Views: Compositional Reasoning for Concurrent Programs

Thomas Dinsdale-Young
Imperial College
td202@doc.ic.ac.uk

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

Philippa Gardner
Imperial College
pg@doc.ic.ac.uk

Matthew Parkinson
Microsoft Research
mattpark@microsoft.com

Hongseok Yang
University of Oxford
hongseok00@gmail.com

Abstract

Compositional abstractions underly many reasoning principles for concurrent programs: the concurrent environment is abstracted in order to reason about a thread in isolation; and these abstractions are composed to reason about a program consisting of many threads. For instance, separation logic uses formulae that describe part of the state, abstracting the rest; when two threads use disjoint state, their specifications can be composed with the separating conjunction. Type systems abstract the state to the types of variables; threads may be composed when they agree on the types of shared variables.

In this paper, we present the "Concurrent Views Framework", a metatheory of concurrent reasoning principles. The theory is parameterised by an abstraction of state with a notion of composition, which we call *views*. The metatheory is remarkably simple, but highly applicable: the rely-guarantee method, concurrent separation logic, concurrent abstract predicates, type systems for recursive references and for unique pointers, and even an adaptation of the Owicki-Gries method can all be seen as instances of the Concurrent Views Framework. Moreover, our metatheory proves each of these systems is sound without requiring induction on the operational semantics.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Logics of programs

General Terms Theory, Verification

Keywords concurrency; axiomatic semantics; compositional reasoning

1. Introduction

This paper aims to find the core principles underlying compositional reasoning systems for (first-order) concurrent programs. Compositional reasoning means that we consider each component in isolation without having to know the precise concurrent, or sequential, context in which it will be placed. This is essential for reasoning about incomplete code or libraries: the context is not known. In a concurrent setting, compositional reasoning allows a thread to be considered in isolation, rather than considering all possible interleavings of a program.

Type systems and program logics are two common forms of compositional reasoning. They strike a balance between invariant properties that must be preserved during the execution of a concurrent program, and operations that may be performed. Standard type systems and rely-guarantee methods [20] focus on preserving global properties: for example, the typing of the memory. Such approaches are good at handling sharing and interference, but work less well with stateful behaviour. Contrastingly, separation logic [18, 27] focuses on local portions of the state, which can be manipulated independently. This approach is good at handling stateful behaviour, but works less well with sharing and interference. There have been forays into the middle ground such as linear types [1, 23, 29] and related capability systems [7], which allow strong (type-changing) updates to memory, and SAGL [15], RGSep [31], LRG [14] and deny-guarantee [9, 13], which combine ideas from rely-guarantee and separation logic to enable reasoning about fine-grained concurrency in the heap. These developments have led to increasingly elaborate reasoning systems, each introducing new features to tackle specific applications of compositional reasoning and ad-hoc metatheory to justify these features.

Given the diversity of these approaches, it is not clear that they have significant common ground. Can the ad-hoc metatheory of each be generalised to give a paradigm in which all can be understood?

We argue that in fact the above reasoning systems employ a common approach to compositionality. They provide thread-specific abstractions of the state, which embody enough information to prove properties about the behaviour of a thread whilst allowing for the possible behaviours of other threads.

We introduce the "Concurrent Views Framework" ("Views Framework" or "Views" for short), which captures this compositional reasoning by introducing the notion of *views*. Intuitively, a thread's view consists of abstract *knowledge* about the current state of the machine and the thread's *rights* to change the state of the machine. The knowledge of a thread must be *stable* under the operations of concurrent threads: no other thread may have rights to invalidate the thread's knowledge. Conversely, no thread can have knowledge that another thread has the right to invalidate. Views are *compositional*: knowledge and rights may be distributed between threads and recombined.

To illustrate the Views Framework, first consider type systems for an imperative concurrent language. In a simple type system, the types of variables are invariant. Each thread's view is provided by

POPL'13, January 23–25, 2013, Rome, Italy.

a typing context, which embodies the knowledge that the values of variables agree with their types, and the rights to change the state such that this typing is preserved. When views are composed, they must agree on the types of all variables they share. In a type system that permits strong (*i.e.* type-changing) updates, threads again have knowledge that variables agree with their types, but may make updates that change the types of variables. Threads' views may be consistently composed only if they describe disjoint sets of variables, which each thread can be seen to *own*. Note that, since heap locations may be aliased by multiple variables, it is not in general permissible to update their types. However, a type system may include unique reference types, that confer ownership of heap locations, and hence allow type-changing updates. In this paper, we show how a simple type system (§2.2), a type system with strong updates, recursive types and subtyping (§7.1), and a type system with unique references (§7.2) can all be formalised in Views.

