traditionally:
Owicki-Gries/CSL for space (description of state, temporally invariant)
RG/RGSep/SAGL for time (description of interference, spatially invariant=stability?)

subjectivity = RG for space; a building block but not the main idea of this work, should only be a sidenote/not the title

Machines in Interactive Space Time = space AND time spec, composable by interaction between machines

Subjective Concurrent Protocol Logic

lock spec: abstract description of locked/unlocked
lock impl: concrete details about realization in memory/program

what's the problem? what's the solution?

Abstract

vague review of recent past

Modularity is a curse of logic-based verification of programs with shared-memory concurrency. Its challenge is manifold: one should be able to check the safety properties of a thread running in isolation, as well as to specify, what are the "safe" states of a particular data structure and check them separately, abstracting from the implementation of other components of a system. The last decade witnessed a blast of approaches in attempt to tackle the modularity problem by employing different combinations of *concurrent separation logic* for spatial reasoning about safety properties and *rely/guarantee* for specifying temporal invariant of programs running in parallel. Expressive enough, all these approaches pay their price by either dealing with a too low-level model and forcing a client of a logic to give verbose specifications, or by boiling the logical reasoning down to the underlying semantic model.

In this work, we take a different view on the model of concurrent computations and provide a simple yet powerful logical framework, enabling reasoning about concurrently running independent modules *at the level of specifications*. The key novelty is the way to specify the module protocols in the form of communicating state transition systems (STS) that describe valid program states and the transitions the program is allowed to make. spacetime spec

module decomposition = client+libraries, each its own automata/machine

We show how the reasoning in terms of STS makes it simple to crystallise the behavioural invariants of a module, and provide an *entanglement* operator to build large systems from an arbitrary number of modules. Furthermore, our framework naturally incorporates the notion of *scoped* locks, allowing one to retract programs from larger protocols into smaller ones. We have proved the soundness of our logic with respect to the Brookes-style denotational semantics. Finally, we implemented the semantics, metatheory, logic itself and a number of examples as a shallow embedding in Coq.

entanglement is a fuzzy recipe, not a concrete definition

1. Introduction

how's the logic different from SCSL?
just more locks?

One of the biggest challenges of concurrency logics in general, and in logics for verification of shared memory programs in particular, is achieving compositional reasoning ~~in all its forms.~~ There are many facets to compositionality. One may be looking at thread-level compositionality (find who mentioned this before), or ADT-level modularity (CAP, Jacobs-Piessens), or modularity wrt. granularity (fine-grained vs. coarse-grained in CaReSL), etc.

In this paper we consider abstraction and modularity *at the level of specifications*, as means of decomposing proofs about concurrent programs into smaller, manageable, and reusable chunks. To explain these notions in concrete terms, let us consider specific scenarios. sounds like the same boilerplate as usual

In this paper, we take that an essential component of a specification of a concurrent programs is a state transition system (STS), which describes the valid states of the program, and the transitions between the states that the program is allowed to make.

reasoning from lock spec

For example, a mutual exclusion protocol followed by programs that use one lock, may abstractly be represented by an STS in Figure 1(a). The lock itself may be in one of two states (locked or unlocked), but any individual thread t may need to make a finer distinction of the locked states: it may need to differentiate between the lock being owned by t itself, or by some other thread running concurrently with t.

By abstraction at the level of specification in this particular example, we consider the ability of a formal system to represent a mutual exclusion protocol in the form of an STS such as the one in Figure 1(a), while ignoring all the information about the low-level particulars of the locking mechanism (e.g., is it a spin lock, or a ticketed lock, etc.)

exponential complexity of combining systems

As the number of locks that one program uses grows, the complexity of the STS that specifies the program behavior grows exponentially. For example, Figure 1(b) shows a *product* STS for a locking protocol over two independent locks, which squares the number of states and transitions when compared to the STS in Figure 1(a).

By modularity at the level of specification, we consider the ability of a formal system to ignore the those states and transitions of the STS, that may be irrelevant for the verification of a particular program.

injection allows reasoning about the relevant footprint
and embedding in a larger context w more interference

For example, a program using two locks may consist of calls to two functions each of which uses distinct single locks. While the specification for the larger program may require the product STS from Figure 1(b), it should be possible to specify and verify the component functions wrt. the one-lock STS from Figure 1(a), and then *inject* such proofs in the larger contexts of the product STS, without the need for re-verification. inject/retract terminology is obscure

Dually, a program that creates a lock in a scoped manner (that is, allocates a lock, and upon termination, discards it), is internally governed by a locking protocol, but this fact should not be visible from the outside by the programs that run concurrently with it. Thus, it should be possible to *retract* programs from larger protocols into smaller ones, and the fact that some program locally allocates a scoped lock, should not affect the verification of its clients.

this paper's content

In this paper, we propose a very general, yet conceptually very simple logic, Subjective Concurrency Protocol Logic (SCPL), that enables the abstraction modularity of specifications in the above general sense (i.e., not only in the case of mutual exclusion and locking). SCPL follows Hoare- and Concurrent Separation Logic (CSL) traditions for shared memory concurrency: in addition to using STSs for specifying the protocol that the program follows, SCPL also uses pre- and postconditions to identify the STS states in which the program may start and terminate its execution. spacetime, not just CSL

More specifically, we identify a class of properties that an STS's should satisfy in order to support the described specification abstraction and modularity. We call the STS belonging to this class *concurroids*(*AN: Find a better name; this one too similar to hemorrhoids*). ✓ We consider *composing* concurroids by means of *operators* that preserve the concurroid properties.

Concurroids are *communicating* STS's as we endow them with the ability to exchange *ownership* of portions of their states. For example, a concurroid for a lock protecting a shared resource in the RI or CSL style, upon making a transition from an unlocked to a locked state, *gives away* the heap belonging to a shared resource by means of an *external transition* (i.e., a transition towards an

still haven't given a crisp problem
beginning to see a lot of vague terminology
(inject, rectract, concurroid)

injection: extension by orthogonal/noninterfering components
retraction: interference hiding

a less generic term than "STS"?

say sooner that ownership transfer is a local interaction
between private and the relevant library (lock, alloc)

explain connection wrt complementary/dual operations (a and hat-a)
in process calculus: dual ops are actions performed by processes, nondeterministic matching
here: serve to entangle automata, but automata are inert and the semanticist defines which compose

entanglement
bind two external transitions of the parts
into an internal transition of the whole

external machine

outside world). Dually, upon making a transition from a locked to an unlocked state, it *accepts* a heap from the outside world.

A particularly prominent operator for building concurroids is *entanglement*. It combines two smaller concurroid STS's by connecting their dually-polarized external transitions. complementary The entanglement operator is tightly connected to specification-level modularity, as it can be seen as an STS equivalent of separating conjunction how so? "*" from CSL. For example, we illustrate how the usual ownership transfer from CSL, may be represented as a communication between a concurroid for private state, with the concurroid for locking (and/or, allocation).

wrt related work

Of course, we are not the first to consider abstraction and modularity at the specification level, as described above.

Brookes, Abstract SL

When it comes to modularity of specification and reasoning, CSL is our inspiration. However, for CSL, the whole issue of the growing specification complexity, as the number of locks increases, is made non-existent, as the logic *hardwires* a specific mutual exclusion protocol that every program has to respect. The *semantics* of the logic then has to take of the growing number of locks (and vague this has not been trivial, see Brookes), but the program specifications themselves do not specify the locking behavior.

