x\ \mu_s\ \mathsf{else}\ \mu_s = \mathsf{empty} \wedge x \in \{\mathsf{null}\} \cup \mu\}$

Figure 4.1: Proof outine for span.

x is a node in the graph, because $x \in \mu_s \sqsubseteq \mathsf{nodes}\ g$, hence x_l and x_r, as children of x, must be in the graph as well (or null).

The preconditions of the recursive call also feature the conjunct $\gamma \leqslant_\mu g$, which is indexed by $\mu = \mu_s \cup \mu_o$. But notice that this quantity is equal in both threads. Even though the sub-components μ_s in the two threads are different, the sub-components μ_o compensate for this difference, as per the definition of $\circledast$ (Chapter 2).

Similarly, in line 16, we revert the process, and just collect the μ_s components from the two threads and the frame, together with the conditionals, which all involve immutable values. For clarity, we introduce existentially-bound variables t_l and t_r to explicitly name the trees collected in the μ_s of the two recursive calls. By naming these trees explicitly, in the rest of the proof, we can track their provenance.

Lines 19 and 20 are not difficult to handle, though one has to introduce framing in order to propagate the properties from lines 16-18 that are not directly given in the precondition of the prune actions. We do not spell out the details, as they are easy to recover, following stability of subspan discussed in the previous section.

Finally, inferring line 25, requires considering four different cases, depending on the possible values of the booleans r_l and r_r. We consider here only the case when r_l and r_r are both true. The other three cases are similar, and simpler.

In the case r_l and r_r are true, the recursive calls returned trees t_l and t_r, rooted in x_l and x_r, respectively. Thus, $x_l \in t_l$ and $x_r \in t_r$. We also know that t_l and t_r are disjoint sets of nodes, because they are marked by two concurrent calls.

The tree that we will return as the result of the enclosing recursive call is $t = \{x\} \cup t_l \cup t_r$. We need to show that

$$\mathsf{subspan}_{\gamma,g,\mu}\ x\ t,$$

that is, $x \neq \mathsf{null}$, $\mathsf{subtree}\ \gamma\ x\ t$, $\mathsf{front}\ \gamma\ t \sqsubseteq t$ and $\mathsf{front}\ g\ t \sqsubseteq \mu$.

The $x \neq \mathsf{null}$ property trivially follows from the fact that $x \in \mu_s$, and is hence a node in $\mathsf{nodes}\ g$ by resource invariant, as we already discussed above, in relation to lines 19 and 20.

The property

$$\mathsf{subtree}\ \gamma\ x\ t$$

says that t is a sub-tree of γ, rooted in x. The fact that x is the root is obvious, since the children of x are x_l and x_r, the roots of t_l and t_r, respectively. In order to show that t is a tree, we also need observe that there are no edges from nodes in t_l to nodes in t_r, and vice versa. This follows from the maximality of the trees t_l and t_r, i.e., from the properties

$$\mathsf{front}\ \gamma\ t_l \sqsubseteq t_l \quad \text{and} \quad \mathsf{front}\ \gamma\ t_r \sqsubseteq t_r$$

present in our initial hypotheses. Indeed, if there existed an edge from t_l to t_r, the front of t_l would include the ending node of that edge, which is a node in t_r. By the hypotheses, such a node would be a node in both t_l and t_r, which is contradicted by the disjointness of these trees.

We also easily obtain that

$$\mathsf{front}\ \gamma\ t \sqsubseteq t.$$

Indeed, by basic graph-theoretic properties, and the assumption of maximality of t_l and t_r in γ, $\mathsf{front}\ \gamma\ t = \mathsf{front}\ \gamma\ (\{x\} \cup t_l \cup t_r) = \mathsf{ch}\ \gamma\ x \cup \mathsf{front}\ \gamma\ t_l \cup \mathsf{front}\ \gamma\ t_r \sqsubseteq \{x_l, x_r\} \cup t_l \cup t_r$. The latter is a subset of $t_l \cup t_r$ since $x_l \in t_l$ and $x_r \in t_r$, because x_l and x_r are the roots, and thus nodes in their respective trees. But $t_l \cup t_r \sqsubseteq t$, by definition, thus giving us the required property.

Finally,

$$\mathsf{front}\ g\ t \sqsubseteq \mu.$$

Similar as above, $\mathsf{front}\ g\ t = \mathsf{front}\ g\ (\{x\} \cup t_l \cup t_r) = \mathsf{ch}\ g\ x \cup \mathsf{front}\ g\ t_l \cup \mathsf{front}\ g\ t_r$. By assumption, the fronts of t_l and t_r are marked, and hence the above quantity is a subset of $\{x_l, x_r\} \cup \mu$. But x_l and x_r are also marked, since they are nodes in t_l and t_r, respectively, and t_l and t_r are subsets of μ_s. Thus, the whole union mast be marked.

Exercise 4.2.1. Specify and verify a sequential union-find algorithm. Can you make use of auxiliary state? Which PCM of auxiliary state is appropriate?

Bibliography

[1] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.

[2] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science (TCS)*, 375(1-3):227–270, 2007.

[3] C. Calcagno, P. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science (TCS)*, 298(3):557–581, Apr. 2003.

[4] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378, 2007.

[5] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifing concurrent C. In *TPHOL*, pages 23–42, 2009.

[6] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, pages 207–231, 2014.

[7] Class `Exchanger<V>`, Java Platform SE 8 Documentation. Available from http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Exchanger.html. Accessed June 24, 2015.

[8] K. Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, Computer Laboratory, 2004. UCAM-CL-TR-579.

[9] N. Hemed and N. Rinetzky. *Brief announcement*: Concurrency-Aware Linearizability. In *PODC*, pages 209–211, 2014.

[10] N. Hemed, N. Rinetzky, and V. Vafeiadis. Modular verification of concurrency-aware linearizability. In *DISC*, pages 371–387, 2015.

[11] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* M. Kaufmann, 2008.

[12] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, 1990.

[13] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[14] C. Hur, D. Dreyer, and V. Vafeiadis. Separation logic in the presence of garbage collection. In *LICS*, pages 247–256, 2011.

[15] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[16] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.

[17] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.

[18] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310, 2014.

[19] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. Extended Version., 2014. Available at http://software.imdea.org/~aleks/fcsl/concurroids-extended.pdf.

[20] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *ICFP*, pages 62–73, 2006.

[21] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240, 2008.

[22] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs.

In *POPL*, pages 261–274, 2010.

[23] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.

[24] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Th. Comp. Sci.*, 375(1-3), 2007.

[25] P. W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). In *Software Safety and Security - Tools for Analysis and Verification*, NATO Science for Peace and Security Series, pages 286–318. IOS Press, 2012.

[26] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[27] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science (MSCS)*, 11(4):511–540, Aug. 2001.

[28] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[29] J. C. Reynolds. An introduction to separation logic. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/www15818As2011/cs818A3-11.html, February 2011. Lecture notes for the course CMU 15-818A3.

[30] W. N. Scherer III, D. Lea, and M. L. Scott. A scalable elimination-based exchange channel. In *OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages*, 2005.

[31] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87, 2015.

[32] I. Sergey, A. Nanevski, and A. Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, pages 333–358, 2015.

[33] I. Sergey, A. Nanevski, A. Banerjee, and G. A. Delbianco. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. arXiv:1509.06220, Sep 2015.

[34] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, pages 149–168, 2014.

[35] R. K. Treiber. Systems programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.