Views: Compositional Reasoning for Concurrent Programs (Draft Extended Version)

Thomas Dinsdale-Young Lars Birkedal Philippa Gardner
Matthew Parkinson Hongseok Yang

January 23, 2012

Abstract

We present a framework for reasoning compositionally about concurrent programs. At its core is the notion of a *view*: an abstraction of the state that takes account of the possible interference due to other threads. Threads' views are composable, and an update to the state by one thread must preserve the views of other threads. We prove soundness for our framework, and demonstrate its utility by studying examples. In particular, we show that several variants of concurrent separation logic, type systems, and concurrent abstract predicates can all be seen as instantiations of our framework. Soundness for particular instantiations simply follows from the soundness of our framework.

1 Introduction

There has been a recent flurry of research on models, logics and type systems for reasoning modularly about programs with dynamically allocated mutable state. There have been type systems extended with linear types [1, 2, 3] and related capability systems [4] that enforce a mixture of local and global properties. In particular, they allow for so-called strong update, where an assignment to a variable can change the type of the variable. On the program logic side there have been a series of logics extending separation logic [5] to reason about various notions of sharing. In the sequential setting, the antiframe rule [6, 7] and higher-order frame rules [8] allow for the preservation of selected global properties. For concurrent languages, SAGL [9], RGSep [10], logics for storable locks [11, 12, 13], deny-guarantee [14] and concurrent abstract predicates [15, 16] have been proposed to reason about shared usages of mutable state. There has even been a combination of separation logic with a type system [17].

We present a framework for reasoning compositionally about concurrent programs, which unifies all these styles of reasoning. Our fundamental idea is that threads have different *views* of the machine. Intuitively, a thread's view consists of information about the current state of the machine, the *right* of the thread to modify the state as long as the environment's view is stable (invariant) with respect to such changes, and the thread's *right* to the stability of its own view with respect to changes being made by the environment. Our framework captures the subtle interplay between a thread's view and the environment's view, and allows us to express when views are compatible and hence can be *composed*. This compositional structure of views has close connections with the structure of BI-algebras [18, 19], from which separation logic evolved. One may worry whether our views framework is more complicated than the earlier concrete models that it unifies. As we shall see, that is not the case. Indeed, because our views framework abstracts and distills the essence of compositional concurrent reasoning, the framework itself is arguably a lot simpler than most of the recent advanced concrete models.

Examples Views enable us to reason about a wide range of type systems and program logics. For example, with a standard type system, a view of a thread might be: a variable x has type τ, the memory is well-typed, and the thread has the right to make any type-preserving changes to the memory (including allocation of new memory cells). The environment must hold a compatible view: it must agree with the typing of x. The environment's view also permits it to make type-preserving changes. The thread and environment's views are stable with respect to such changes, since the changes only make the typing information grow monotonically. We are also able to handle non-monotonic type systems, in which updates may change the type of variables, by simply requiring more separation between the thread and the environment's view of the variables and memory.

Another example is disjoint concurrent separation logic where a view might be: location 10 has value 4, and the thread owns location 10. Since the thread owns location 10, the environment cannot own it. Consequently, the thread has the right to change it, but the environment does not, and cannot depend upon it remaining unchanged. Conversely, the thread has no right to change any locations in the environment's view, as they must be separate from the thread's view. The thread and environment's views are thus stable, and two views can be composed provided they are disjoint. Following the original work on concurrent separation logic, there have been several proposals for reasoning about shared mutable state, for example, separation logic with permissions, SAGL, and deny-guarantee. We handle these examples by simply allowing more sharing in a thread's view.

Views A *view* of a thread provides a *partial, abstract* description of the concrete state of a machine. It is partial in that it only describes the state accessed by the thread, such as the cell with location 10. It is abstract in that the verifier can use any additional information to help with their reasoning, such as types, ghost state or permissions. Such additional information has no representation in the concrete state but is a useful fiction for the verifier. Since views describe partial state, an essential property of views is that they are *compositional.* This means that the views of two threads must either be disjoint (e.g., disjoint heaps in separation logic) or must agree on any overlap (e.g., consistent on their types).

The verifier has the flexibility to change the view, using an *angelic preorder,* which does not change the underlying concrete state but does change the abstract state: for example, types can change using a monotonic subtyping relation and permissions can be reassigned across abstract states.

Program Logic We provide a *program logic* for a simple concurrent programming language, parameterised by a set of atomic commands. The views provide the pre- and postconditions of the commands. The logic is parameterised by axioms for the atomic commands and the structural rules include a frame rule and an angelic consequence rule. We present several example program logics using this framework, including weak and strong type theories, several concurrent separation logics, and a combination of separation logic with type theory.

Soundness We provide a *soundness* result, relating our program logic to the underlying operational semantics. The operational semantics is defined on concrete machine states, parameterised by an interpretation of the atomic commands. To relate the program logic to the operational semantics, we require that any abstract view can be *erased* to a set of concrete machine states. This erasure function satisfies two properties: *monotonicity*, declaring that the angelic preorder is erased to the subset relation on sets of machine states; and *atomic soundness*, declaring that the view axiomatisation of atomic commands fits with the interpretation of the atomic commands in the operational semantics. With these properties, we prove a general soundness result, using a technique first introduced by Vafeiadis [20]. To prove soundness for specific type theories and separation logics, it is then enough to display an erasure function satisfying monotonicity and atomic soundness.

Interference Using the constructions presented so far, we can reason about several type theories and concurrent separation logics. These examples have the common pattern that race conditions can be prevented by separation (composition) and abstraction. However, many examples have finer race conditions than this, with a subtle interplay between the effect of the current thread and the effect of the environment. To reason about such behaviour, we need a notion of *interference*: the interference of the environment on a thread's view, and the interference of the thread on the environment's view.

We define an *interference operator* R on views, with simple properties capturing the subtle interplay between the thread's and environment's view. The key properties are that R provides the minimal *stabilisation* of a view with respect to the environment, and that it is well-behaved with respect to *composition*: that is, interference on the composition of views does no more than the interference on the component views. For Kripke models of type theories, the interference operator corresponds to the future world relation of the Kripke model, and thus it allows all type-preserving changes. With disjoint concurrent separation logic, the interference operation is the identity since the environment cannot change a thread's portion of the heap. With complex concurrent separation logics, the interference relation is constructed from the permissions the environment has to change a thread's view.

The interplay between compositionality and interference, plus stabilty, leads to the surprising property that the subset of stable views satisfies all the properties of a view model. Thus, interference operators provide a way of constructing views which allow more fine-grained races between threads. Interference operators do not change the underlying theory, and our general soundness result still applies. We thus believe that views capture the fundamental interplay between composition and interference that is inherent in concurrency.

2 Program Logic

We present a program logic for a concurrent programming language, parameterised by atomic commands, whose assertions are views. We give some simple illustrative examples here, and more complicated examples in section 4.3.

2.1 Concurrent Programming Language

Our language is built from standard composite commands, and parameterised by a set of atomic commands.

Parameter 1 (Atomic Commands)**.** *Assume a countable set of (syntactic) atomic commands* Atom*, ranged over by* a*.*

Definition 1 (Language Syntax)**.** *The set of (syntactic) commands,* Comm*, ranged over by* C*, is defined by the following grammar:*

$$C ::= a \mid \texttt{skip} \mid C;C \mid C+C \mid C \parallel C \mid C^*.$$

2.2 Views

Views form assertions in our program logic. Their properties are axiomatised in our definition of a view model.

Definition 2 (View Model)**.** *A* view model *consists of a set* View*, ranged over by* p, q, r*, with the following structure:*

- *a commutative monoid* $(\mathsf{View}, *, u)$*,*
- *a complete join-semilattice* $(\mathsf{View}, \vDash, \bigvee, \bot)$*, and*

it is subject to the following law:

- Distributivity: *the* $*$ *operator distributes over* $\bigvee$*: that is,*

$$p * \bigvee \{q_i\}_{i \in I} = \bigvee \{p * q_i\}_{i \in I} .$$

Lemma 1. *With respect to* $\vDash$*, the* $*$ *operator is monotone in both of its arguments.*

Remark. The definition of a view model implies additional structure that is not part of the definition. Since views form a complete join-semilattice, they form a complete lattice:

$$\bigwedge \{p_i\}_{i \in I} \overset{\text{def}}{=} \bigvee \{q \mid \forall i.\, q \vDash p_i\} .$$

Also, the distributivity property implies that the composition $*$ is a residuated monoid, with $q * -$ having right adjoint $q \mathrel{-\!\!*} -$: $q \mathrel{-\!\!*} r \overset{\text{def}}{=} \bigvee \{p \mid q * p \vDash r\}$. The adjunction $q * p \vDash r \iff p \vDash q \mathrel{-\!\!*} r$ is trivial from left to right, and follows from distributivity and monotonicity from right to left. The corresponding distributivity property may not hold for $\wedge$, and so a view model is not a Heyting algebra (it does not necessarily have an implication, $\rightarrow$). With implication, a view model is a BI-algebra [18]. Conversely, a BI-algebra is a view model if it is complete as a lattice.

An important class of BI-algebras arises from *separation algebras*, a concept introduced by Calcagno, O'Hearn and Yang [21] to generalise separation logic. We use a variant of separation algebras with multiple units [22, 23] and no cancellativity as a common source of view models.

Definition 3 (Separation Algebra). *A* separation algebra $(\mathcal{M}, \bullet, I)$ *is a partial, commutative monoid with multiple units. Namely, it is a set $\mathcal{M}$ equipped with a partial binary operator $\bullet : \mathcal{M} \times \mathcal{M} \rightharpoonup \mathcal{M}$ and designated unit set $I \subseteq \mathcal{M}$ satisfying:*

- Commutativity: $m_1 \bullet m_2 = m_2 \bullet m_1$ *when either is defined;*
- Associativity: $m_1 \bullet (m_2 \bullet m_3) = (m_1 \bullet m_2) \bullet m_3$ *when either is defined;*
- Existence of Unit: *for all $m \in \mathcal{M}$ there exists $i \in I$ such that $i \bullet m = m$; and*
- Minimality of Unit: *for all $m \in \mathcal{M}$ and $i \in I$, if $i \bullet m$ is defined then $i \bullet m = m$.*

Definition 4 (Separation View Model). *Each separation algebra $(\mathcal{M}, \bullet, I)$ induces a* separation view model $(\mathcal{P}(\mathcal{M}), *, I, \subseteq, \bigcup, \emptyset)$*, where*

$$p_1 * p_2 \stackrel{\text{def}}{=} \{m_1 \bullet m_2 \mid m_1 \in p_1, m_2 \in p_2\} .$$

2.3 Angelic Preorder

Views provide an partial, abstract description of the concrete state of the machine. The angelic preorder enables the verifier to change the instrumentation (e.g. weakening the type of a variable to a supertype, reassigning permissions), as long as the underlying concrete state remains the same.

Definition 5 (Angelic Preorder). *Given a view model,* View*, an* angelic preorder $\prec \subseteq \mathsf{View} \times \mathsf{View}$ *is a reflexive, transitive relation on* View *that is subject to the following laws:*

- Locality: *for all $p, q, r \in \mathsf{View}$, if $p \prec q$ then $p * r \prec q * r$*
- $\vDash$-closure: $\vDash \subseteq \prec$
- $\bigvee$-closure: *if $p_i \prec q$ for all $i \in I$ then $\bigvee \{p_i\}_{i \in I} \prec q$.*

For any view model, $\vDash$ is always an angelic preorder (by Lemma 1), and is inherently minimal with respect to all possible angelic preorders on the view model.

2.4 Program Logic

We define a *program logic* for our programming language, in which views provide the pre- and postconditions of the commands. We declare axioms for the atomic commands, and rules for composition (frame and disjoint concurrency) and angelic consequence.

Parameter 2 (View Model). *Assume a view model*

$$(\mathsf{View}, *, u, \vDash, \bigvee, \bot).$$

Parameter 3 (Angelic Preorder). *Assume an angelic preorder on* View, $\prec$.

Parameter 4 (Axiomatisation). *Assume a set of axioms* $\mathsf{Axiom} \subseteq \mathsf{View} \times \mathsf{Atom} \times \mathsf{View}$.