In rely-guarantee approaches (Jones), one *can* express user-level protocols, just like in SCSP. In fact, one may view SCSP specifications as a generalization of rely-guarantee. Where rely-guarantee uses state predicates to specify the transitions and the interference of the program, we use concurroids.

but we're large footprint
use RG space+time to spec whole system, not just the parts where interference happens

LRG, CAP/Views, CaReSL

As concurroids are endowed with much more structure than ordinary state predicates, we believe SCSP enables much higher level of abstraction in specification and reasoning than RG approaches. For example, while Local Rely Guarantee (LRG) has considered modularity at the specification level – where the specifications are ordinary rely and guarantee predicates – it has not been shown to support abstraction. Similarly, while CAP and Views has considered abstraction at the specification level – being able to assign one-and-the-same specification in the form of an abstract predicate to both spin- and ticketed locks - they have not been shown to support modularity (ie., injection and retraction), such as the one we consider here. Similarly, CaReSL has not so far considered retraction, although it too uses STS's for specification (though not with the same algebraic structure of concurroids), but yes injection?

we do...

We believe that one advantage of the higher-level of abstraction afforded by concurroids is that *proofs* of specific programs in SCSP become streamlined and concise (though a formal comparison of proofs with the related approaches seems difficult to carry out presently)(*AN: Hmm, why not; some Views stuff has been done in Coq, e.g., their joins library verification. We don't have man power to pull that off, though*). ✓ To validate this point, we have implemented SCSP as a shallow embedding in Coq, proved all of its meta theory, and carried out the formal verification of a number of example programs.(*AN: Hmm, so what; some point should be made here, but I don't know which.*) ✓ lessons of mechanization? forced to find the right abstractions?

ppr structure

The rest of the paper is structures as follows. In Section 2, we describe the semantics of concurroids, and the related concepts. In Section 3, we illustrate the mathematical structure of *actions* for a given concurroid. Actions are the primitive concurrent operations of lowest granularity (e.g., memory operations, CAS, etc), but they can also manipulate *auxiliary state*. Actions have to satisfy a number of algebraic laws, in order to match composability of concurroids. In Section **??** we describe the inference rules of SCPL, and illustrate injection and retraction. In Section **??** we show how the embedding of SCPL within the type theory of CiC (as implemented in Coq), provides linguistic mechanisms for abstraction. In particular, we show how one can formally ascribe one and the same specification (i.e., type) to two different locking implementations (spin locks and ticketed locks). In Section **??** we illustrate a concurroid for readers-writers (should we?), and for allocation (should we?).

we can kick the crap out of C(R)AP wrt simplicity of:
constructing semantics, proving specs, composing systems with multiple libs
but let's not get stuck on measuring ourselves against them

these aren't ready yet, but will be great for a final version

avoid duplicating terminology: pointer/location/label

2. State, concurroids, entanglement

Main mathematical concepts We use finite maps in many different guises in the formulation of SCLP, so we introduce some common notation for them. component is vague, find another name

Heaps are finite maps from non-null pointers to values of an arbitrary type. *PCM components* are finite maps from labels to values of an arbitrary partial commutative monoid (PCM). *Type components* are finite maps from labels to values of an arbitrary type. Pointers and labels are isomorphic to natural numbers.

confusing name
I thought it'd be:
Map Loc Type,
not:
Map Loc (ex A, A)

We overload the notation $x \mapsto v$ to denote a heap (resp. pcm or type component), containing a single pointer x (resp. label x), storing the value v. If we want to make the type A of v explicit, we write $x \mapsto_A v$. Finite maps can be nested in a stratified manner, e.g., we can have a component storing a heap,

gets ambiguous when 'v' is a type, 'A' is a kind

this is dense, need to say heap is a pcm
where are the pcm's in priv|-> [self|->h, ...]?

$$\mathsf{label} \mapsto_{\mathsf{heap}} (x \mapsto_{\mathsf{nat}} 3),$$

this needs about 1/2column of explanation with examples building up heaps, SJO, modules

but heaps can't store components. We use empty for the empty finite map/heap/component. Given a finite map f, we write $\mathsf{dom}\ f$ for the set of arguments on which f is defined.

PCMs A PCM consists of a ~~type~~ set U, and a partial binary operation $\oplus$ on U with a neutral element 1_U. The operation $\oplus$ is commutative and associative. The PCM also contains a boolean predicate valid why needed? on U, which selects a subset of U. All the elements out of that subset are considered *undefined*. The partiality of $\oplus$ is modeled by having it return an undefined element. use (+,0) or (*,1)

Finite maps (with an addition of an undefined element) form a PCM with the operation of *disjoint union*, which is undefined if the maps overlap on some argument, and with empty as the unit element.

need a less generic name other than "state"

Subjective state and joining The state of a concurroid is divided into three parts, which we call self, joint, and other, part.

The intuition is that when a thread t is ascribed a concurroid as its specification, the self part of the concurroid state is treated by t as a chunk of private state. Other is the chunk state that the thread can't modify, as it is private to the threads running in parallel with t. swap order Joint is the state that is shared by t and other. Thus, the specification via concurroids are always specific to the view-point of the thread being specified, and we will frequently refer to this thread as "self", hmmmm overloading the terminoliogy with state. The totality of all threads running in parallel with the "self" thread, is called "other", again, overloading the terminology. show example <hS | hJ | hO>

We refer to such an approach with 3-way split of the state, and the thread-relative style of specification in terms of "self" and "other" as *subjectivity*, following the previous work on SCSL [**?**].

[1] joint is shared but inaccessible except by transferring into self (or other)

Self and Other are PCM components, and Joint is a type component. huh? Self and Other are PCM components, because the contents of their labels will ~~often~~ have to be *joined* by $\oplus$ in order to describe the global state invariants. We won't need such operation with the Joint component, so we don't impose that its labels must store values that belong to a PCM.

[1] The existing approaches to concurrency logics divide the state into only private and shared portion, and ignore the other portion, as the thread t can't modify it. In SCPL we expose the other part. As argued by the previous work on SCSL, the other part is important for specifying the global invariants of the state. But this point is better made in the related work section.

SCSL: subjectivity as a 2-part (self+other) means to specify auxiliary state
here: 3-way partition to logically distinguish zones of possible interference
Self: thread-private, G=any, R=id
Joint: shared, G=xfer=R
Other: env-private, G=id, R=any
and footprint preserving!

stick to one notation for the 'contents' relation
not both = and |->
consider Reynolds' [l:v] (more compact), although it clashes with the sacrosanct types

need to introduce metavariables for [self=S,...] and for priv|->[...]

consider an abbreviation for self/joint/other map
<S|J|O> := [self: S, joint: J, other: O]

We use the following notation for the describing the 3 components of a state.

need to say that ',' is map union and/or finmap constructor?

$$\begin{array}{l}[\mathit{self} = (\mathsf{label}_{11} \mapsto x_{11}, \mathsf{label}_{12} \mapsto x_{12}, \ldots)\\ \mathit{joint} = \mathsf{label}_{21} \mapsto x_{21}, \mathsf{label}_{22} \mapsto x_{22}, \ldots\\ \mathit{other} = \mathsf{label}_{31} \mapsto x_{31}, \mathsf{label}_{32} \mapsto x_{32}, \ldots]\end{array}$$

parens

where all the labels may be different.