Definition 6 (Program Logic). *The program logic consists of judgements of the form $\{p\}\ C\ \{q\}$, where $p, q \in$* View *provide the precondition and postcondition of command $C \in$* Comm*. The proof rules for these judgements are given in Fig. 1.*

$$\frac{(p, a, q) \in \mathsf{Axiom}}{\{p\}\ a\ \{q\}} \qquad \frac{p \prec p' \quad \{p'\}\ C\ \{q'\} \quad q' \prec q}{\{p\}\ C\ \{q\}}$$

$$\frac{\{p\}\ C\ \{q\}}{\{p * r\}\ C\ \{q * r\}} \qquad \frac{\forall i \in I.\, \{p_i\}\ C\ \{q\}}{\{\bigvee \{p_i\}_{i \in I}\}\ C\ \{q\}} \qquad \frac{}{\{p\}\ \mathtt{skip}\ \{p\}}$$

$$\frac{\{p\}\ C_1\ \{r\} \quad \{r\}\ C_2\ \{q\}}{\{p\}\ C_1; C_2\ \{q\}} \qquad \frac{\{p\}\ C_1\ \{q\} \quad \{p\}\ C_2\ \{q\}}{\{p\}\ C_1 + C_2\ \{q\}}$$

$$\frac{\{p_1\}\ C_1\ \{q_1\} \quad \{p_2\}\ C_2\ \{q_2\}}{\{p_1 * p_2\}\ C_1 \parallel C_2\ \{q_1 * q_2\}} \qquad \frac{\{p\}\ C\ \{p\}}{\{p\}\ C^*\ \{p\}}$$

Figure 1: The Proof Rules for our Program Logic.

The intended semantics of $\{p\}\ C\ \{q\}$ is that if the program C is run to termination from an initial state that is described by the view p, then the resulting state will be described by the view q. This is a partial correctness interpretation: the judgements say nothing about non-terminating executions.

Most of the proof rules are standard rules from disjoint concurrent separation logic. They include the frame rule, which captures the intuition that a program's view can be extended with a composable view, and the disjoint concurrency rule, which allows the views of two threads to be composed. The axiom rules are self-explanatory. The rule for the angelic preorder is a modified version of the standard rule of consequence, which it subsumes. It enables the verifier to modify the instrumentation without performing a machine-level update.

2.5 Examples

We illustrate our program logic with some simple examples. These examples build on a language whose atomic primitives are heap update commands.

Definition 7 (Atomic Heap Commands)**.** *Assume a set of variable names* Var*, ranged over by* $\boldsymbol{x}$ *and* $\boldsymbol{y}$*, and a set of values* Val*, ranged over by* v*, of which a subset* $\mathsf{Loc} \subseteq \mathsf{Val}$ *represents heap addresses, ranged over by* l*. The syntax of atomic heap commands,* Atom_H*, is defined by the grammar:*

$$a ::= \boldsymbol{x} := \boldsymbol{y} \ \big|\ [\boldsymbol{x}] := v \ \big|\ [\boldsymbol{x}] := \boldsymbol{y} \ \big|\ \boldsymbol{x} := [\boldsymbol{y}] \ \big|\ \boldsymbol{x} := \mathsf{ref}\ \boldsymbol{y}.$$

The first of these commands loads the contents of variable $\boldsymbol{y}$ into variable $\boldsymbol{x}$. The second and third load, respectively, a literal value v and the contents of $\boldsymbol{y}$ into the heap location whose address is in $\boldsymbol{x}$. The fourth loads the contents of the heap location whose address is in $\boldsymbol{y}$ into $\boldsymbol{x}$. The final command allocates a new heap location with initial contents from $\boldsymbol{y}$ and loads its address into $\boldsymbol{x}$.

2.5.1 Disjoint Concurrent Separation Logic

Judgements of disjoint concurrent separation logic are, as in the views framework, triples of the form $\{p\}\ C\ \{q\}$. Abstractly, the state is treated as a resource, which is divided up by individual variables and heap addresses. Thus, p and q describe resources, which hold information about part of the state. Formally, p and q are views from the separation view model induced by the separation algebra $(\mathcal{M}_\mathsf{DCSL}, \uplus, \{\emptyset\})$, where

$$\mathcal{M}_\mathsf{DCSL} \stackrel{\text{def}}{=} (\mathsf{Var} \uplus \mathsf{Loc}) \rightharpoonup_{\text{fin}} \mathsf{Val}.$$

That is, $\mathcal{M}_\mathsf{DCSL}$ is the set of finite partial functions from variables and heap addresses to values, with the partial monoid operation given by the union of partial functions with disjoint domains $\uplus$, and the unit consisting of only the partial function with the empty domain, $\emptyset$.

Elements of $\mathcal{M}_\mathsf{DCSL}$ declare ownership of the variables and heap addresses that belong to their domains, as well as defining their values. Significantly, they do not declare information about parts of the state which are not owned. Views $p, q \in \mathcal{P}(\mathcal{M}_\mathsf{DCSL})$ are sets of these abstract states.

The view $\boldsymbol{x} \Rrightarrow v$ denotes the singleton set of the partial function mapping variable $\boldsymbol{x}$ to value v, and $\boldsymbol{x} \Rrightarrow \_$ denotes the set of all partial functions that only map variable $\boldsymbol{x}$ to a value. Similarly, the views $l \mapsto v$ and $l \mapsto \_$ map heap address l to v or any value respectively. The view $\exists v.\, p(v)$ is the (infinite) join of $p(v)$ for all values of v.

Separation logic does not allow angelic update, and so the angelic preorder is simply $\vDash$ which is itself $\subseteq$.

The axiomatisation for separation logic is given by the schema:

$$\{\boldsymbol{x} \Rrightarrow \_ * \boldsymbol{y} \Rrightarrow v\}\ \boldsymbol{x} := \boldsymbol{y}\ \{\boldsymbol{x} \Rrightarrow v * \boldsymbol{y} \Rrightarrow v\}$$

$$\{\boldsymbol{x} \Rrightarrow l * l \mapsto \_\}\ [\boldsymbol{x}] := v\ \{\boldsymbol{x} \Rrightarrow l * l \mapsto v\}$$

$$\{\boldsymbol{x} \Rrightarrow l * l \mapsto \_ * \boldsymbol{y} \Rrightarrow v\}\ [\boldsymbol{x}] := \boldsymbol{y}\ \{\boldsymbol{x} \Rrightarrow l * l \mapsto v * \boldsymbol{y} \Rrightarrow v\}$$

$$\{\boldsymbol{x} \Rrightarrow l * l \mapsto v * \boldsymbol{y} \Rrightarrow \_\}\ \boldsymbol{y} := [\boldsymbol{x}]\ \{\boldsymbol{x} \Rrightarrow l * l \mapsto v * \boldsymbol{y} \Rrightarrow v\}$$

$$\{\boldsymbol{x} \Rrightarrow \_ * \boldsymbol{y} \Rrightarrow v\}\ \boldsymbol{x} := \mathsf{ref}\ \boldsymbol{y}\ \{\exists l.\, \boldsymbol{x} \Rrightarrow l * l \mapsto v * \boldsymbol{y} \Rrightarrow v\}$$

2.5.2 Weak-update Type System

Consider a simple type system for heap update, which types variables and heap cells with the set of types Type, ranged over by τ, and defined by:

$$\tau \ ::= \ \mathsf{val} \ \big| \ \mathsf{ref}\ \tau.$$

The type val indicates that a variable or heap cell contains some unspecified value, while the type $\mathsf{ref}\ \tau$ indicates that it contains the address of a heap cell whose contents is typed as τ. A typing context $\Gamma : \mathsf{Var} \rightharpoonup \mathsf{Type}$ is a partial function which assigns types to variables. In a weak update type system, the types of variables are fixed, and all assignments must preserve the typing. For the heap update language, we define such a type system by the typing rules:

$$\frac{}{\boldsymbol{x} : \tau, \boldsymbol{y} : \tau \vdash \boldsymbol{x} := \boldsymbol{y}} \qquad \frac{}{\boldsymbol{x} : \mathsf{ref\ val} \vdash [\boldsymbol{x}] := v}$$

$$\frac{}{\boldsymbol{x} : \mathsf{ref}\ \tau, \boldsymbol{y} : \tau \vdash [\boldsymbol{x}] := \boldsymbol{y}} \quad \frac{}{\boldsymbol{x} : \mathsf{ref}\ \tau, \boldsymbol{y} : \tau \vdash \boldsymbol{y} := [\boldsymbol{x}]}$$

$$\frac{}{\boldsymbol{x} : \mathsf{ref}\ \tau, \boldsymbol{y} : \tau \vdash \boldsymbol{x} := \mathsf{ref}\ \boldsymbol{y}} \qquad \frac{}{\Gamma \vdash \mathtt{skip}}$$

$$\frac{\Gamma \vdash C_1 \quad \Gamma \vdash C_2 \quad \mathsf{op} \in \{;, +, \|\}}{\Gamma \vdash C_1\ \mathsf{op}\ C_2} \quad \frac{\Gamma \vdash C}{\Gamma \vdash C^*} \qquad \frac{\Gamma \vdash C}{\Gamma, \Gamma' \vdash C}$$

The intended meaning of a typing judgement $\Gamma \vdash C$ is that, whenever the program C is executed from an initial state in which the variables can be typed according to Γ, the program does not fault and results in a state in which the variables can still be typed according to Γ.

We can form a separation algebra $(\mathcal{M}_{\mathsf{WTS}}, \cup, \{\emptyset\})$ of typing contexts, where $\mathcal{M}_{\mathsf{WTS}} \overset{\text{def}}{=} \mathsf{Var} \rightharpoonup \mathsf{Type}$, the monoid operator is union (of relations, restricted to when the result is a function) and the unit is the set of the partial function with the empty domain. In this separation algebra, two typing contexts can be combined when they agree on the types of all variables that are common to both: this is the implicit meaning of the "," operator in the typing rules above. This induces a view model in which the views are sets of typing contexts.

We fit the type system into the views framework by interpreting the judgement $\Gamma \vdash C$ as $\{\{\Gamma\}\}\ C\ \{\{\Gamma\}\}$. By taking the first five rules of the type system as axioms, and $\vDash$ as the angelic partial order, we obtain an instance of the views framework. The remaining rules of the type system are then easily justified by the proof rules of the views program logic. The most interesting of these is the last, the weakening rule, which is an instance of the frame rule, with the frame $\{\Gamma'\}$.

2.5.3 Strong-update Type System

In the previous example, every command preserves the types of variables: they are weak updates. We now consider a type system in which strong updates, which may change the type of variables, are permitted. In this system, each thread has its own local variables, which allows the types of the variables to be updated, since they are not shared with any other threads. The types of heap locations cannot be updated, however, since multiple threads may have aliases to the same location.

Typing judgements here use the same typing contexts as in the weak-update example. Since type-changing updates are permitted, judgements have input and output typing contexts. The type system is defined by the following typing rules:

$$\frac{}{x : \tau_0, y : \tau \vdash x := y \dashv x : \tau, y : \tau}$$

$$\frac{}{x : \mathsf{ref\ val} \vdash [x] := v \dashv x : \mathsf{ref\ val}}$$

$$\frac{}{x : \mathsf{ref}\ \tau, y : \tau \vdash [x] := y \dashv x : \mathsf{ref}\ \tau, y : \tau}$$

$$\frac{}{x : \mathsf{ref}\ \tau, y : \tau_0 \vdash y := [x] \dashv x : \mathsf{ref}\ \tau, y : \tau}$$

$$\frac{}{x : \tau_0, y : \tau \vdash x := \mathsf{ref}\ y \dashv x : \mathsf{ref}\ \tau, y : \tau}$$

$$\frac{}{\Gamma \vdash \mathtt{skip} \dashv \Gamma} \qquad \frac{\Gamma_1 \vdash C \dashv \Gamma_2}{\Gamma_1 \uplus \Gamma \vdash C \dashv \Gamma_2 \uplus \Gamma}$$

$$\frac{\Gamma_1 \vdash C_1 \dashv \Gamma_2 \quad \Gamma_2 \vdash C_2 \dashv \Gamma_3}{\Gamma_1 \vdash C_1; C_2 \dashv \Gamma_3} \qquad \frac{\Gamma \vdash C \dashv \Gamma}{\Gamma \vdash C^* \dashv \Gamma}$$

$$\frac{\Gamma \vdash C_1 \dashv \Gamma' \quad \Gamma \vdash C_2 \dashv \Gamma'}{\Gamma \vdash C_1 + C_2 \dashv \Gamma'} \qquad \frac{\Gamma_1 \vdash C_1 \dashv \Gamma_1' \quad \Gamma_2 \vdash C_2 \dashv \Gamma_2'}{\Gamma_1 \uplus \Gamma_2 \vdash C_1 \| C_2 \dashv \Gamma_1' \uplus \Gamma_2'}$$

The interpretation of a typing judgement $\Gamma \vdash C \dashv \Gamma'$ is that, when C is executed to termination from an initial state satisfying the typing context Γ, it will not fault and will result in a state satisfying the typing context Γ'.