When the components contain a unique label, we use

is this notation necessary? it's going to be confusing to keep track of which is being used

$$\mathsf{label} \mapsto [\mathit{self} = x, \mathit{joint} = y, \mathit{other} = z]$$

maybe introduce a matrix notation

to abbreviate

I call this transposition: FinMap K1 (FinMap K2 V) <-> FinMap K2 (FinMap K1 V)

$$[\mathit{self} = \mathsf{label} \mapsto x, \mathit{joint} = \mathsf{label} \mapsto y, \mathit{other} = \mathit{label} \mapsto z]$$

We use special notation for states where one component contains a single label, but others are empty.

please not more notation, use eg: label |-> (self |-> x) or label |-> <x|1|1>
say that the other self/joint/other labels are implicitly empty wrt the appropriate PCM/type A

$$\begin{array}{rcl}\mathit{label@self} \mapsto x &=& [\mathit{self} = \mathit{label} \mapsto x, \mathit{joint} = \mathit{empty}, \mathit{other} = \mathit{empty}]\\ \mathit{label@joint} \mapsto x &=& [\mathit{self} = \mathit{empty}, \mathit{joint} = \mathit{label} \mapsto x, \mathit{other} = \mathit{empty}]\\ \mathit{label@other} \mapsto x &=& [\mathit{self} = \mathit{empty}, \mathit{joint} = \mathit{empty}, \mathit{other} = \mathit{label} \mapsto x]\end{array}$$

As states are triples of finite maps, they form a PCM with the operation of *pointwise* disjoint union. We are thus justified in denoting this operation with the PCM notation $\oplus$. Similarly, we use 1_{state} (or simply 1 when the PCM is determined from the context), to denote the state consisting of three empty finite maps.

but not when Joint isn't a PCM

Further notation: Given a state s, we write $s.\mathit{self}$, $s.\mathit{joint}$ and $s.\mathit{other}$ for the three components of s. I like this notation, should be introduced with finmaps

split **~~*Zipping,*~~ *zigging, zagging*** In addition to $\oplus$ operation on PCM-components, which joins up the finite maps disjointly, we need an operation $\otimes$ (pronounced zip) which joins the *contents* of equally named labels.

assumes labels are "in order", under what order?

$$\begin{array}{rcl}\mathsf{empty} \otimes \mathsf{empty} &=& \mathsf{empty}\\ (\mathsf{label} \mapsto_U v_1), t_1 \otimes (\mathsf{label} \mapsto_U v_2), t_2 &=& (\mathsf{label} \mapsto v_1 \oplus v_2), t_1 \otimes t_2\end{array}$$

Notice that $t_1 \otimes t_2$ may be undefined if t_1 and t_2 contain different labels, or if some label contains values of different types in t_1 and t_2.

Given a state $s = [\mathit{self} = t_S, \mathit{joint} = t_J, \mathit{other} = t_O]$ and a component t, we define *zigging* $s \lhd t$ and *zagging* $s \rhd t$ of s and t, as follows

alignment to highlight delta

$$\begin{array}{l}s \lhd t = [\mathit{self} = t_s \underline{\otimes t}, \mathit{joint} = t_J, \mathit{other} = t_O]\\ s \rhd t = [\mathit{self} = t_s, \mathit{joint} = t_J, \mathit{other} = t_O \underline{\otimes t}]\end{array}$$

Erasure The split into the various components, and label fields within them is a logical representation of the actual physical state in which the programs are executed. The logical representation contains much more information than needed for execution. For example, some labels may store values that correspond to *auxiliary* state that should be erased before execution. We assume that the labels storing values of non-heap types are auxiliary, and introduce *erasure* operators to remove them: $\lfloor - \rfloor_{\mathsf{comp}}$ which works on components (pcm or type ones), and $\lfloor - \rfloor_{\mathsf{state}}$ which works on states.

the logical partition is auxiliary state, but maybe too shocking to say

aux state needs to be mentioned sooner
only needed for verification,
no relation to "before execution"

give types of erasure functions

$$\begin{array}{rcl}\lfloor \mathsf{empty} \rfloor_{\mathsf{comp}} &=& \mathsf{empty}_{\mathsf{heap}}\\ \lfloor \mathsf{label} \mapsto_A v, t \rfloor_{\mathsf{comp}} &=& (\mathsf{if}\, \underline{A = \mathsf{heap}}\ \mathsf{then}\ v\ \mathsf{else}\ \mathsf{empty}_{\mathsf{heap}}) \oplus \lfloor t \rfloor_{\mathsf{comp}}\end{array}$$

type equality is heavy machinery, can it be avoided?

$$\lfloor [\mathit{self} = t_1, \mathit{joint} = t_2, \mathit{other} = t_3] \rfloor_{\mathsf{state}} = \lfloor t_1 \rfloor_{\mathsf{comp}} \oplus \lfloor t_2 \rfloor_{\mathsf{comp}} \oplus \lfloor t_3 \rfloor_{\mathsf{comp}}$$

When there is no confusion, we will simply write $\lfloor - \rfloor$ without the index.

Prop

Lemma 1. *Let s_1, s_2 be either components or states. Then: $\lfloor s_1 \oplus s_2 \rfloor = \lfloor s_1 \rfloor \oplus \lfloor s_2 \rfloor$ if $\mathsf{valid}(s_1 \oplus s_2)$, and is undefined otherwise.*

We proceed to illustrate the subjective division of state, by presenting two basic concurroids: priv for private state only, and spinlock for spinlocks in the style of CSL (see some figure).

Both concurroids are indexed by a label which identifies their individual states, in a larger, compounded state. Thus, for example, we can have concurroids spinlock l_1 and spinlock l_2, to repent two different spinlocks. first introduce private and lock SJO's, then show that they can be combined by adding the metametalabel

Private state The concurroid priv for private state (with the label p), splits the state as follows:

$$p \mapsto [\mathit{self} = h_{\mathsf{S}}, \mathit{joint} = (), \mathit{other} = h_{\mathsf{O}}]$$

The self and other components are two heaps h_{S} and h_{O}, and the joint component is the (unique) value of the unit type, symbolizing that the concurroid doesn't deal with shared state. The heap h_{S} is the local heap for threads specified by priv, and h_{O} is the heap belonging to the environment.

One condition on the state space of the priv concurroid is that h_{O} and h_{S} are disjoint heaps, i.e., that no location can be private to two different concurrent threads at the same time. Using the notation for PCMs, we write this condition as valid $(h_{\mathsf{S}} \oplus h_{\mathsf{O}})$. for Heap

We denote the set of states of a concurroid by coh (for *coherent* states). When we want to make the concurroid explicit, we index coh with the name of the concurroid, but omit the index when it can be inferred.

In the case of priv, the coherent states are defined as:

introduce all relevant concepts in one go, not interweaving priv/lock

$$\mathsf{coh}_{(\mathsf{priv}\ p)} = \{p \mapsto [\mathit{self} = h_{\mathsf{S}}, \mathit{joint} = (), \mathit{other} = h_{\mathsf{O}}] | \mathsf{valid}\ (h_{\mathsf{S}} \oplus h_{\mathsf{O}})\}$$

Spin lock The concurroid for spin locks (over the label sl) splits the state as follows.

$$\mathit{sl} \mapsto [\mathit{self} = \mathsf{m}_{\mathsf{S}}, \mathit{joint} = (\mathit{lk} \mapsto b) \oplus h_{\mathsf{R}}, \mathit{other} = \mathsf{m}_{\mathsf{O}}]$$

The joint component contains the shared *heap*, consisting of the actual lock address lk which is a pointer storing the boolean b, and the heap h_{R} that the lock protects.