We define the views model by considering an appropriate separation algebra. This time, the separation algebra is $(\mathcal{M}_{\mathsf{STS}}, \uplus, \{\emptyset\})$, where $\mathcal{M}_{\mathsf{STS}} \stackrel{\text{def}}{=} \mathsf{Var} \rightharpoonup \mathsf{Type}$. Rather than the union, as for weak update, we take the disjoint union of partial functions, as for separation logic. This enforces the intended discipline of ownership: threads may modify the types and values of variables that they own, but have no knowledge or rights to any other variables. Again, this separation algebra induces a model in which views are sets of typing contexts.

We interpret the type judgement $\Gamma \vdash C \dashv \Gamma'$ as $\{\{\Gamma\}\}\ C\ \{\{\Gamma'\}\}$ within the views framework. As before, we take the typing axioms as the axioms of our framework instance. By choosing the angelic preorder $\vDash$, the rules of this type system can be seen simply as instances of the rules from the views framework.

We can add the following subtyping rule to the system:

$$\frac{\Gamma, x : \mathsf{val} \vdash C \dashv \Gamma'}{\Gamma, x : \mathsf{ref}\ \tau \vdash C \dashv \Gamma'}$$

We can justify this rule by choosing an angelic preorder such that $\{\Gamma, x : \mathsf{ref}\ \tau\} \prec \{\Gamma, x : \mathsf{val}\}$. A suitable choice is

$$\begin{aligned} p \prec q \iff^{\text{def}} &\forall \Gamma \in p.\, \exists \Gamma' \in q.\, \mathrm{dom}\,(\Gamma) = \mathrm{dom}\,(\Gamma') \text{ and} \\ &\forall x \in \mathrm{dom}\,(\Gamma)\,.\, \Gamma'(x) \in \{\mathsf{val}, \Gamma(x)\} \end{aligned}$$

which satisfies the requirements of an angelic preorder.

2.5.4 Typed Separation Logic

Separation logic has been used to reason about programming languages with types, such as Java [24]. However, separation-logic reasoning typically ignores the types, although the work of Tan *et al.* [17] is a notable exception. Since type information is shared and global, and separation-logic reasoning is local, it previously seemed difficult to integrate the two systems. However, with our framework, it is easy. The system we present here only allows weak updates of the heap. In §5.3, we consider a more sophisticated combination which allows strong updates.

We can axiomatise the atomic commands by combining the axioms of the strong-update type system and separation logic:

$$\frac{\Gamma_1 \vdash a \dashv \Gamma_2 \quad \{p\}\ a\ \{q\}}{\Gamma_1 \vdash \{p\}\ a\ \{q\} \dashv \Gamma_2}$$

If the atomic command is allowed by the type system and the separation logic, then it is allowed in the combined system. We can also add axioms, such as the following, which derive from only one of the systems:

$$\frac{\Gamma_1 \vdash x := [y] \dashv \Gamma_2}{\Gamma_1 \vdash \{\boldsymbol{x} \Rrightarrow \_\}\ x := [y]\ \{\boldsymbol{x} \Rrightarrow \_\} \dashv \Gamma_2}$$

$$\frac{\{p\}\ x := [y]\ \{q\}}{x : \_ \vdash \{p\}\ x := [y]\ \{q\} \dashv x : \mathsf{val}}$$

Importantly, anything that is changed must be allowed by both systems, but the ability to access something only needs to be justified in one of the underlying systems.

To model this in the framework, we simply take the product of the two earlier models, $\mathsf{View}_{\mathsf{TSL}} \stackrel{\text{def}}{=} \mathsf{View}_{\mathsf{STS}} \times \mathsf{View}_{\mathsf{DCSL}}$, and lift the operators in the obvious way. We interpret the judgements $\Gamma_1 \vdash \{p\}\ C\ \{q\} \dashv \Gamma_2$ as $\{\{\Gamma_1\}, p\}\ C\ \{\{\Gamma_2\}, q\}$.

3 Operational Semantics and Soundness

3.1 Operational Semantics

The operational semantics of our language is parametrised by a model of machine states and an interpretation of the atomic commands as state transformers on this model.

Parameter 5 (Machine States)**.** *Assume a set of* machine states $\mathcal{S}$*, ranged over by* s*.*

Definition 8 (Machine Actions)**.** Machine Actions*, ranged over by* α*, are defined as a set of non-deterministic state transformers* $\mathsf{Action} \stackrel{\text{def}}{=} \mathcal{S} \to \mathcal{P}(\mathcal{S})$*. We use* id *to denote the identity action:* $\mathsf{id}(s) \stackrel{\text{def}}{=} \{s\}$*.*

Where necessary, we lift actions to sets of states: for $S \in \mathcal{P}(\mathcal{S})$, $\alpha(S) = \bigcup\{\alpha(s) \mid s \in S\}$. This lifting is monotone.

Parameter 6 (Interpretation of Atomic Commands)**.** *Assume a function* $[\![-]\!] : \mathsf{Atom} \to \mathsf{Action}$ *that associates each atomic command with a machine action.*

From machine state s, the set of states $[\![a]\!](s)$ is the set of possible outcomes of running the atomic command a. If the set is empty, then the command blocks. Here, we consider partial correctness, and so ignore blocking executions.

We define the operational semantics of the language using a labelled transition system. Transitions are between commands, and are labelled by the machine action that is executed in performing the transition. The action id plays a special role, since it represents the effect of computation steps in which the state is not changed.

The labelled transition system splits the control-flow aspect of execution, represented by the transitions between commands, and the state-transforming aspect of execution, represented by the labelling of the transitions. This makes it easy to reinterpret programs over a more abstract state-space, while preserving the control-flow structure, which simplifies the soundness proof of our logic.

Definition 9 (Labelled Transition System and Operational Semantics)**.** *The labelled transition relation* $- \xrightarrow{-} - : \mathsf{Comm} \times \mathsf{Action} \times \mathsf{Comm}$ *and the multi-step operational transition relation* $-, - \to^{*} -, - : (\mathsf{Comm} \times \mathcal{S}) \times (\mathsf{Comm} \times \mathcal{S})$ *are defined by the rules given in Fig. 2.*

$$\frac{C_1 \xrightarrow{\alpha} C_1'}{C_1; C_2 \xrightarrow{\alpha} C_1'; C_2} \qquad \frac{}{C_1 + C_2 \xrightarrow{\mathsf{id}} C_1} \qquad \frac{}{C_1 + C_2 \xrightarrow{\mathsf{id}} C_2}$$

$$\frac{C_1 \xrightarrow{\alpha} C_1'}{C_1 \parallel C_2 \xrightarrow{\alpha} C_1' \parallel C_2} \qquad \frac{C_2 \xrightarrow{\alpha} C_2'}{C_1 \parallel C_2 \xrightarrow{\alpha} C_1 \parallel C_2'} \qquad \frac{}{a \xrightarrow{[\![a]\!]} \mathtt{skip}}$$

$$\frac{}{\mathtt{skip}; C_2 \xrightarrow{\mathsf{id}} C_2} \qquad \frac{}{\mathtt{skip} \parallel C_2 \xrightarrow{\mathsf{id}} C_2} \qquad \frac{}{C_1 \parallel \mathtt{skip} \xrightarrow{\mathsf{id}} C_1}$$

$$\frac{}{C^* \xrightarrow{\mathsf{id}} C; C^*} \qquad \frac{}{C^* \xrightarrow{\mathsf{id}} \mathtt{skip}}$$

$$\frac{}{C, s \to^* C, s} \qquad \frac{C_1 \xrightarrow{\alpha} C_2 \quad s_2 \in \alpha(s_1) \quad C_2, s_2 \to^* C_3, s_3}{C_1, s_1 \to^* C_3, s_3}$$

Figure 2: The Operational semantics

3.2 Soundness

We prove that our program logic is sound with respect to the operational semantics. To do this, we must relate the views (partial, abstract states) with the machine states (concrete, complete states). This is achieved by an *erasure* function.

Parameter 7 (Erasure). *Assume an* erasure function

$$\lfloor - \rfloor : \mathsf{View} \to \mathcal{P}(\mathcal{S})$$

which maps views to sets of machine states, and is a join-semilattice homomorphism: $\lfloor \bigvee \{p_i\}_{i \in I} \rfloor = \bigcup \{\lfloor p_i \rfloor\}_{i \in I}$.

This erasure function must satisfy two properties: *monotonicity*, stating that the erasure of the angelic preorder is set inclusion, and *atomic soundness*, stating that the erasure of axioms corresponds to the interpretation of the atomic commands in the operational semantics.

Property 8 (Monotonicity of Erasure). *The erasure function is monotone from* $(\mathsf{View}, \prec)$ *to* $(\mathcal{P}(\mathcal{S}), \subseteq)$*:*

$$\forall p, q \in \mathsf{View}.\, p \prec q \implies \lfloor p \rfloor \subseteq \lfloor q \rfloor.$$

Remark. We have already noted that $\vDash$ is the minimal angelic preorder. For a given erasure function, there is also a well-defined notion of a maximal angelic preorder, which is the greatest angelic preorder for which monotonicity holds. This preorder, $\prec_{\max}$, may be defined as follows:

$$p \prec_{\max} q \overset{\text{def}}{\iff} \forall r.\, \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor.$$

It is easy to see that this satisfies the required properties, and that monotonicity of erasure will hold. It is also clear that if $p \prec q$ for some angelic preorder for which monotonicity holds, it will be the case that $p \prec_{\max} q$. This means that $\prec_{\max}$ really is maximal with respect to angelic preorders.

To define atomic soundness, we first define an action judgement, $\alpha \Vdash \{p\}\{q\}$, which interprets α as updating a thread's view from p to q provided it preserves any environment view.

Definition 10 (Action Judgement).

$$\alpha \Vdash \{p\}\{q\} \overset{\text{def}}{\iff} \forall r \in \mathsf{View}.\, \alpha\left(\lfloor p * r \rfloor\right) \subseteq \lfloor q * r \rfloor.$$

Lemma 2 (Basic Locality). *For all* $p, q, r \in \mathsf{View}$*,* $\alpha \in \mathsf{Action}$*, if* $\alpha \Vdash \{p\}\{q\}$ *then* $\alpha \Vdash \{p * r\}\{q * r\}$.

Lemma 3 (Basic $\prec$-closure). *For all* $p, p', q, q' \in \mathsf{View}$*,* $\alpha \in \mathsf{Action}$*, if* $p \prec p'$*,* $\alpha \Vdash \{p'\}\{q'\}$ *and* $q' \prec q$ *then* $\alpha \Vdash \{p\}\{q\}$.

Property 9 (Atomic Soundness). *For every* $(p, a, q) \in \mathsf{Axiom}$*, we have* $[\![a]\!] \Vdash \{p\}\{q\}$.

The following lemma gives an alternative characterisation of the action judgement for separation view models, which is typically simpler to check.

Lemma 4. *If* View *is a separation view model induced by* $(\mathcal{M}, \bullet, I)$*, then*

$$\alpha \Vdash \{p\}\{q\} \iff \forall m \in p, m' \in \mathcal{M}.\, \alpha(\lfloor \{m \bullet m'\} \rfloor) \subseteq \lfloor q * \{m'\} \rfloor.$$

Definition 11 (Semantic Judgement)**.** *For command* $C \in \mathsf{Comm}$ *and views* $p, q \in \mathsf{View}$*, the semantic judgement* $\Vdash \{p\}\ C\ \{q\}$ *is defined coinductively as follows.* $\Vdash \{p\}\ C\ \{q\}$ *holds if and only if the following two conditions are satisfied:*

1. *for all* $\alpha \in \mathsf{Action}$*,* $C' \in \mathsf{Comm}$*, if* $C \xrightarrow{\alpha} C'$*, then there exists* $p' \in \mathsf{View}$ *such that* $\alpha \Vdash \{p\}\{p'\}$ *and* $\Vdash \{p'\}\ C\ \{q\}$*;*
2. *if* $C \equiv \mathtt{skip}$ *then* $p \prec q$*.*

For each of the proof rules there is a corresponding lemma establishing that it holds for the semantic judgement. Lemma 2 is used in the proofs for the frame and concurrency rules, while Lemma 3 is used for the angelic consequence rule. Together, these lemmas establish the following result.

Lemma 5. *If* $\{p\}\ C\ \{q\}$ *is derivable in the program logic, then* $\Vdash \{p\}\ C\ \{q\}$*.*

To establish soundness, we link the semantic judgement to the multi-step operational semantics.

Lemma 6. *If* $\Vdash \{p\}\ C\ \{q\}$*, then for all* $s \in \lfloor p \rfloor$ *and* $s' \in \mathcal{S}$*, if* $(C, s) \rightarrow^* (\mathtt{skip}, s')$ *then* $s' \in \lfloor q \rfloor$*.*

This lemma follows from the definitions. The proof makes use of the fact that $*$ has a unit, and is the only place in the paper that we explicitly use this fact.

Theorem 1 (Soundness)**.** *Assume that* $\{p\}\ C\ \{q\}$ *is derivable in the program logic. Then, for all* $s \in \lfloor p \rfloor$ *and* $s' \in \mathcal{S}$*, if* $(C, s) \rightarrow^* (\mathtt{skip}, s')$ *then* $s' \in \lfloor q \rfloor$*.*

These results have been machine checked with Coq, and the proof scripts are available [25].

3.3 Examples

We return to the examples that we introduced in 2.5 to consider their soundness. First, we provide an operational interpretation for the heap-update commands.

Definition 12 (Heap States)**.** *Machine states for the heap-update commands are partial functions from variables and locations to values. There is also an exceptional faulting state, denoted* $\lightning$*, which represents the result of an invalid memory access. Formally,* $\mathcal{S}_{\mathsf{H}} \stackrel{\text{def}}{=} ((\mathsf{Var} \uplus \mathsf{Loc}) \rightharpoonup_{\text{fin}} \mathsf{Val}) \uplus \{\lightning\}$*.*

Definition 13 (Heap Command Semantics)**.** *The semantics of the atomic heap-update commands are given by:*

$$\llbracket \boldsymbol{x} := \boldsymbol{y} \rrbracket(s) \stackrel{\text{def}}{=} \begin{cases} \{s[\boldsymbol{x} \mapsto s(\boldsymbol{y})]\} & \textit{if } \boldsymbol{y} \in \mathrm{dom}(s) \\ \{\lightning\} & \textit{otherwise} \end{cases}$$

$$\llbracket [\boldsymbol{x}] := v \rrbracket(s) \stackrel{\text{def}}{=} \begin{cases} \{s[s(\boldsymbol{x}) \mapsto v]\} & \textit{if } \boldsymbol{x}, s(\boldsymbol{x}) \in \mathrm{dom}(s) \\ \{\lightning\} & \textit{otherwise} \end{cases}$$

$$\llbracket [\boldsymbol{x}] := \boldsymbol{y} \rrbracket(s) \stackrel{\text{def}}{=} \begin{cases} \{s[s(\boldsymbol{x}) \mapsto s(\boldsymbol{y})]\} & \textit{if } \boldsymbol{x}, \boldsymbol{y}, s(\boldsymbol{x}) \in \mathrm{dom}(s) \\ \{\lightning\} & \textit{otherwise} \end{cases}$$

$$\llbracket \boldsymbol{y} := [\boldsymbol{x}] \rrbracket(s) \stackrel{\text{def}}{=} \begin{cases} \{s[\boldsymbol{y} \mapsto s(s(\boldsymbol{x}))]\} & \textit{if } \boldsymbol{x}, s(\boldsymbol{x}) \in \mathrm{dom}(s) \\ \{\lightning\} & \textit{otherwise} \end{cases}$$

$$\llbracket \boldsymbol{x} := \mathsf{ref}\ \boldsymbol{y} \rrbracket(s) \stackrel{\text{def}}{=} \begin{cases} \{s[\boldsymbol{x} \mapsto l, l \mapsto s(\boldsymbol{y})] \mid l \in \mathsf{Loc} \setminus \mathrm{dom}(s)\} & \\ & \textit{if } \boldsymbol{y} \in \mathrm{dom}(s) \\ \{\lightning\} & \textit{otherwise.} \end{cases}$$

Here we extend dom *to* $\mathcal{S}_{\mathsf{H}}$ *by taking* $\mathrm{dom}(\lightning) = \emptyset$*.*

3.3.1 Disjoint Concurrent Separation Logic

Since separation-logic views are sets of partial functions from variables and locations to values, they can be seen as sets of heap states. Thus, we can define a simple notion of erasure.

Definition 14 (DCSL Erasure)**.** *The DCSL erasure function* $\lfloor - \rfloor_{\mathsf{DCSL}} : \mathsf{View}_{\mathcal{M}_{\mathsf{DCSL}}} \to \mathcal{P}(\mathcal{S})$ *is defined by* $\lfloor p \rfloor_{\mathsf{DCSL}} \stackrel{\text{def}}{=} p$.

It is trivial to see that $\lfloor - \rfloor_{\mathsf{DCSL}}$ is a join-semilattice homomorphism, and that it satisfies Property 8 (Monotonicity of Erasure). All that remains to be shown is Property 9 (Atomic Soundness). As a representative example, we consider the heap-lookup axiom:

$$\{\boldsymbol{x} \Rightarrow l * l \mapsto v * \boldsymbol{y} \Rightarrow \_\}\ \boldsymbol{y} := [\boldsymbol{x}]\ \{\boldsymbol{x} \Rightarrow l * l \mapsto v * \boldsymbol{y} \Rightarrow v\}.$$

By appeal to Lemma 4, it is sufficient to show that, for arbitrary $m_r \in \mathcal{M}_{\mathsf{DCSL}}$ and $w \in \mathsf{Val}$,

$$[\![\boldsymbol{y} := [\boldsymbol{x}]]\!](\lfloor \{[\boldsymbol{x} \mapsto l, l \mapsto v, \boldsymbol{y} \mapsto w] \bullet m_r\} \rfloor) \subseteq \lfloor \{[\boldsymbol{x} \mapsto l, l \mapsto v, \boldsymbol{y} \mapsto v] \bullet m_r\} \rfloor.$$

If $\boldsymbol{x}$, $\boldsymbol{y}$ or l is in $\mathrm{dom}(m_r)$ then the composition yields $\emptyset$ and so the inclusion holds trivially. Otherwise, by the semantics of the command,

$$\begin{aligned}
&[\![\boldsymbol{y} := [\boldsymbol{x}]]\!](\lfloor \{[\boldsymbol{x} \mapsto l, l \mapsto v, \boldsymbol{y} \mapsto w] \bullet m_r\} \rfloor) \\
&\qquad = \{m_r[\boldsymbol{x} \mapsto l, l \mapsto v, \boldsymbol{y} \mapsto w][\boldsymbol{y} \mapsto v]\} \\
&\qquad = \{m_r[\boldsymbol{x} \mapsto l, l \mapsto v, \boldsymbol{y} \mapsto v]\} \\
&\qquad = \lfloor \{[\boldsymbol{x} \mapsto l, l \mapsto v, \boldsymbol{y} \mapsto v] \bullet m_r\} \rfloor
\end{aligned}$$

as required.

Remark. Although mathematically they may be defined the same way, intuitively a separation-logic state and a heap state represent different things. In a separation-logic state, the partiality of the function means that the rest of the state is unknown and not accessible by the thread. In a heap state, the partiality of the function means that the undefined parts are unallocated.

Erasure can be seen as completing the state by treating the unknown regions as unallocated. We could define erasure differently, by completing the state in *every possible way*. For example $\lfloor \boldsymbol{x} \Rightarrow 5 \rfloor$ would be the set of all states in which $\boldsymbol{x}$ has value 5. By changing the notion of erasure like this, we obtain a slightly different notion of soundness.

Note that the erasure function does not cover the machine state space: in particular, there is no view p with $\lightning \in \lfloor p \rfloor$. This means that separation-logic triples do not permit memory faults to occur, which is part of the standard interpretation.

3.3.2 Weak-update Type System

We interpret typing contexts as the set of states which are well-typed with respect to the context. Consequently, in order to define erasure, we must define a notion of typing for states.

Definition 15 (State Typing)**.** *The state typing judgement* $\Gamma; \Theta \vdash s$*, where* $\Gamma : \mathsf{Var} \rightharpoonup \mathsf{Type}$*,* $s \in \mathcal{S}_{\mathsf{H}}$ *and* $\Theta : \mathsf{Loc} \rightharpoonup \mathsf{Type}$ *ranges over heap typing contexts, is defined as follows:*

$$\Gamma; \Theta \vdash s \stackrel{\text{def}}{\iff} \forall \boldsymbol{x} \in \mathrm{dom}(\Gamma) \,.\, \Theta \vdash s(\boldsymbol{x}) : \Gamma(\boldsymbol{x}) \ \wedge\ \forall l \in \mathrm{dom}(\Theta) \,.\, \Theta \vdash s(l) : \Theta(l)$$

where

$$\Theta \vdash v : \tau \stackrel{\text{def}}{\iff} \tau = \mathsf{val} \vee \tau = \mathsf{ref}\ (\Theta(v)).$$

The state typing essentially ensures that every typed variable and location has a value consistent with its type. Specifically, this means that references must refer to addresses that have the appropriate type. Note that it would not be possible to have $\boldsymbol{x}$ and $\boldsymbol{y}$ referencing the same location in the typing context $\boldsymbol{x} : \mathsf{ref\ val}, \boldsymbol{y} : \mathsf{ref\ ref\ val}$. This is necessary, since otherwise an update to the location via $\boldsymbol{x}$ could invalidate the type of $\boldsymbol{y}$.

Definition 16 (Weak Type Erasure)**.** *The WTS erasure function* $\lfloor - \rfloor_{\mathsf{WTS}} : \mathsf{View}_{\mathcal{M}_{\mathsf{WTS}}} \to \mathcal{P}(\mathcal{S}_{\mathsf{H}})$ *is defined by*

$$\lfloor \mathbf{\Gamma} \rfloor_{\mathsf{WTS}} \overset{\text{def}}{=} \{ s \in \mathcal{S}_{\mathsf{H}} \mid \exists \Theta. \exists \Gamma \in \mathbf{\Gamma}. \Gamma; \Theta \vdash s \}$$

It is again clear that this definition of erasure is a join-semilattice homomorphism, and that it satisfies Property 8 (Monotonicity of Erasure). It remains to show Property 9 (Atomic Soundness).

We consider the typing rule for updating a reference as an example. We need to show, for any τ and $\Gamma = (\boldsymbol{x} : \mathsf{ref}\ \tau, \boldsymbol{y} : \tau)$, that $[\![[\boldsymbol{x}] := \boldsymbol{y}]\!] \Vdash \{\{\Gamma\}\}\{\{\Gamma\}\}$. It is sufficient to show, for arbitrary Γ', that

$$[\![[\boldsymbol{x}] := \boldsymbol{y}]\!] \left(\lfloor \{\Gamma \cup \Gamma'\} \rfloor \right) \subseteq \lfloor \{\Gamma \cup \Gamma'\} \rfloor .$$

Suppose that $s \in \lfloor \{\Gamma \cup \Gamma'\} \rfloor$; it suffices to show that $s' = s[s(\boldsymbol{x}) \mapsto s(\boldsymbol{y})] \in \lfloor \{\Gamma \cup \Gamma'\} \rfloor$. There must be some Θ such that $\Gamma \cup \Gamma'; \Theta \vdash s$. We check that $\Gamma \cup \Gamma'; \Theta \vdash s'$. Since s' only differs from s in the value at $s(\boldsymbol{x})$, we only need to check that $\Theta \vdash s'(s(\boldsymbol{x})) : \Theta(s(\boldsymbol{x}))$. Now, for $\boldsymbol{x}$ to have type $\mathsf{ref}\ \tau$, it must be that $\Theta(s(\boldsymbol{x})) = \tau$. Furthermore, for $\boldsymbol{y}$ to have type τ, it must be that $\Theta \vdash s(\boldsymbol{y}) : \tau$. Hence, $\Gamma \cup \Gamma'; \Theta \vdash s'$, and so $s' \in \lfloor \{\Gamma \cup \Gamma'\} \rfloor$, as required.