The concurroid encodes the mutual exclusion protocol following the CSL and Owicki-Gries' resource invariant method [?], whereby a thread that acquires the lock also acquires ownership of the heap h_{R}. When a thread releases the lock, the heap h_{R} is moved from the thread's private state, back into the shared portion. When the lock is free, the shared heap h_{R} satisfies a predetermined invariant I. More formally, the components of spinlock state satisfy the property:

$$\mathsf{if}\ b\ \mathsf{then}\ I(h_{\mathsf{R}})\mathsf{else}\ h_{\mathsf{R}} = \mathsf{empty}$$

While the boolean b specifies whether the lock taken or not, it doesn't specify who owns it – the "self" or "other" threads. The fields m_{S} and m_{O} from the self and other components, are *auxiliary state* that provides this information. They can be either Own or ~~NOwn~~. For example, if $\mathsf{m}_{\mathsf{S}} = \mathsf{Own}$ (resp. NOwn), then the "self" thread owns (resp. doesn't own) the lock, and similarly for m_{O}. m_{S} and m_{O} can't both be equal Own at the same time, as the lock can't be owned by two different concurrent threads at the same time. However, if the lock is taken, i.e. $b = \mathit{true}$ than one of m_{S} and m_{O} must be Own.

The type mutex = {Own, NOwn} forms a PCM with the following partial operation which describes *joining* of lock ownership:

$$\begin{array}{rcl}\mathsf{Own} \oplus \mathsf{NOwn} &=& \mathsf{NOwn} \oplus \mathsf{Own} = \mathsf{Own},\\ \mathsf{NOwn} \oplus \mathsf{NOwn} &=& \mathsf{NOwn}\\ \mathsf{Own} \oplus \mathsf{Own} &=& \mathsf{undefined}\end{array}$$

Collecting the described conditions, the coherence set for the spinlock concurroid is formally defined as:

$$\begin{array}{l}\mathsf{coh}_{(\mathsf{spinlock}\ sl)} =\\ \quad \{\mathit{sl} \mapsto [\mathit{self} = \mathsf{m}_{\mathsf{S}}, \mathit{joint} = (\mathit{lk} \mapsto b) \oplus h_{\mathsf{R}}, \mathit{other} = \mathsf{m}_{\mathsf{O}}] \mid\\ \quad\ \mathsf{valid}(\mathsf{m}_{\mathsf{S}} \oplus \mathsf{m}_{\mathsf{O}}) \wedge \mathsf{valid}((\mathit{lk} \mapsto b) \oplus h_{\mathsf{R}}) \wedge\\ \quad\ \mathsf{if}\ b\ \mathsf{then}\ h_{\mathsf{R}} = \mathsf{empty} \wedge \mathsf{m}_{\mathsf{S}} \oplus \mathsf{m}_{\mathsf{O}} = \mathsf{Own}\\ \quad\quad \mathsf{else}\ I\ h_{\mathsf{R}} \wedge \mathsf{m}_{\mathsf{S}} \oplus \mathsf{m}_{\mathsf{O}} = \mathsf{NOwn}\}\end{array}$$

a chockful, can it be any simpler?

coh = cohOwn + cohNOwn

acquire/release shouldn't be used to name transitions (it applies to locks, allocators, any own xfer in general)
it's overloaded and clashes with the act of acquiring/releasing a lock
i don't like send/receive because the modules are not agents (confusing overlap with process calculi)
consider: give/take or something else

Transitions Transitions are relations on states describing how the state of concurroid may change. Each concurroid has three different transitions: (1) internal, (2) acquire, and (3) release transition.

The internal transition describes the modifications to the contents of the concurroid state, such as, e.g. changing the value stored into some pointer of the private state. Acquire and release transitions are indexed by a heap, and describe how the concurroid can receive (dually send) a chunk of heap from/to the outside world.

Transitions of the priv concurroid The internal transition of the priv concurroid allows that the self heap h^1_S in the input state may change to an arbitrary h^2_S in the result state, *as long as h^1_S and h^2_S have the same domains (i.e., contain the same pointers, possibly storing different values)*. The other state must remain unchanged, as that state is private to other threads.

$$\begin{aligned}\mathsf{internal}_{\mathsf{priv}\ p} = \{(&p \mapsto [self = h^1_\mathsf{S}, joint = (), other = h_\mathsf{O}],\\ &p \mapsto [self = h^2_\mathsf{S}, joint = (), other = h_\mathsf{O}]) \mid\\ &\mathsf{dom}\ h^1_\mathsf{S} = \mathsf{dom}\ h^2_\mathsf{S} \wedge\\ &p \mapsto [self = h^i_\mathsf{S}, joint = (), other = h_\mathsf{O}] \in \mathsf{coh}_{\mathsf{priv}\ p}\}\end{aligned}$$

The acquire transition states that priv can receive a heap h, only if h is disjoint from the two heaps already in the concurroid.

align to clarify delta

some way to get valid/coh implicitly?

$$\begin{aligned}\mathsf{acquire}_{\mathsf{priv}\ p}\ h = \{(&p \mapsto [self = h_\mathsf{S}, joint = (), other = h_\mathsf{O}],\\ &p \mapsto [self = h_\mathsf{S} \underline{\oplus h}, joint = (), other = h_\mathsf{O}]) \mid\\ &\underline{\mathsf{valid}}(h_\mathsf{S} \oplus h \oplus h_\mathsf{O}) \wedge\\ &\mathsf{p} \mapsto [self = h_\mathsf{S}, joint = (), other = h_\mathsf{O}] \in \underline{\mathsf{coh}_{\mathsf{priv}\ p}}\}\end{aligned}$$

Dually, the release transitionstates that priv can give away a heap h only if h is contained within its private heap.

$$\begin{aligned}&\mathsf{release}_{\mathsf{priv}\ p}\ h =\\ &\quad\{(p \mapsto [self = h_\mathsf{S} \underline{\oplus h}, joint = (), other = h_\mathsf{O}],\\ &\quad\ \ p \mapsto [self = h_\mathsf{S}, joint = (), other = h_\mathsf{O}]) \mid\\ &\quad\ \ p \mapsto [self = h_\mathsf{S} \oplus h, joint = (), other = h_\mathsf{O}] \in \underline{\mathsf{coh}_{\mathsf{priv}\ p}}\}\end{aligned}$$

In a sense, the acquire and release transitions may be seen as generalization of allocation and deallocation, as they account for a concurroid (and the programs the concurroid may specify) increasing and decreasing the number of pointer in its state. And indeed, SCSP doesn't contain primitive operations for allocation and deallocation, as these will be encoded inside SCSP as interactions of the priv concurroid with another concurroid for allocation. However, acquire and release transitions are much more general.