3.3.3 Strong-update Type System

For the strong-update type system, we use an analogous erasure to that of Definition 16 for weak update. It is straightforward to prove monotonicity and atomic soundness. To demonstrate the proof of atomic soundness, we choose allocation as our representative example. We require to show, for arbitrary τ, τ_0, that

$$[\![\boldsymbol{x} := \mathsf{ref}\ \boldsymbol{y}]\!] \Vdash \{\{\boldsymbol{x} : \tau_0, \boldsymbol{y} : \tau\}\}\{\{\boldsymbol{x} : \mathsf{ref}\ \tau, \boldsymbol{y} : \tau\}\}.$$

It is sufficient to show, for arbitrary Γ', that

$$[\![\boldsymbol{x} := \mathsf{ref}\ \boldsymbol{y}]\!] \left(\lfloor \{(\boldsymbol{x} : \tau_0, \boldsymbol{y} : \tau) \uplus \Gamma'\} \rfloor \right) \subseteq \lfloor \{(\boldsymbol{x} : \mathsf{ref}\ \tau, \boldsymbol{y} : \tau) \uplus \Gamma'\} \rfloor .$$

Suppose that $s \in \lfloor \{(\boldsymbol{x} : \tau_0, \boldsymbol{y} : \tau) \uplus \Gamma'\} \rfloor$; it suffices to show that

$$s' = s[\boldsymbol{x} \mapsto l, l \mapsto s(y)] \in \lfloor \{(\boldsymbol{x} : \mathsf{ref}\ \tau, \boldsymbol{y} : \tau) \uplus \Gamma'\} \rfloor$$

for arbitrary $l \in \mathsf{Loc} \setminus \mathrm{dom}\,(s)$. There must be some Θ with $(\boldsymbol{x} : \tau_0, \boldsymbol{y} : \tau) \uplus \Gamma'; \Theta \vdash s$. We show that

$$(\boldsymbol{x} : \mathsf{ref}\ \tau, \boldsymbol{y} : \tau) \uplus \Gamma'; \Theta, l : \tau \vdash s'$$

Since s and s' only differ on the values of $\boldsymbol{x}$ and l, and the typing contexts also only differ on the types of these, we need only show $\Theta, l : \tau \vdash s'(\boldsymbol{x}) : \tau$ and $\Theta, l : \tau \vdash s'(l) : \tau$. The first of these holds since $s'(\boldsymbol{x}) = l$. The second holds since $\Theta \vdash s(\boldsymbol{y}) : \tau$ and $s'(l) = s(\boldsymbol{y})$. Hence $s' \in \lfloor \{(\boldsymbol{x} : \mathsf{ref}\ \tau, \boldsymbol{y} : \tau) \uplus \Gamma'\} \rfloor$, as required.

3.3.4 Typed separation logic

The soundness of this system follows primarily from the two systems on which it is built. We define the erasure as the intersection of the erasures of the underlying systems.

Definition 17 (TSL erasure)**.** *The TSL erasure function* $\lfloor - \rfloor_{\mathsf{TSL}} : \mathsf{View}_{\mathsf{TSL}} \to \mathcal{P}(\mathcal{S}_{\mathsf{H}})$ *is defined by*

$$\lfloor \mathbf{\Gamma}, p \rfloor_{\mathsf{TSL}} \overset{\text{def}}{=} \lfloor \mathbf{\Gamma} \rfloor_{\mathsf{STS}} \cap \lfloor p * \mathsf{true} \rfloor_{\mathsf{DCSL}} .$$

In the separation logic component, we allow the state to be larger than specified by the formula to account for any state that the type system might be additionally describing. The type system's erasure is already closed under larger states.

The soundness of the combined axiom follows directly from the soundness of the underlying two axioms, and the angelic preorder satisfies monotonicity as the two underlying orders satisfy monotonicity.

To show the soundness of the axiom

$$\frac{\Gamma_1 \vdash x := [y] \dashv \Gamma_2}{\Gamma_1 \vdash \{\boldsymbol{x} \Rrightarrow \_\}\ x := [y]\ \{\boldsymbol{x} \Rrightarrow \_\} \dashv \Gamma_2}$$

we assume, by the soundness of the strong-update type system, that

$$x := [y] \Vdash \{\{\Gamma_1\}\}\{\{\Gamma_2\}\}.$$

It is sufficient to show that, for arbitrary m_r and Γ',

$$[\![x := [y]]\!]\,(\lfloor \Gamma_1 \uplus \Gamma', \boldsymbol{x} \Rrightarrow \_ \uplus m_r \rfloor) \subseteq (\lfloor \Gamma_2 \uplus \Gamma', \boldsymbol{x} \Rrightarrow \_ \uplus m_r \rfloor)$$

Let $s \in \lfloor \Gamma_1 \uplus \Gamma', \boldsymbol{x} \Rrightarrow \_ \uplus m_r \rfloor$. By definition, $s \in \lfloor \Gamma_1 \uplus \Gamma' \rfloor_{\mathsf{STS}}$, and so, for all $s' \in [\![x := [y]]\!](s)$, $s' \in \lfloor \Gamma_2 \uplus \Gamma' \rfloor_{\mathsf{STS}}$ by the soundness of the strong-update type system. It remains to show that $s' \in \lfloor \boldsymbol{x} \Rrightarrow \_ \uplus m_r \rfloor$ also. We know that $s' \neq \lightning$ (since the strong-update type system prevents this) and we know that $s \in \lfloor \boldsymbol{x} \Rrightarrow \_ \uplus m_r \rfloor$. The semantics of the command ensures that s' only differs from s in the value of variable x, so we can conclude that $s' \in \lfloor \boldsymbol{x} \Rrightarrow \_ \uplus m_r \rfloor$, as required.

The soundness of the other axiom has a similar proof: since the separation logic triple guarantees against faulting and only x may be modified, any disjoint typing context is preserved by the command, while x may be typed as val.

4 Views and Interference

Views model partial, abstracted information about machine states that are immune to interference from other threads. However, typically we start with a view model that does not consider possible interference. We then account for interference by using an *interference operator* that captures how a given view may evolve under interference.

4.1 Interference Operators

Definition 18 (Interference Operator)**.** *Given view model* View*, an* interference operator $R : \mathsf{View} \to \mathsf{View}$ *is function on* View *that satisfies the following properties:*

$$R(p * q) \vDash R(p) * R(q) \tag{1}$$

$$R(u) = u \tag{2}$$

$$R\left(\bigvee \{q_i\}_{i \in I}\right) = \bigvee \{R(q_i)\}_{i \in I} \tag{3}$$

$$p \vDash R(p) \tag{4}$$

$$R(R(p)) \vDash R(p) \tag{5}$$

Property (3) *implies that R is monotone:*

$$p \vDash q \implies R(p) \vDash R(q) \tag{6}$$

Monotonicity (6)*, inflation* (4) *and idempotency* (5) *correspond to R being a closure operator.*

A view p is *stable* if $R(p) = p$. Given an unstable view p, $R(p)$ is the *stabilisation* of p. In fact, $R(p)$ is the least (with respect to $\vDash$) stable view that is greater than p. (Minimality can be seen as a consequence of properties (6) and (5).)

The separation property of interference operators (1) has the motto "more resource means less interference": the interference acting on the combined resource $p * q$ does no more than the interference acting separately on p and q. With (4) and (5), it implies that the composition of stable views is also stable:

$$R(p) * R(q) \vDash R(R(p) * R(q)) \vDash R(R(p)) * R(R(q)) \vDash R(p) * R(q)$$
$$\therefore \qquad R(p) * R(q) = R(R(p) * R(q))$$

Consequently, the subset of stable views themselves form a view model with the same composition, unit and join.

Lemma 7. *If R is an interference operator on* View *then* $R(\mathsf{View}) = \{R(p) \mid p \in \mathsf{View}\}$ *is a view model.*

We can also show that stable views are closed under meets:

$$\begin{aligned} \bigwedge \{R(p_i)\}_{i\in I} &\vDash R\left(\bigwedge \{R(p_i)\}_{i\in I}\right) \\ &= R\left(\bigvee \{q \mid \forall i.\, q \vDash R(p_i)\}\right) \\ &= \bigvee \{R(q) \mid \forall i.\, q \vDash R(p_i)\} \\ &= \bigvee \{R(q) \mid \forall i.\, R(q) \vDash R(p_i)\} \\ &\vDash \bigvee \{q \mid \forall i.\, q \vDash R(p_i)\} \\ &= \bigwedge \{R(p_i)\}_{i\in I} \\ \therefore \quad \bigwedge \{R(p_i)\}_{i\in I} &= R\left(\bigwedge \{R(p_i)\}_{i\in I}\right) \end{aligned}$$

Since meet is a greatest lower bound on View, it must also give a greatest lower bound on $R(\mathsf{View})$. As a corollary, if View is a Heyting algebra then $R(\mathsf{View})$ will also be a Heyting algebra.

Remark. It is possible to replace property (3) with (6) to obtain a weaker notion of interference operator. Again, $R(\mathsf{View})$ will be a view model, however, with a different join operator from that of View (in $R(\mathsf{View})$, the join of a family of elements is R applied to the join calculated in View).

Interference operators provide an approach to constructing view models, which may be used in our proof system. By restricting the proof system to the stable views, we alter the set of possible axioms: we forbid axioms with unstable pre- and postconditions; and we add new axioms that only preserve stable frames—such axioms could fail to preserve some unstable frame and be unsound in the original proof system.

4.2 Interference Relations

When a view model is generated from a separation algebra, we can characterise interference using interference relations.

Definition 19 (Interference Relation)**.** *An* interference relation $R \subseteq \mathcal{M} \times \mathcal{M}$ *is a preorder which satisfies the following decomposition property: for all $m_1, m_2, m, m' \in \mathcal{M}$ with $m = m_1 \bullet m_2$ and $m \mathrel{R} m'$, there exist $m_1', m_2' \in \mathcal{M}$ with $m_1 \mathrel{R} m_1'$, $m_2 \mathrel{R} m_2'$ and $m' = m_1' \bullet m_2'$.*

By lifting the interference relation to sets, we can obtain an interference operator. In fact, the following lemma establishes that there is a one-to-one correspondence between interference relations and interference operators.

Lemma 8. *Let* $\mathsf{View}_{\mathcal{M}}$ *be the view model induced by separation algebra $(\mathcal{M}, \bullet, I)$. Then there is a bijective correspondence, $\hat{-}$, between interference relations on $\mathcal{M}$ and interference operators on* $\mathsf{View}_{\mathcal{M}}$ *with*

$$m' \in \hat{R}(p) \iff \exists m \in p.\, m \mathrel{R} m' \ .$$

In this setting, the action judgement (Definition 10) has an equivalent formulation that is typically simpler to check.

Lemma 9.

$$\alpha \Vdash \{p\}\{q\} \iff \forall m \in \mathcal{M}.\, \alpha\left(\lfloor p * \{m\} \rfloor\right) \subseteq \left\lfloor q * \hat{R}(\{m\}) \right\rfloor .$$

4.3 Examples

We give examples of views with interference. We first explore the running examples of the paper. Although our example view models have been constructed without interference, it is instructive to consider how interference could be used in their definition. Interference allows us to separate instantaneous knowledge about the program state (which is subject to change as a result of the actions of other threads) and knowledge about how the state can evolve as a result of other threads' actions. The knowledge about the state can be seen as unstructured and global, whereas the instantaneous knowledge about the evolution of the state is a separable resource that is local to each thread. We also study the fine-grained reasoning of concurrent abstract predicates [15]. We would not know how to construct this example without using interference.

4.3.1 Separation Logic with Ownership

Rather than the view model introduced in 2.5, we construct a view model for disjoint concurrent separation logic by instrumenting machine states (excluding $\lightning$) with an ownership mask, which provides explicit permissions for stating which variables and addresses are "owned". Our set of instrumented states is:

$$\mathcal{M}_{\mathsf{MSL}} \overset{\text{def}}{=} (\mathsf{Var} \uplus \mathsf{Loc}) \rightharpoonup_{\text{fin}} (\mathsf{Val} \times \{0,1\}) \ .$$

Given instrumented state $m \in \mathcal{M}_{\mathsf{MSL}}$, for each variable $\boldsymbol{x}$ (or address l), the first component of $m(\boldsymbol{x})$ (or $m(l)$) is its actual value in the machine, while the second component indicates whether or not the variable (or location) is owned; if $m(\boldsymbol{x})$ is undefined, the variable is not in the state at all; if $m(l)$ is undefined then l is not allocated. Composition is defined by:

$$\begin{aligned} m_1 \bullet m_2 = m \overset{\text{def}}{\iff} & \operatorname{dom}(m_1) = \operatorname{dom}(m_2) = \operatorname{dom}(m) \\ & \wedge \forall k \in \operatorname{dom}(m) \,.\, m_1(k){\downarrow_1} = m_2(k){\downarrow_1} = m(k){\downarrow_1} \\ & \wedge m_1(k){\downarrow_2} + m_2(k){\downarrow_2} = m(k){\downarrow_2} \end{aligned}$$

Composition requires that the state components of the composed states are the same as that of the composite, and that their ownership masks sum to give the mask of the composite. This ensures that each variable and location is uniquely owned.

This composition is well-defined, associative and commutative. To complete the separation algebra $(\mathcal{M}_{\mathsf{MSL}}, \bullet, I)$, it remains to give the unit:

$$I = \{m \in \mathcal{M}_{\mathsf{MSL}} \mid \forall k \in \operatorname{dom}(m) \,.\, m(k){\downarrow_2} = 0\} \ .$$

It is also easy to see that this is indeed the unit.