Transitions of the* spinlock *concurroid For example, we use them to encode that upon locking and unlocking, the lock-protected resource changes ownership, switching from shared to private state, and back. We overloading the transition names, and for the spinlock concurroid, define them as follows.

$$\begin{aligned}&\mathsf{acquire}_{\mathsf{spinlock}\ sl}\ h =\\ &\quad\{(sl \mapsto [self = \mathsf{Own}, joint = lk \mapsto \mathsf{true}, other = \mathsf{NOwn}],\\ &\quad\ \ sl \mapsto [self = \mathsf{NOwn}, joint = lk \mapsto \mathsf{false} \oplus h, other = \mathsf{NOwn}]) \mid\\ &\quad\ \ \mathsf{valid}(lk \mapsto \mathsf{false} \oplus h) \wedge I\ h\}\end{aligned}$$

$$\begin{aligned}&\mathsf{release}_{\mathsf{spinlock}\ sl}\ h =\\ &\quad\{(sl \mapsto [self = \mathsf{NOwn}, joint = lk \mapsto \mathsf{false} \oplus h, other = \mathsf{NOwn}],\\ &\quad\ \ sl \mapsto [self = \mathsf{Own}, joint = lk \mapsto \mathsf{true}, other = \mathsf{NOwn}]) \mid\\ &\quad\ \ \mathsf{valid}(lk \mapsto \mathsf{false} \oplus h) \wedge I\ h\}\end{aligned}$$

In other words, spinlock acquires a heap h only upon unlocking the lock; that is, setting the contents of the lock lk to false and the lock self status $\mathsf{m_S}$ to NOwn. The acquired heap h is disjoint from the lock lk and satisfies the invariant I. Dually, the lock-protected shared heap h is released only upon a succesful locking, which sets the contents of lk to true, and the self lock status $\mathsf{m_S}$ to Own.

The internal transition of spinlock is trivially the identity, symbolizing that lock's heap and auxiliary state can only be changed by means of acquire and release tranitions.

$$\mathsf{internal}_{\mathsf{spinlock}\ sl} = \{(s, s) \mid s \in \mathsf{coh}_{\mathsf{spinlock}\ sl}\}$$

Entanglement Given concurroids V and W, their entaglement $V \bowtie W$ is a new concurroid that unions the states of V and W, while also *connecting* the transitions of V and W of opposite polarity (as we define shortly).

Intiutively, the idea is that $V \bowtie W$ is a product STS for V and W, and is used to specify programs that logically perform an ownership transfer between their components.

For example, in the concurroid $(\mathsf{priv}\ p) \bowtie (\mathsf{spinlock}\ sl)$, the $\mathsf{acquire}h$ transition of priv p will be "connected" with the release h transitions of release h. In particular, upon locking, the heap h will be *transfered* out of the joint portion of spinlock sl, and received into the self component of priv p. Such a connection thereby encodes the logical protocol of Owicki-Gries style resources (also used in CSL), whereby upon locking, the shared heap protected by the lock is transfered into the private ownership of the locking thread.

Entangled states The coherent states of $V \bowtie W$ are those that can be obtained as a union of coherent states of V and of W, *with disjoint erasures*.

$$\mathsf{coh}_{V \bowtie W} = \{s_1 \oplus s_2 \mid s_1 \in \mathsf{coh}_V \wedge s_2 \in \mathsf{coh}_W \wedge \mathsf{valid}\lfloor s_1 \oplus s_2 \rfloor\}$$

By Lemma 1, the condition $\mathsf{valid}\lfloor s_1 \oplus s_2 \rfloor$ implies that $\lfloor s_1 \rfloor$ and $\lfloor s_2 \rfloor$ are disjoint heaps. It also implies that s_1 and s_2 *have disjoint labels*. Indeed, had s_1 and s_2 shared labels, then $s_1 \oplus s_2$ would be undefinded, making $\lfloor s_1 \oplus s_2 \rfloor$ undefined as well, by Lemma 1.

Entangled transitions The internal transition of $V \bowtie W$ is defined as:

$$\begin{aligned}&\mathsf{internal}_{V \bowtie W} = \{(s_1 \oplus s_2, s_1' \oplus s_2') \mid s_1 \oplus s_2, s_1' \oplus s_2' \in \mathsf{coh}_{V \bowtie W} \wedge\\ &\quad (s_1, s_1') \in \mathsf{internal}_V \wedge s_2 = s_2' \vee\\ &\quad s_1 = s_1' \wedge (s_2, s_2') \in \mathsf{internal}_W \vee\\ &\quad \exists h : \mathsf{heap}.\\ &\qquad (s_1, s_1') \in \mathsf{acquire}_V\ h \wedge (s_2, s_2') \in \mathsf{release}_W\ h \vee\\ &\qquad (s_1, s_1') \in \mathsf{release}_V\ h \wedge (s_2, s_2') \in \mathsf{acquire}_W\ h\}\end{aligned}$$

use algebraic definitions:
internal_V|><|W :=
  internal_V |><| coh_W +
  coh_V |><| internal_W
  acq_V(h) |><| rel_W(h)
  rel_V(h) |><| acq_W(h)

The conjunct $s_1 \oplus s_2, s_1' \oplus s_2' \in \mathsf{coh}_{V \bowtie W}$ ensures that the transition only applies to states that belong to the state space of the $V \bowtie W$ STS. After that, $V \bowtie W$ can internally step whenever V steps but W stays idle ($(s_1 \oplus s_1') \in \mathsf{internal}_V \wedge s_2 = s_2'$), or whenever W steps but V stays idle ($s_1 = s_1' \wedge (s_2 \oplus s_2') \in \mathsf{internal}_W$, or when V and W exchange the ownership of some heap h.

The entangled concurroid $V \bowtie W$ should itself have a release and an acquire transition in order to communicate and entangle with additional concurroids. For example, entangling $(\mathsf{priv}\ p) \bowtie (\mathsf{spinlock}\ sl_1)$ gives us a concurroid to describe the behavior of programs with state and *one* spinlock sl_1. If we want to add another spinlock sl_2, we'd have construct $(\mathsf{priv}\ p) \bowtie (\mathsf{spinlock}\ sl_1) \bowtie (\mathsf{spinlock}\ sl_2)$.

In this "iterated" entanglement, we want to avoid that the concurroids for spinlocks sl_1 and sl_2 communicate between themselves. Indeed, we don't want the locking of sl_2 to transfer the ownership of the sl_2's shared heap to sl_1; we want the heap to be transfered to the private state.

To disable such communication, we de-symmetrize the definition of the acquire and release transitions of $V \bowtie W$, so that they inherit the behavior of the acquire and release transitions of V, but ignore those of W. Formally:

entanglement isn't symmetric, it's defined on a case by case basis acq_V*W = acq_V*id_W

$$\mathsf{acquire}_{V \bowtie W}\ h = \{(s_1 \oplus s_2, s_1' \oplus s_2') \mid s_1 \oplus s_2, s_1' \oplus s_2' \in \mathsf{coh}_{V \bowtie W} \wedge (s_1, s_1') \in \mathsf{acquire}_V\ h \wedge s_2 = s_2'\}$$

$$\mathsf{release}_{V \bowtie W}\ h = \{(s_1 \oplus s_2, s_1' \oplus s_2') \mid s_1 \oplus s_2, s_1' \oplus s_2' \in \mathsf{coh}_{V \bowtie W} \wedge (s_1, s_1') \in \mathsf{release}_V\ h \wedge s_2 = s_2'\}$$

Lemma 2 (Reordering properties of $\bowtie$)**.**

$$\begin{aligned} \mathsf{coh}_{V \bowtie W} &= \mathsf{coh}_{W \bowtie V} \\ \mathsf{coh}_{(U \bowtie V) \bowtie W} &= \mathsf{coh}_{U \bowtie (V \bowtie W)} \\ \\ \mathsf{internal}_{V \bowtie W} &= \mathsf{internal}_{W \bowtie V} \\ \mathsf{internal}_{(U \bowtie V) \bowtie W} &= \mathsf{internal}_{(U \bowtie W) \bowtie V} \\ \mathsf{acquire}_{(U \bowtie V) \bowtie W} &= \mathsf{acquire}_{(U \bowtie W) \bowtie V} \\ \mathsf{release}_{(U \bowtie V) \bowtie W} &= \mathsf{release}_{(U \bowtie W) \bowtie V} \end{aligned}$$

2.1 Concurroids abstractly

A concurroid is a tuple $V = (D, \mathsf{coh}, \mathsf{internal}, \mathsf{acquire}, \mathsf{release})$ satsisfying a number of properties (listed below).