If we were to construct a view model based on this separation algebra, the commands we could reason about would be very limited: they could not alter the (machine) state at all. To see this, suppose we are able to derive the triple $\{\{m\}\}\ C\ \{q\}$. Then, by the frame rule, we could also derive the triple $\{\{m \bullet m_i\}\}\ C\ \{q * \{m_i\}\}$, where $m_i \in I$ is the unit element for m. By the definition of composition, this is equivalent to

$$\Big\{\{m\}\Big\}\ C\ \left\{\left\{m' \in q \;\middle|\; \begin{array}{l} \operatorname{dom}(m) = \operatorname{dom}(m') \wedge \\ \forall k \in \operatorname{dom}(m) \,.\, m(k){\downarrow_1} = m'(k){\downarrow_1} \end{array}\right\}\right\}$$

Thus, C cannot change the values of any variables or locations.

The problem here is that programs are required to preserve all frames, and therefore all values. However, the intention behind ownership is that only variables and locations that are owned are guaranteed to be preserved by other threads. Thus, instead of preserving all frames, we wish only to preserve all *stable* frames for a suitable notion of stability. Such a notion can be obtained by defining an interference relation:

$$m \mathrel{R} m' \overset{\text{def}}{\iff} \forall k \in \operatorname{dom}(m) \,.\, m(k){\downarrow_1} > 0 \implies m'(k) = m(k)$$

Intuitively, this interference relation expresses that the environment can do anything that does not alter the variables and locations owned by the thread. It is not difficult to see that the interference relation satisfies the decomposition property: any change that does not alter the variables or locations owned by either thread does not alter the variables or locations owned by each thread individually.

If we now consider the view model induced by the separation algebra under this interference relation, $R(\mathsf{View}_{\mathsf{MSL}})$, we obtain a notion of view that is specific about variables and locations that are owned, but can say nothing at all about variables and locations that are not owned. Now, threads are at liberty to mutate variables and heap locations they own, and allocate locations that are not owned by other threads, since these operations preserve stable frames. (Note that composition plays an important role here, since it enforces that the environment cannot also own variables and locations that belong to the thread.)

We define an operation $\iota : R(\mathsf{View}_{\mathsf{MSL}}) \to \mathsf{View}_{\mathsf{DCSL}}$ by:

$$\iota(p) \overset{\text{def}}{=} \left\{m \in \mathcal{M}_{\mathsf{DCSL}} \;\middle|\; \begin{array}{l} \exists m' \in p. \forall k, v. \\ \quad m(k) = v \iff m'(k) = (v, 1) \end{array}\right\}$$

This operation in fact defines an isomorphism between the two view models, so we can really think of the separation logic model as being constructed in this way.

It is a small step to define a separation logic with fractional permissions [26] in a similar fashion. Instead of a bit mask from the set $\{0, 1\}$, variables and locations are associated with fractional permissions from the interval $[0, 1]$. Composition and interference can be defined in the same way as above. A thread may depend upon the environment preserving the value of anything for which it holds a non-zero permission. Conversely, in order to mutate a variable or location, a thread must ensure the environment cannot have a non-zero permission: the thread must have permission 1 itself.

4.3.2 Type Systems

We construct a model for the weak-update type system, in a similar fashion to above, by instrumenting machine states with type information. The set of instrumented states can be constructed by: $\mathcal{M}_{\mathsf{WUTS}} \overset{\text{def}}{=} \{(s, \Gamma, \Theta) \mid \Gamma; \Theta \vdash s\}$. We turn this into a separation algebra $(\mathcal{M}_{\mathsf{WUTS}}, \bullet, \mathcal{M}_{\mathsf{WUTS}})$ by defining

$$(s, \Gamma_1, \Theta_1) \bullet (s, \Gamma_2, \Theta_2) \overset{\text{def}}{=} (s, \Gamma_1 \cup \Gamma_2, \Theta_1 \cup \Theta_2)$$

with $\bullet$ undefined when the states are different or the result is not well-typed.

Again, if we construct a view model from this separation algebra, operations will not be allowed to mutate the state. However, we can define an interference relation, R, on $\mathcal{M}_{\mathsf{WUTS}}$ to be the least relation such that $(s, \Gamma, \Theta)\, R\, (s', \Gamma, \Theta')$ for all $s, s', \Gamma, \Theta, \Theta'$. This leads to a definition of views in which only the typing of variables is fixed. We interpret typing contexts as views in the following way: $[\![\Gamma]\!] \overset{\text{def}}{=} \{(s, \Gamma, \Theta) : (s, \Gamma, \Theta) \in \mathcal{M}_{\mathsf{WUTS}}\}$. This interpretation gives rise to an isomorphism between this view model and the one considered in 2.5.

The strong-update type system can be modelled by augmenting the instrumentation with an ownership mask on variables (though not the heap). Composition is defined to ensure disjoint ownership, as in the separation logic example. Interference then permits any changes that preserve the types of owned variables.

4.3.3 Concurrent Abstract Predicates

Concurrent Abstract Predicates (CAP) [15] extends separation logic with shared regions that can be mutated by multiple threads through atomic operations. Each shared region is associated with some interference environment which stipulates how it may be mutated, by defining a collection of actions. Threads can mutate the shared state only by performing actions for which they have a capability resource, which may be exclusive or non-exclusive. An example of a CAP assertion is the following:

$$\exists r.\, [\textsc{Lock}]^r_\pi * \boxed{(x \mapsto 0 * [\textsc{Unlock}]^r_1) \vee x \mapsto 1}^{\,r}_{I(r)} \tag{7}$$

This assertion represents a shared mutex in the heap at address x. The boxed part of the assertion describes the shared region: either the mutex is available (x has value 0) or it is locked (x has value 1). If it is available, the exclusive capability $[\textsc{Unlock}]^r_1$ belongs to the region. The thread itself has a non-exclusive capability $[\textsc{Lock}]^r_\pi$, which will permit it to lock the mutex. The actions corresponding to the capabilities are defined by the interference environment $I(r)$:

$$I(r) \overset{\text{def}}{=} \left(\begin{array}{ll} \textsc{Lock} & : x \mapsto 0 * [\textsc{Unlock}]^r_1 \rightsquigarrow x \mapsto 1, \\ \textsc{Unlock} : & x \mapsto 1 \rightsquigarrow x \mapsto 0 * [\textsc{Unlock}]^r_1 \end{array} \right)$$

The interference environment stipulates that a thread holding a Lock capability may (atomically) update x from 0 to 1 in the shared region, simultaneously removing the capability $[\textsc{Unlock}]^r_1$; a thread holding an Unlock capability may update x from 1 to 0 while returning $[\textsc{Unlock}]^r_1$ to the shared region. By locking the mutex, a thread acquires the exclusive capability to unlock it. This guarantees that only one thread can have locked the mutex at a time, and only that thread can unlock it. Having locked the mutex (through an atomic compare-and-swap operation), a thread's view may be described by the following assertion:

$$\exists r.\, [\textsc{Unlock}]^r_1 * [\textsc{Lock}]^r_\pi * \boxed{x \mapsto 1}^{\,r}_{I(r)} \tag{8}$$

Whereas in (7) the thread does not know whether the mutex is locked or not, in (8) it knows that it is locked. It can only make such an assertion because it holds the exclusive capability $[\textsc{Unlock}]^r_1$; without that, the assertion would not be stable. (In separation logic, capabilities are always directly associated with the information they pertain to, as in $l \mapsto 7$, whereas in CAP they can be separated.)

CAP assertions are modelled by sets of states instrumented with ownership, capabilities and region information. To use CAP, we must work with stable assertions — views — in order to account for possible interactions between threads. These stable assertions are defined by closure under an interference relation, which allows the environment to update regions in any way for which it can have a suitable capability. Appendix A details the construction of this model.

In our previous examples, interference was not essential. Since the interference relations were equivalences, views were sets of equivalence classes, and it was easy to define canonical representations of these classes. CAP, on the other hand, is difficult to model without considering interference explicitly. One reason for this might be that interference is directional, and not simply an equivalence. Consequently, expressing entailments as inclusions between closed sets is likely to be the simplest way of constructing a suitable view model. Another reason is that it is convenient to construct views from assertions that are not themselves stable, hence it is convenient to have a model in which we can represent both stable and unstable assertions.

5 Examples

We present a series of further instantiations of our framework, to illustrate how our framework can capture a variety of models studied in the literature.

5.1 Type system with recursive types

In this example, we extend the strong update type system to a more powerful set of types. In particular, we add recursive types and pairs so that we can represent interesting data structures like list.

The syntax of types is

$$\tau \ ::= \ \mathsf{val} \ \big| \ \mathsf{null} \ \big| \ \mathsf{ref}\ \tau\ \tau \ \big| \ \mathsf{nullable}\ \tau \ \big| \ \mu X.\,\tau \ \big| \ X$$

The type null represents the type inhabited just by the value 0. The reference type, ref τ_1 τ_2, is extended from earlier to point to a pair of locations with types τ_1 and τ_2 respectively. We add a type to allow a type to contain null, nullable τ. Note that ref cannot be a null pointer. Finally, we add recursive types and type variables, to allow the representation of recursive data types. We restrict consideration to *good* types, where for each recursive type its variable occurs only under a reference type.

As an example, we can define recursive data types like lists:

$$\mathsf{list}\tau \stackrel{\text{def}}{=} \mu X.\,\mathsf{nullable}\ (\mathsf{ref}\,\tau\,\mathsf{X})$$

Thus a list is possibly null. If it is not null, it is a reference to a pair of a τ and a listτ.

We can then give the rules for atomic commands as:

$$\frac{}{\boldsymbol{x} : \mathsf{nullable}\,\tau \vdash \mathsf{assumeNull}(\boldsymbol{x}) \dashv \boldsymbol{x} : \mathsf{null}}$$

$$\frac{}{\boldsymbol{x} : \mathsf{nullable}\,\tau \vdash \mathsf{assumeNotNull}(\boldsymbol{x}) \dashv \boldsymbol{x} : \tau}$$

$$\frac{}{\boldsymbol{x} : \_, \boldsymbol{y} : \tau \vdash \boldsymbol{x} := \boldsymbol{y} \dashv \boldsymbol{x} : \tau, \boldsymbol{y} : \tau}$$

$$\frac{}{\boldsymbol{x} : \mathsf{ref}\ \tau_1\,\tau_2, \boldsymbol{y} : \tau_1 \vdash \boldsymbol{x}.\mathsf{fst} := \boldsymbol{y} \dashv \boldsymbol{x} : \mathsf{ref}\ \tau_1\,\tau_2, \boldsymbol{y} : \tau_1}$$

$$\frac{}{\boldsymbol{x} : \mathsf{ref}\ \tau_1\,\tau_2, \boldsymbol{y} : \tau_2 \vdash \boldsymbol{x}.\mathsf{snd} := \boldsymbol{y} \dashv \boldsymbol{x} : \mathsf{ref}\ \tau_1\,\tau_2, \boldsymbol{y} : \tau_2}$$

$$\frac{}{\boldsymbol{x} : \mathsf{ref}\ \tau_1\,\tau_2, \boldsymbol{y} : \_ \vdash \boldsymbol{y} := \boldsymbol{x}.\mathsf{fst} \dashv \boldsymbol{x} : \mathsf{ref}\ \tau_1\,\tau_2, \boldsymbol{y} : \tau_1}$$

$$\frac{}{\boldsymbol{x} : \mathsf{ref}\ \tau_1\,\tau_2, \boldsymbol{y} : \_ \vdash \boldsymbol{y} := \boldsymbol{x}.\mathsf{snd} \dashv \boldsymbol{x} : \mathsf{ref}\ \tau_1\,\tau_2, \boldsymbol{y} : \tau_2}$$

$$\frac{}{\boldsymbol{x} : \_, \boldsymbol{y} : \tau_1, \boldsymbol{z} : \tau_2 \vdash \boldsymbol{x} := \mathsf{ref}\ \boldsymbol{y}\ \boldsymbol{z} \dashv \boldsymbol{x} : \mathsf{ref}\ \tau_1\,\tau_2, \boldsymbol{y} : \tau_1, \boldsymbol{z} : \tau_2}$$

The rules for non-atomic commands are the same as the earlier example.