D is a set of labels, refered to as the *label footprint* of A, listing the labels that appear in the well-formed states. coh is the set of well-formed states, the transition internal is a relation on states, and so are acquire(h) and release h for every heap h.

coherence-set properties The state sets of a concurroid satisfy the following properties, for every state s.

$$\begin{array}{lcl} \mathsf{coh}_\mathsf{D} & : & s \in \mathsf{coh} \implies \mathsf{dom}(s.\mathit{self}) = \mathsf{dom}(s.\mathit{joint}) = \mathsf{dom}(s.\mathit{other}) = D \\ \mathsf{coh}_\mathsf{H} & : & s \in \mathsf{coh} \implies \mathsf{valid}\lfloor s \rfloor \\ \mathsf{coh}_\mathsf{SO} & : & s \in \mathsf{coh} \implies \mathsf{valid}(s.\mathit{self} \otimes s.\mathit{other}) \\ \mathsf{coh}_\mathsf{Z} & : & \forall t.\, s \triangleleft t \in \mathsf{coh} \iff s \triangleright t \in \mathsf{coh}. \end{array}$$

Properties coh_D and coh_H are straightforward, so we don't discuss them.

coh_SO requires that composition of the self and other view of the concurroid state is well-formed. To illustrate, in the case of the concurroid priv, coh_SO corresponds to the requirement that h_S and h_O are disjoint heaps, i.e. that the private states of parallel threads can't overlap. In the case of the concurroid spinlock, coh_SO corresponds to the requirement that the mutexes $\mathsf{m_S}$ and $\mathsf{m_O}$ are not both Own, i.e. that parallel threads can't simultanously own one and the same lock. In the case of entanglement priv $p \bowtie$ spinlock sl, coh_SO corresponds to the conjunction of the coh_SO properties for the individual concurroids.

coh_Z allows that the boundary between the self and other components can be realigned, as the component t added onto the self (i.e., zigged) can always be moved onto the other (i.e. zagged), and vice-versa. Concretely, coh_Z in the case of priv shows that the state $[\mathit{self} = h_\mathsf{S} \oplus h, \mathit{joint} = (), \mathit{other} = h_\mathsf{O}]$ is coherent iff the state $[\mathit{self} = h_\mathsf{S}, \mathit{joint} = (), \mathit{other} = h_\mathsf{O} + h]$ is coherent too.

The coh_Z property will be important when considering properties of fork-join composition of threads. Following CSL, in SCPL we consider that the private state of a thread is divided among its children upon forking. The left child becomes part of the environment for the right child, and vice versa. Thus, the state assigned to the left child's self component, is zagged onto the right child's other component (and vice versa). coh_Z property thus ensures that upon forking, a thread that is in a coherent state leaves both of its children in coherent states too, albeit the particular alignments of the boundary between self and other may differ among the children themselves and the parent thread.

Transition properties Let G vary over a transitions internal, and acquire h and release h (for every heap h). Then G is required to satisfy the following properties.

$$\begin{array}{lcl} \mathsf{trans_C} & : & (s, s') \in G \implies s \in \mathit{coh} \wedge s' \in \mathit{coh} \\ \mathsf{trans_E} & : & (s, s') \in G \implies s.\mathit{other} = s'.\mathit{other} \\ \mathsf{trans_Z} & : & \forall t.\ s.\mathit{other} = s'.\mathit{other} \implies \\ & & \quad (s \triangleright t, s' \triangleright t) \in G \implies (s \triangleleft t, s' \triangleleft t) \in G \end{array}$$

The $\mathsf{trans_C}$ property restricts G to only relate states that are coherent, that is, they belong to the state space of the STS.

The $\mathsf{trans_E}$ property restricts that G can't modify the other component of the concurroid. Thus, G only bounds how the concurroid can change the self and joint component. In relation to the work on Rely-Guarantee reasoning [**?**, **?**, **?**], the $\mathsf{trans_E}$ property implies that the transitions of concurroids are *guarantee* transitions. Given a concurroid V, the transitions that correspond to rely transitions in Rely-guarantee reasoning (that is, they bound the steps of the environment threads), may be computed as the internal, acquire and release transitions of the *transposed* concurroid $V^\top$, in which the self and other components of the states in the definition of coh set and the relations are interchanged.

The $\mathsf{trans_Z}$ property show shows that a chunk t may be moved from the other components of related states (zagged) into the self components (zigged).

More concretely, in the case of priv after the expansion of the definitions of $\triangleright$ and $\triangleleft$, and some simplification, the $\mathsf{trans_Z}$ property states that, if:

$$\begin{array}{l}([\mathit{self} = h_\mathsf{S}, \mathit{joint} = (), \mathit{other} = h_\mathsf{O} + h], \\ \ [\mathit{self} = h_\mathsf{S}', \mathit{joint} = (), \mathit{other} = h_\mathsf{O} + h]) \in G\end{array}$$

then

$$\begin{array}{l}([\mathit{self} = h_\mathsf{S} \oplus h, \mathit{joint} = (), \mathit{other} = h_\mathsf{O}], \\ \ [\mathit{self} = h_\mathsf{S}' \oplus h, \mathit{joint} = (), \mathit{other} = h_\mathsf{O}]) \in G\end{array}$$

minimal footprint

Thus $\mathsf{trans_Z}$ represents a generalized notion of *locality* of transitions, whereby if a transition relates states with certain self components, then it also relates states in which the self components have been *enlarged* (i.e., framed by) by a fixed quantity.

Whereas related work on separation logic only considers framing by a heap, $\mathsf{trans_Z}$ considers framing by any component.

Whereas in related work the framing heap is materialized out of nowhere, $\mathsf{trans_Z}$ requires that the framing component t is appropriated from the environment; that is, t has to be part of other in the premise of $\mathsf{trans_Z}$, and is removed from other in the conclusion of $\mathsf{trans_Z}$.

Specific properties of transitions Additionally, there's a few properties that only internal transitions, or only acquire and release transitions has to satisfy.

$$\begin{array}{lcl} \mathsf{inter\_refl} & : & s \in \mathsf{coh} \implies (s, s) \in \mathsf{internal}_A \\ \mathsf{inter\_foot} & : & (s, s') \in \mathsf{internal}_A\ h \implies \\ & & \quad \mathsf{valid}\lfloor s \rfloor = \mathsf{valid}\lfloor s' \rfloor \wedge \\ & & \quad \mathsf{dom}\lfloor s \rfloor = \mathsf{dom}\lfloor s' \rfloor \\ \mathsf{acq\_foot} & : & \forall h{:}\mathsf{heap}.\ (s, s') \in \mathsf{acquire}_A\ h \implies \\ & & \qquad \mathsf{valid}(\lfloor s \rfloor \oplus h) = \mathsf{valid}\lfloor s' \rfloor \wedge \\ & & \qquad \mathsf{dom}(\lfloor s \rfloor \oplus h) = \mathsf{dom}\lfloor s' \rfloor \\ \mathsf{rel\_foot} & : & \forall h{:}\mathsf{heap}.\ (s, s') \in \mathsf{release}_A\ h \implies \\ & & \qquad \mathsf{valid}\lfloor s \rfloor = \mathsf{valid}(h \oplus \lfloor s' \rfloor) \wedge \\ & & \qquad \mathsf{dom}\lfloor s \rfloor = \mathsf{dom}(h \oplus \lfloor s' \rfloor) \end{array}$$

inter_refl shows that the internal transition is reflexive, thus allowing the programs specified by the concurroid to be *idle*, that is, transition to their initial state.

inter_foot requires that internal transition preserves the heap obtained by erasure of states. Related states may move the ownership of various heaps between self and joint components; they may also change the auxiliary state, and change the *contents* of individual heap pointers. However, they can't add new pointers or remove old ones. Adding new pointers or removing old ones is the job of acquire and release transitions, as formalized by acq_foot and rel_foot properties.