To allow us to introduce nullable types and to fold and unfold recursive definitions, we add a subtyping relation, $\prec$, which is the least preorder satisfying:

$$\begin{gathered} \tau \prec \mathsf{nullable}\ \tau \\ \mathsf{null} \prec \mathsf{nullable}\ \tau \\ \mu X.\tau \prec \tau[\mu X.\tau / X] \\ \tau[\mu X.\tau / X] \prec \mu X.\tau \end{gathered}$$

This is lifted to typing contexts as

$$\Gamma_1 \prec \Gamma_2 \iff \forall x : \tau_2 \in \Gamma_2 . \exists x : \tau_1 \in \Gamma_1 . \tau_1 \prec \tau_2$$

and leads to the subtyping rule:

$$\frac{\Gamma_1 \prec \Gamma_1' \quad \Gamma_1' \vdash C_1 \dashv \Gamma_2' \quad \Gamma_2' \prec \Gamma_2}{\Gamma_1 \vdash C \dashv \Gamma_2}$$

We define a relation $\Theta \vdash v : \tau$ expressing when value v has type τ wrt. a heap typing Θ, mapping locations to types. Note that values range over integers.

$$\frac{}{\Theta \vdash v : \mathsf{val}} \qquad \frac{}{\Theta \vdash 0 : \mathsf{null}}$$

$$\frac{\Theta \vdash v : \mathsf{null}}{\Theta \vdash v : \mathsf{nullable}\ \tau} \qquad \frac{\Theta \vdash v : \tau}{\Theta \vdash v : \mathsf{nullable}\ \tau}$$

$$\frac{\Theta \vdash v : \tau[\mu X.\tau / X]}{\Theta \vdash v : \mu X.\tau} \qquad \frac{}{\Theta \vdash v : \mathsf{ref}\ (\Theta(v))\ (\Theta(v+1))}$$

We check a configuration is well-typed by checking each location and variable has the correct type with respect to the typing of the heap and stack.

$$\Gamma, \Theta \vdash_\mu s \overset{\text{def}}{\iff} \forall x \in \mathrm{dom}(\Gamma).\, \Theta \vdash s(x) : \Gamma(x) \ \wedge\ \forall l \in \mathrm{dom}(\Theta).\, \Theta \vdash s(l) : \Theta(l)$$

We use typing contexts, Γ, as instrumented states. We give erasure as any state that has a valid heap typing with respect to the type context of variables.

$$\lfloor \Gamma \rfloor_\mu \overset{\text{def}}{=} \{ s \mid \exists \Theta.\, \Gamma, \Theta \vdash_\mu s \}$$

We use the existential to represent that there is a way to type the heap consistently with all requirements the stack makes.

As before, we define the composition on typing contexts as disjoint function composition. To show the logic is sound we must show atomic soundness (Property 9), and we must show that subtyping is valid angelic preorder (Property 8).

5.2 Unique types

Previous example had an atomic allocate and instantiate instruction, $x := \mathsf{ref} y\, z$. But this is not very realistic as this would be done in many operations. In this example we extend the types from the previous section to allow us to separate allocation from instantiation.

We give a new high-level syntax to types, that allows us to represent a unique reference at the top-level:

$$t \ ::= \ \tau \mid \mathsf{uref}\ \tau\ \tau$$

We change Γ to map to these extended types.

We can then give rules for the atomic commands to allocate and update a unique reference

$$x : \_ \vdash x := \mathsf{ref} \dashv x : \mathsf{uref\,int\,int}$$
$$x : \mathsf{uref}\ \_\,\tau_2, y : \tau_1 \vdash x.\mathsf{fst} := y : \mathsf{uref} \dashv x : \mathsf{uref}\,\tau_1\,\tau_2, y : \tau_1$$
$$x : \mathsf{uref}\,\tau_1\,\_, y : \tau_2 \vdash x.\mathsf{snd} := y : \mathsf{uref} \dashv x : \mathsf{uref}\,\tau_1\,\tau_2, y : \tau_2$$

These rules allow us to alter the type of something in the heap. This is sound as we are currently the only thread to have access to the location.

We need to add to the subtyping relation

$$\mathsf{uref}\,\tau_1\,\tau_2 \prec \mathsf{ref}\,\tau_1\,\tau_2$$

This allows us to use the subtyping relation to forget the uniqueness as some point.

We need to adapt the notion of well typed to ensure that uref are the only reference to that location. We check a value has a uref type in the same way as a standard reference

$$\frac{}{\Theta \vdash v : \mathsf{uref}\,(\Theta(v))\,(\Theta(v+1))}$$

We check that each unique reference can only be viewed as a reference by a single stack variable:

$$\begin{array}{l}\Gamma, \Theta \vdash_{\mu u} s \overset{\text{def}}{\iff} \\ \quad \Gamma, \Theta \vdash_{\mu} s \wedge \\ \quad \forall x : \mathsf{uref}\,\_\,\_ \in \mathrm{dom}(\Gamma). \\ \qquad\qquad\qquad\qquad \forall y \in \mathsf{Vars}.\, x \neq y \wedge \mathsf{uref}\,\_\,\_ \prec \Gamma(y) \Rightarrow s(y) \neq s(x) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall l.\mathsf{uref}\,\_\,\_ \prec \Theta(l) \Rightarrow s(l) \neq s(x)\end{array}$$

Note this does not mean that the location of a unique reference cannot occur in another stack location, but it cannot occur as something potentially considered as a reference. Technically,

$$x : \mathsf{uref}\,\tau_1\,\tau_2, y : \_ \vdash y := x \dashv x : \mathsf{uref}\,\tau_1\,\tau_2, y : \mathsf{val}$$

is a sound rule in this model. We require this kind of property of our model as we only distinguish between integers and references at the type level. The concrete model does not separate them. Thus, it is possible for the allocator to use a location for which the integer is currently in use, but this doesn't mater. If it is being treated as a location, then it would be allocated and thus the allocator would not double allocate it.

We then define erasure as

$$\lfloor \Gamma \rfloor_{\mu u} \overset{\text{def}}{=} \{s \mid \exists \Theta.\, \Gamma, \Theta \vdash_{\mu u} s\}$$

and must show the axioms and pre-order have the required properties (Property 9 and Property 8).

5.3 Separation Logic combined with a type system

We can extend the uniqueness in the previous example by using separation logic to handle the locations we are dealing with uniquely. This example is inspired by Tan *et al.* [17].

We provide a separation logic like assertion language with the ability to describe a location as having a type:

$$P \ ::= \ \mathbf{x} \Rightarrow l \ \big|\ l : \tau \ \big|\ l \mapsto l, l \ \big|\ \ldots$$

We use a model of the heap that contains values for each location that is being handled by separation logic, and types for the weak update locations.

$$\mathbb{H} : \mathsf{Locs} \to (\mathsf{Val} + \mathsf{Types})_\bot$$

and the stack is just a partial function:

$$S : \mathsf{Vars} \to \mathsf{Val}_\bot$$

The composition requires that if both sides are non-bottom, then they must agree on the type information, otherwise we behave just like separation logic.

$$\mathbb{H} \in \mathbb{H}_1 \bullet \mathbb{H}_2 \iff \forall l.\, \{\mathbb{H}(l), \bot\} = \{\mathbb{H}_1(l), \mathbb{H}_2(l)\} \\ \vee\, \exists \tau.\, \tau = \mathbb{H}(l) = \mathbb{H}_1(l) = \mathbb{H}_2(l)$$

We can coerce a separation logic part of memory to typed memory by

$$\frac{\tau \prec \mathsf{uref}\, \tau_1\, \tau_2}{(\exists l_1, l_2.\, l \mapsto l_1, l_2 \;*\; l_1 : \tau_1 \;*\; l_2 : \tau_2) \;\Rightarrow\; l : \tau}$$

Note we are also allowed to freely duplicate type information

$$l : \tau \;\Rightarrow\; l : \tau * l : \tau$$

Now we can define the erasure of these states. First, we define when a standard heap is compatible with a typed heap:

$$\mathsf{states}(S, \mathbb{H}, s) \stackrel{\text{def}}{=} \forall l \in \mathsf{Locs}, v.\mathbb{H}(l) = v \Rightarrow s(l) = v \\ \wedge\, \forall x \in \mathsf{Vars}.S(x) = s(x)$$

This requires that the standard heap contain at least all the values of the typed heap.

We define a heap typing as compatible with a typed heap, if it contains all the typings of the typed heap:

$$\mathsf{comptypes}(\mathbb{H}, \Theta) \stackrel{\text{def}}{=} \forall l.\mathbb{H}(l) = \tau \Rightarrow \Theta(l) = \tau \\ \wedge\, \forall l, v.\mathbb{H}(l) = v \Rightarrow l \notin dom(\Theta)$$

It is important that the heap typing does not contain entries for any part of the heap that is treated using separation logic. This enforces that typed memory can only reference typed memory.

We can simplify our notion of well-typing to just talk about the heap, as we are treating the stack using separation logic

$$S, \mathbb{H} \vdash_{TSL} s \stackrel{\text{def}}{=} \\ \exists \Theta.\ \mathsf{comptypes}(\mathbb{H}, \Theta) \wedge \mathsf{states}(S, \mathbb{H}, s) \wedge \\ \forall l \in dom(\Theta).\ \Theta \vdash H(l) : \Theta(l)$$

Thus erasure can be given as

$$\lfloor S, \mathbb{H} \rfloor_{TSL} = \{H, S \mid \mathsf{welltype}(H, \mathbb{H})\}$$

6 Related Work

Our composition operator provides a logical notion of separation, which, as we have demonstrated by examples, need not be realized by physical separation in the concrete machine. This idea of fictional separation has been used in recent work on separation logics for concurrent languages [15, 9, 10, 14]. Similar ideas are also useful in a purely sequential setting to enable modular reasoning about abstract data structures implemented using physical sharing, but for which a logical notion of separation can be defined [27, 28, 29].

The soundness of Pottier's capability system [30] is based on an axiom that is similar to our definition of interference relation, and the soundness proof of concurrent abstract predicates [15] also uses an equivalent lemma. Our framework does not have an explicit notion of guarantee, so many of the other properties required in both Pottier's work and concurrent abstract predicates are not required. Feng's LRG [31] also provides conditions such that the stable predicates can be composed. The condition requires fences to delimit the scope of interference, which we do not require. LRG is the only combination of rely-guarantee with separation logic we have not encoded into our framework, and remains future work.

7 Conclusions

We have introduced views as a general framework in which a wide variety of compositional reasoning approaches can be constructed, understood and proved sound. We find it surprising and revealing that diverse approaches such as separation logic and type theory can be understood in an elegant, unifying setting. We have shown a selection of program logics that can be seen as applications of our framework.

Ten years ago, Reynolds asked "whether the dividing line between types and assertions can be erased" [32]. We have shown there is no dividing line: they are just different kinds of *view*. Given the range and complexity of the reasoning systems captured by the views framework, the framework and its soundness proof are surprisingly elegant. Our framework provides a unifying insight into the principles underlying a range of compositional reasoning systems. Ultimately, views capture the fundamental interplay between locality and interference that is inherent in concurrency.

In the future, we intend to apply our framework to further examples. For instance, we plan to consider atomic commands for asynchronous message passing; our framework could conceivably encode some form of session types. We will also consider languages with richer features, such as first class functions and continuations.

References

[1] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," *TOPLAS*, vol. 21, no. 3, pp. 527–568, 1999.

[2] A. Ahmed, M. Fluet, and G. Morrisett, "L^3: A linear language with locations," *Fundam. Inform.*, vol. 77, no. 4, pp. 397–449, 2007.

[3] F. Smith, D. Walker, and J. G. Morrisett, "Alias types," in *ESOP*, 2000.

[4] A. Charguéraud and F. Pottier, "Functional translation of a calculus of capabilities," in *ICFP*, 2008, pp. 213–224.

[5] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *CSL*, 2001, pp. 1–19.

[6] J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, and B. Reus, "A semantic foundation for hidden state," in *FOSSACS*, 2010, pp. 2–16.

[7] J. Schwinghammer, L. Birkedal, and K. Støvring, "A step-indexed Kripke model of hidden state via recursive properties on recursively defined metric spaces," in *FOSSACS*, 2011.

[8] L. Birkedal, N. Torp-Smith, and H. Yang, "Semantics of separation-logic typing and higher-order frame rules for Algol-like languages," *Logical Methods in Computer Science*, vol. 2, no. 5:1, 2006.

[9] X. Feng, R. Ferreira, and Z. Shao, "On the relationship between concurrent separation logic and assume-guarantee reasoning," in *ESOP*, 2007, pp. 173–188.

[10] V. Vafeiadis and M. J. Parkinson, "A marriage of rely/guarantee and separation logic," in *CONCUR*, 2007, pp. 256–271.

[11] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv, "Local reasoning for storable locks and threads," in *APLAS*, 2007.

[12] A. Hobor, "Oracle semantics," Ph.D. dissertation, Princeton University, Department of Computer Science, Princeton, NJ, October 2008.

[13] A. Buisse, L. Birkedal, and K. Støvring, "A step-indexed Kripke model of separation logic for storable locks," in *MFPS*, 2011.

[14] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis, “Deny-guarantee reasoning,” in *ESOP*, 2009, pp. 363–377.