Lemma 3 (Entanglement)**.** *If V and W are concurroids, then $V \bowtie W$ is a concurroid as well, whose label footprint is the union of the label footprints of V and W.*

3. Actions

Actions are the primive objects for programming in the language of SCSP. They perform *atomic* steps from state to state, and have the lowest granularity. An action can operationally be thought of as a single memory operation, such as e.g., reading from, writing into, or CAS-ing over a heap cell. An action may also be idle. Additionally, actions may perform return a value of some given type A, as well as manipule the auxiliary state, and perform ownership transfer (i.e., reallign the boundaries between self, joint and other components).

An action is always associated with some concurroid V in the following sense. If an action modifies state s into state s', then $(s, s') \in \mathsf{internal}_A$.[2] We use the typing judgmnet $a : \mathsf{action}\ V\ A$, to denote that the action a is associated with the concurroid V, and returns a value of type A.

We denotationly model an action a by a pair of components satisfying a number of properties (to be described later). The components are: (1) the predicate safe_a (on states) describing the states in which the action is safe to execute, and (2) the relation step_a relating the initial state, the ending state, and the ending result of a.

To illustrate, we present in more detail the action write $(x : \mathsf{ptr})\ (v : A)$ (associated with priv p concurroid), which writes v into the pointer x, and the action trylock with no arguments (associated with priv $p \bowtie$ spinlock sl concurroid), which tries to acquire the spinlock of the concurroid spinlock sl, and transfer the heap protected by the spinlock into the ownership of priv p.

Let $V = \mathsf{priv}\ p$. The components of write $x\ v : \mathsf{action}\ V\ \mathsf{unit}$ action are:

$$\begin{aligned}\mathsf{safe}_{\mathsf{write}\ x\ v} &= \{s | s \in \mathsf{coh}_V \wedge \exists w : B.\, h : \mathsf{heap}.\, s.\mathit{self} = p \mapsto (x \mapsto w \oplus h)\}\\ \mathsf{step}_{\mathsf{write}\ x\ v} &= \{(s, s', ()) | s, s' \in \mathsf{coh}_V \wedge \exists w\ h.\\ &\qquad s.\mathit{self} = p \mapsto (x \mapsto w) \oplus h \wedge\\ &\qquad s'.\mathit{self} = p \mapsto (x \mapsto v) \oplus h\}\end{aligned}$$

The write $x\ v$ action is safe if the pointer x exists in the private heap of the concurroid, storing some value w of arbitary type B. The step of the action is to replace w with v, while leaving the rest h of the private heap unchanged.

Let $W = \mathsf{priv}\ p \bowtie \mathsf{spinlock}\ sl$. The components of trylock : action W bool are:

$$\begin{aligned}&\mathsf{safe}_{\mathsf{trylock}} = \mathsf{coh}_W\\ &\mathsf{step}_{\mathsf{trylock}} = \{(s, s', r) | s, s' \in \mathsf{coh}_W \wedge \exists h_S\ h_O\ \mathsf{m_S}\ \mathsf{m_O}\ b\ h.\\ &\quad s = p \mapsto [\mathit{self} = h_S, \mathit{joint} = (), \mathit{other} = h_O] \oplus\\ &\qquad sl \mapsto [\mathit{self} = \mathsf{m_S}, \mathit{joint} = lk \mapsto b \oplus h, \mathit{other} = \mathsf{m_O}] \wedge\\ &\quad \text{if } b \text{ then } r = \mathsf{false} \wedge s' = s\\ &\quad \text{else } r = \mathsf{true} \wedge\\ &\qquad s' = p \mapsto [\mathit{self} = h_S \oplus h, \mathit{joint} = (), \mathit{other} = h_O] \oplus\\ &\qquad\quad sl \mapsto [\mathit{self} = \mathsf{Own}, \mathit{joint} = lk \mapsto \mathsf{true}, \mathit{other} = \mathsf{NOwn}]\}\end{aligned}$$

The intuition is that trylock is a denotational encoding of the behavior of the CAS operation. The step relation specifies that trylock inspects the lock implemented as a pointer lk. If lk is false (representing unlocked), the action atomically tries to change lk to true (representing locked). The return value $r = \mathsf{true}$ indicates succesful locking, and the output state s' is changed by transfering the ownership of the heap h from the joint to the self component. If lk is initially true, then trylock terminates without changing anything.

The action is safe to perform on all coherent states of W, as each such states contains the pointer lk representing the lock.

Actions abstractly The components of the action $a : \mathsf{action} V A$ satisfy the following properties.

$$\begin{aligned}\mathsf{safe_C} &: s \in \mathsf{safe}_a \implies s \in \mathsf{coh}_V\\ \mathsf{safe_Z} &: s \rhd t \in \mathsf{safe}_a \implies s \lhd t \in \mathsf{safe}_a\\ \mathsf{step_S} &: (s, s', v) \in \mathsf{step}_a \implies s \in \mathsf{safe}_a\\ \mathsf{step_T} &: (s, s', v) \in \mathsf{step}_a \implies (s, s') \in \mathsf{internal}_V\\ \mathsf{step_F} &: s \rhd t \in \mathsf{safe}_a \implies\\ &\quad (s \lhd t, s', v) \in \mathsf{step}_a \implies\\ &\quad \exists s''.\, s' = s'' \lhd t \wedge (s \rhd t, s'' \rhd t, v) \in \mathsf{step}_a\\ \mathsf{step_E} &: \mathsf{valid}(\lfloor s \rfloor \oplus h) \implies\\ &\quad \lfloor s \rfloor \oplus h = \lfloor s' \rfloor \oplus h' \implies\\ &\quad (s, t, v) \in \mathsf{step}_a \implies (s', t', v') \in \mathsf{step}_a \implies\\ &\quad v = v' \wedge \lfloor t \rfloor \oplus h = \lfloor t' \rfloor \oplus h'\end{aligned}$$

The properties $\mathsf{safe_C}$, $\mathsf{step_S}$, and and $\mathsf{step_T}$ are straightforward.

$\mathsf{safe_Z}$ states that if the action is safe in a state with a smaller self component (because the other component is enlarged by t), the action is also safe if we increase the self component by t. This property corresponds to *safety monotonicity* in abstract separation logic.