[15] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, “Concurrent abstract predicates,” in *ECOOP*, 2010.

[16] M. Dodds, S. Jagannathan, and M. J. Parkinson, “Modular reasoning for deterministic parallelism,” in *POPL*, 2011.

[17] G. Tan, Z. Shao, X. Feng, and H. Cai, “Weak updates and separation logic,” in *APLAS*. Springer-Verlag, 2009, pp. 178–193.

[18] P. W. O’Hearn and D. J. Pym, “The logic of bunched implications,” *Bulletin of Symbolic Logic*, vol. 5, no. 2, pp. 215–244, 1999.

[19] D. J. Pym, *The Semantics and Proof Theory of the Logic of Bunched Implications*, ser. Applied Logic Series. Springer, 2002, vol. 26.

[20] V. Vafeiadis, “Concurrent separation logic and operational semantics,” in *MFPS*, 2011.

[21] C. Calcagno, P. W. O’Hearn, and H. Yang, “Local action and abstract separation logic,” in *LICS*. IEEE Computer Society, 2007, pp. 366–378.

[22] C. Calcagno, P. Gardner, and U. Zarfaty, “Local reasoning about data update,” *ENTCS*, vol. 172, pp. 133–175, 2007.

[23] R. Dockins, A. Hobor, and A. W. Appel, “A fresh look at separation algebras and share accounting,” in *APLAS*, 2009.

[24] M. J. Parkinson, “Local reasoning for java,” Ph.D. dissertation, University of Cambridge, November 2005.

[25] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang, “Addditional material,” http://sites.google.com/site/viewsmodel/, 2012.

[26] R. Bornat, C. Calcagno, P. O’Hearn, and M. Parkinson, “Permission accounting in separation logic,” in *POPL’05*, 2005.

[27] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse, “Abstraction and refinement for local reasoning,” in *VSTTE*, 2010.

[28] N. Krishnaswami, L. Birkedal, and J. Aldrich, “Verifying event-driven programs using ramified frame properties,” in *TLDI*, 2010.

[29] J. B. Jensen and L. Birkedal, “Fictional separation logic,” in *Proceedings of ESOP*, 2012, to Appear.

[30] F. Pottier, “Syntactic soundness proof of a type-and-capability system with hidden state,” INRIA, Tech. Rep., 2011, (available from the author).

[31] X. Feng, “Local rely-guarantee reasoning,” in *POPL*, 2009.

[32] J. C. Reynolds, “Separation logic: A logic for shared mutable data structures,” in *LICS*, 2002.

$$\begin{array}{rlrl} t \in \mathsf{Token} & \overset{\text{def}}{=} \mathsf{RID} \times \mathsf{AID} & s \in \mathsf{SState} & \overset{\text{def}}{=} \mathsf{RID} \rightharpoonup_{\text{fin}} \mathsf{LState} \\ \phi \in \mathsf{Cap} & \overset{\text{def}}{=} \mathsf{Token} \to [0,1] & a \in \mathsf{Act} & \overset{\text{def}}{=} \mathcal{P}(\mathsf{SState} \times \mathsf{LState}) \\ l \in \mathsf{LState} & \overset{\text{def}}{=} \mathcal{H} \times \mathsf{Cap} & \varsigma \in \mathsf{AMod} & \overset{\text{def}}{=} \mathsf{Token} \rightharpoonup \mathsf{Act} \end{array}$$

Figure 3: CAP type definitions.

A Concurrent Abstract Predicates Model

The concurrent abstract predicates (CAP) methodology [15] was developed to allow compositional reasoning about concurrent objects that present an abstract view of disjointness, but may be implemented with complex patterns of sharing and interference. This reasoning is realised by modelling the state as a set of shared regions, each encapsulating some portion of the state, and equipped with a protocol for manipulating the region. The protocol for a shared region is specified by associating actions with tokens that identify them. A thread may perform an action associated with a token if it has a capability resource for that token.

We build a view model for CAP by constructing a separation algebra and interference relation. As a basis, we assume a separation algebra $(\mathcal{H}, \bullet_{\mathsf{SL}}, \mathsf{emp})$ (typically the heap separation algebra), with an erasure function to concrete states $\lfloor - \rfloor_{\mathsf{SL}}$. We proceed to extend this with the instrumentation.

We give the type definitions for CAP in Figure 3. We use a set of tokens, written Token, to identify actions, which consist of an identifier for the region to which the action pertains, and an identifier for the action within the context of the region. Capabilities, written Cap, are functions from Token to permission values. Capabilities are composed by the permission composition $\bullet_\pi$. Logical states, written LState, combine a heap component and a capability component. We denote the first projection of a logical state l by l_{H} and the second projection by l_{C}. Composition of logical states is defined component-wise, $\odot : \mathsf{LState} \times \mathsf{LState} \to \mathcal{P}(\mathsf{LState})$: $\odot \overset{\text{def}}{=} (\bullet_{\mathsf{SL}}, \bullet_\pi)$. Shared states, written SState, are (finite) sets of logical states, each of which is associated with a unique region identifier.

Actions define how shared states can be updated. An action updates a single shared region, but may be contingent on the state of other shared regions. An action is therefore expressed as a relation between the original shared state in its entirety, and the new logical state of the shared region over which it is interpreted.[1] The set of actions is written Act. Action models associate actions with tokens. The set of action models, AMod, is ranged over by ς.

An instrumented state combines a local state (from LState), a shared state (from SState) and an action model (from AMod). Instrumented states are subject to a well-formedness constraint. To formalise the well-formedness, we first define the operation $\mathsf{lcol} : \mathsf{LState} \times \mathsf{SState} \times \mathsf{AMod} \rightharpoonup \mathcal{P}(\mathsf{LState})$, which collapses instrumented states to logical states.

$$\mathsf{lcol}(l, s, \varsigma) \overset{\text{def}}{=} l \odot \prod_{r \in \mathrm{dom}(s)}^{\odot} s(r).$$

The well-formedness predicate wf checks that the collapsed state exists, and, moreover, that capabilities only refere to actions that are defined, and that actions are only defined for shared regions that exist.

$$\begin{aligned} \mathsf{wf}(l, s, \varsigma) \overset{\text{def}}{\iff} & \exists l' \in \mathsf{lcol}(l, s, \varsigma). \forall (r, \alpha) \in \mathsf{Token}. \\ & (l'_{\mathrm{C}}(r, \alpha) > 0 \implies (r, \alpha) \in \mathrm{dom}(\varsigma)) \wedge \\ & ((r, \alpha) \in \mathrm{dom}(\varsigma) \implies r \in \mathrm{dom}(s)). \end{aligned}$$

The set of instrumented states is the set of well-formed states.

$$\mathcal{M}_{\mathsf{CAP}} \overset{\text{def}}{=} \{(l, s, \varsigma) \in \mathsf{LState} \times \mathsf{SState} \times \mathsf{AMod} \mid \mathsf{wf}(l, s, \varsigma)\}.$$

We write m_{L}, m_{S} and m_{A} for the first, second and third components of instrumented state m, respectively. Figure 4 illustrates the structure of an instrumented state.

[1] In [15], actions were treated syntactically. The presentation here as relations simplifies the model.

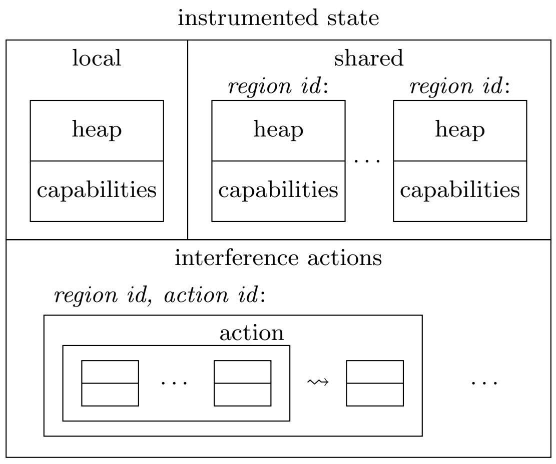

Figure 4: Representation of the structure of an instrumented state

Composition of instrumented states, $\bullet_{\mathsf{CAP}}$, combines the local states and requires that the other components are equal:

$$\bullet_{\mathsf{CAP}} \stackrel{\text{def}}{=} (\odot, \bullet_{=}, \bullet_{=}).$$

We next define the interference relation. Threads can perform updates to the shared state, provided that the update corresponds to some action for which the thread has a capability in its local state. From the perspective of interference, this means a thread should expect interference according to any action corresponding to a capability for which some other thread may have a non-zero capability. This is captured by the relation R_u:

$$\begin{aligned} m\ [R_u]\ m' \stackrel{\text{def}}{\iff}\ & m_{\mathrm{L}} = m'_{\mathrm{L}} \wedge m_{\mathrm{A}} = m'_{\mathrm{A}} \wedge \\ & \exists l \in \mathsf{lcol}(m).\, \exists (r, \alpha) \in \mathsf{Token}. \\ & \quad l_{\mathrm{C}}(r, \alpha) < 1 \wedge (m_{\mathrm{S}}, m'_{\mathrm{S}}(r)) \in m_{\mathrm{A}}(r, \alpha) \wedge \\ & \quad \forall r' \neq r.\, m_{\mathrm{S}}(r') = m'_{\mathrm{S}}(r'). \end{aligned}$$

It is also possible for threads to create new shared regions, in which case they obtain capability 1 for each action defined for the region. From the perspective of interference, a thread should expect a region to be created and actions for the new region to be added to the action model; no capabilities for the new region are initially available (since they are held by the environment thread that created the region). This is captured by the relation R_c:

$$\begin{aligned} m\ [R_c]\ m' \stackrel{\text{def}}{\iff}\ & m_{\mathrm{L}} = m'_{\mathrm{L}} \wedge \exists r \in \mathsf{RID}, \varsigma \in \mathsf{AMod}, l \in \mathsf{LState}. \\ & r \notin \mathrm{dom}(m_{\mathrm{S}}) \wedge m'_{\mathrm{S}} = m_{\mathrm{S}}[r \mapsto l] \wedge \\ & \exists l' \in \mathsf{lcol}(m').\, \forall \alpha \in \mathsf{AID}.\, l'_{\mathrm{C}}(r, \alpha) = 0 \wedge \\ & m'_{\mathrm{A}} = m_{\mathrm{A}} \uplus \varsigma \wedge \forall (r', \alpha') \in \mathrm{dom}(\varsigma).\, r' = r. \end{aligned}$$

Threads may also destroy regions if they have capability 1 for each action defined for the region. Consequently, from the perspective of interference, a thread should expect any region to be destroyed for which no capability can be accounted. This is captured by the relation R_c^{-1}, the inverse of R_c.

The interference relation R_{CAP} combines shared region update, creation and destruction and takes the reflexive-transitive closure:

$$R_{\mathsf{CAP}} \stackrel{\text{def}}{=} (R_u \cup R_c \cup R_c^{-1})^*.$$

We now have the components to construct our view model. To define the angelic preorder, we first define the erasure function, and then take $\prec_{\max}$, the maximal angelic preorder.

The erasure function, $\lfloor m \rfloor_{\mathsf{CAP}}$, is defined, by way of $\lfloor - \rfloor_{\mathsf{SL}}$, as collapsing all the regions and returning the associated heap:

$$\lfloor m \rfloor_{\mathsf{CAP}} \overset{\text{def}}{=} \{h \mid h \in \lfloor l_{\mathrm{H}} \rfloor_{\mathsf{SL}} \wedge l \in \mathsf{lcol}(m)\}.$$

Rather than giving an explicit axiomatisation, we will just consider here what kind of axioms are admissible.

Since local states are treated disjointly, a thread can freely mutate its local state, but has no access to the local state of any other thread. It can manipulate the shared state, but only in a way that some other thread is expecting. Since total capabilities cannot exceed 1, if a thread has, in its local state, a non-zero capability for some action, then there cannot be a thread that is able to see full capability for that action. Therefore, every thread in the environment must be expecting that action to occur.

Since other threads expect shared regions to be created, a thread is able to create a region, initially filling it with part of its local state, and obtaining all capabilities to that region. However, since other threads may also be creating regions, we cannot assert specifically what the region identifier should be, since any fixed choice may be taken by another thread. We assume an infinite population of region identifiers, so there are always free identifiers to choose from.

Since other threads expect regions to be destroyed when they cannot account for any capability to them, a thread is able to destroy a region to which it has all the capabilities. In that case, unless the resource contained in the region is concurrently being disposed, the region's contents revert to the thread's local state.

Aside from the abstraction rules, the proof rules for concurrent abstract predicates are simply the rules that our framework provides for this model.