The $\mathsf{step_F}$ property says that if a steps in a state with a large self component $s \lhd t$, but is already safe to step in a state with a smaller self component $s \rhd t$, then the result state and value obtained by stepping in $s \lhd t$, can be obtained by stepping in $s \rhd t$, and moving t afterwards. This property corresponds to *framing property* in abstract separation logic. [3]

The property step_E shows that the behavior of the action on the concrete input state obtained after erasing the auxiliary fields and the logical partition, doesn't depend on the erased auxiliary fields and the logical partition. In other words, if the input state have *compatible* erasures (that is, erasures which are subheaps of a common heap), then executing the action in the two states results in equal values, and final states that also have compatible erasures. This is a standard property proved in concurrency logics that deal with auxiliary state and code [**?**, **?**].[4]

I probably need a different set of conjunctions and quantifiers to separate operations over predicates from ordinary conjunction, disjunction and existentials used in the semantics.

[2] In our formalization, we also have actions that are associated with acquire and release transitions. However, these are not used for programming, but only for building other internal-related actions. Thus, we don't present them here.

[3] In addition to $\mathsf{step_F}$, we have been proving an additional property for each action: but all ops are deterministic

$$\mathsf{step_Z} : s.\mathit{other} = s'.\mathit{other} \implies (s \rhd t, s' \rhd t, v) \in \mathsf{step}_a \implies (s \lhd t, s' \lhd t, v) \in \mathsf{step}_a$$

This property corresponds to what we have taken up calling *locality*, though it's not at all equivalent to locality in the sense of abstract separation logic. The property is *not* provable from the rest, unless we require that each action is deterministic (which right now seems unecessary). Thus, if we want it, it has to be required as primitive, which is what we have done.
However, I just realized that the property is not used *anywhere* in the development. It's unused in the model, and it's unused in the always lemmas. Thus, I'm suppressing it here.
On the other hand, the similar locality property for *transitions* is useful in the always lemmas, as alredy commented!

[4] Though, these other papers don't deal with compatibility, as far as I can tell.

Notes:

- Is the frame rule questionable? It hasn't been done in Coq yet, but it sounds right, as a specialization of the par rule. Basically, r only frames the existing labels, and no need to check for any stability or anything.
- The rule for retraction will be hard to nail, mainly because the side-conditions that we have are all done *in the semantics*. Showing them as such may look like it breaks the fiction that our assertion logic is simple. It basically may have to be admitted that the assertion logic is full-blown CiC. But how, then, to deal with reviewers who like Brocolli?
- The rule for injection seems right. But we need the $\oplus$ operator at the level of predicates. In Coq, we don't have the framing predicate r explicitly, since we work with states directly. But then, we have to stabilize by requiring that the framing state is changed by environment stepping on W only.
- The rule for actions can be made better. First, maybe I want to redo the action section to present actions a bit differently. Instead of safe and step being components of the action, I could make them specifications of the actions (then denotationally, we model the action as precisely this pair). In fact, instead of safe and step, I could call them pre and post. I should demand that they are stable under V, something I haven't yet done in Coq.

  I will also need a judgment for Hoare typing of actions. But this is a bit annoying, as it forces us to make a connection between logical variables and unary posts on one side, with binary nature of step.

  However, the benefit is that the rule for actions just switches judgments.

$$\frac{\Gamma \vdash \{p\} a : \mathsf{action}\ V\ A\{q\}}{\Gamma \vdash \{p\}\mathsf{act}\ a : A\{q\}\ @\ V}\ \textsc{Action}$$

- The current rule for actions, actually confuses syntactic predicates over states, with semantic predicates over states. In reality in Coq, there's no confusion, since the two things are equivalent by shallow embedding, but this will cause trouble with the reviewers. So some alternative has to be figured out, presumably one that uses $\models$ and broccoli.
- We need to emphasize that the only time we have to reason about stability is when we declare the actions, and when we do injection. This is much simpler than HOCAP, which seems it needs to reason about a number of auxiliary judgments such as dependency, stability, etc.!!!

  Essentially, in most of the cases the steps of the environment are taken care of by the definition of the Hoare type. Thus, if p and q are not already stable under the environment steps, we won't even be able to establish the judgments in the premisses of the rules.

  So we need the stability conditions only when we are not having any premisses – namely in the rules for primitive actions, and in the rule for injection.

$$\frac{\Gamma \vdash \{p_1\}C : A\{q_1\}\ @\ V \qquad \Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2)}{\Gamma \vdash \{p_2\}C : A\{q_2\}\ @\ V}\ \textsc{Conseq}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{e = \mathsf{true} \wedge p\}C_1 : A\{q\}\ @\ V \\ \Gamma \vdash \{e = \mathsf{false} \wedge p\}C_2 : A\{q\}\ @\ V\end{array}}{\Gamma \vdash \{p\}\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : A\{q\}\ @\ V}\ \textsc{If}$$

$$\frac{\Gamma \vdash \{p_1\}C : A\{q_1\}\ @\ V \qquad \Gamma \vdash \{p_2\}C : A\{q_2\}\ @\ V}{\Gamma \vdash \{p_1 \wedge p_2\}C : A\{q_1 \wedge q_2\}\ @\ V}\ \textsc{Conj}$$

$$\frac{\Gamma \vdash \{p\}C : A\{q\}\ @\ V \qquad \alpha \notin \mathsf{dom}\ \Gamma}{\Gamma \vdash \{\exists \alpha{:}B.\, p\}C : A\{\exists \alpha{:}B.\, q\}\ @\ V}\ \textsc{Exist}$$

$$\frac{\Gamma \vdash \{p\}C : A\{q\}\ @\ V}{\Gamma \vdash \{p \circledast r\}C : A\{q \circledast r\}\ @\ V}\ \textsc{Frame}$$

$$\frac{\forall x{:}B.\, \{p\}f(x) : A\{q\}\ @\ V \in \Gamma}{\Gamma \vdash \forall x{:}B.\, \{p\}f(x) : A\{q\}\ @\ V}\ \textsc{Hyp}$$

$$\frac{\Gamma, \forall x{:}B.\, \{p\}f(x) : A\{q\}\ @\ V, x{:}B \vdash \{p\}C : A\{q\}\ @\ V}{\Gamma \vdash \forall x{:}B.\, \{p\}(\mathsf{fix}\ f.\, x.\, C)(x) : A\{q\}\ @\ V}\ \textsc{Fix}$$

$$\frac{\Gamma \vdash \forall x{:}B.\, \{p\}F(x) : A\{q\}\ @\ V \qquad \Gamma \vdash e : B}{\Gamma \vdash \{[e/x]p\}F(e) : A\{[e/x]q\}\ @\ V}\ \textsc{App}$$

$$\frac{\Gamma \vdash \{p\}C : A\{q\}\ @\ V \qquad r\ \text{stable under}\ W}{\Gamma \vdash \{p \oplus r\}\mathsf{inject}\ C : A\{q \oplus r\}\ @\ V \bowtie W}\ \textsc{Inject}$$

$$\frac{\Gamma \vdash \{blah\}C : A\{blah\}\ @\ V \bowtie W \qquad \text{side conditions}}{\Gamma \vdash \{p\}\mathsf{retract}\ S\ \mathsf{as}\ \mathsf{ao}\ C : A\{q\}\ @\ V}\ \textsc{Retract}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \mathsf{action}\ V\ A \\ \Gamma \vdash (\mathsf{safe}_a \wedge \mathsf{this}\ i, \mathsf{fun}\ m.\, (i, m, \mathsf{res}) \in \mathsf{step}_a) \sqsubseteq (p, q)\end{array}}{\Gamma \vdash \{p\}\mathsf{act}\ a : A\{q\}\ @\ V}\ \textsc{Action}$$

References