Now consider program logics for the same concurrent language. With concurrent separation logic, the views are assertions that describe the part of the state *owned* by a thread. These views embody knowledge about the owned part of the state, and confer the exclusive right to modify it. Views are composed with the separating conjunction, which enforces disjoint ownership of the state, so that no thread may alter state owned by another thread. More elaborate logics, such as deny-guarantee [13] and concurrent abstract predicates (CAP) [9], allow assertions to describe shared state and ownership of specific capabilities to update the shared state. Here, the *stability* of assertions is important: an assertion about the shared state must be invariant under operations for which other threads may have capabilities. In this paper, we represent a wide range of example program logics in our Views Framework, including several separation logics (§§3.1, 3.2, 7.4), the Owicki-Gries [24] (§7.3) and rely-guarantee methods [20] (§4) and CAP (§7.5). A fuller description of all our examples can be found in the technical report [11], along with Coq formalisations of many of them.

The Views Framework embodies the essential ingredients for sound compositional reasoning. Given a set of views with a composition operator (denoted $*$), together with an axiomatisation for the atomic commands, Views provides a generic program logic. For soundness, we require a *reification function* from views to sets of machine states and an *axiom soundness* property. The reification function relates partial, abstract views to complete, concrete machine states. Axiom soundness ensures that the axioms for the atomic commands are sound with respect to the reification, in the context of an arbitrary environment view. This embeds compositionality in the *meaning* of 'program C updates the view from p to q': for all environment views r, it must update $p * r$ to $q * r$. This approach has previously been used for extending sequential separation logic for higher-order languages, where it is otherwise difficult to characterise locality properties of commands, which are sufficient for ensuring compositionality [2, 3]. We show that the interpretation also provides a simpler and more general metatheory for logics of *concurrent* programs, identifying simple properties needed to prove soundness in contrast with the complex soundness results in the literature.

The Views Framework provides a generalised frame rule. This rule is essential for directly encoding the weakening rule from rely-guarantee and rules for manipulating resource invariants in concurrent separation logic. The new rule allows a function to be applied uniformly to the pre- and postcondition of a command. Without this rule the encoding of concurrent separation logic, rely-guarantee and Owicki-Gries method would require a far more complex representation of the high-level reasoning with permissions, such as deny-guarantee.

Given the wide range of examples that are instances of our framework, we believe "Views" embodies the core principles underlying compositional reasoning about concurrent programs.

Overview. In §2 we present our core contribution — the Concurrent Views Framework — defining the parameters and properties necessary to instantiate our general program logic. As a pedagogical example, we show how a simple type system may be encoded in Views. In §3, we show how separation algebras and interference can be used to construct instances of the Views Framework, illustrating these constructions with separation logic. In §4, we present a novel generalisation of the frame rule for the Views Framework. This rule makes it possible to encode contextual rules, which we illustrate by the weakening rule of the rely-guarantee method. We present a general combination rule in §5, which generalises the rules of disjunction and conjunction. We give a sufficient condition for its soundness and consider the special cases of disjunction and conjunction. We outline the general soundness proof for the framework in §6. In §7, we show the versatility of Views through five further example instances: recursive types, unique types, the Owicki-Gries method, atomic concurrent separation logic, and concurrent abstract predicates. Finally, we discuss related work in §8 and conclude in §9.

We have formalised all of the metatheory of the paper, as well as many of the examples, in Coq (see [11]).

2. Concurrent Views Framework

In this section, we present the Concurrent Views Framework. In §2.1, we present the programming language and its operational semantics, which is parameterised by a notion of state and a set of primitive atomic commands and their semantics as state transformers. In §2.2, we provide a set of rules for compositional reasoning, which is parameterised by an abstraction of the state equipped with a composition operation — a *views semigroup* — and an axiomatisation of the atomic commands. In §2.3, we give a general soundness result for the framework with respect to the operational semantics, which is parameterised by a relationship between the abstract views and the concrete states, and requires each axiom to satisfy a soundness property with respect to the semantics of the corresponding atomic command. To illustrate the Views Framework, we interleave its definition with an example of how to instantiate it for a simple type system for a heap update language.

2.1 Programming Language and Operational Semantics

We define the Views Framework on a simple programming language that is built from standard composite commands, and parameterised by a set of atomic commands. This enables us to consider many examples without having to change the metatheory.

Parameter A (Atomic Commands)**.** *Assume a 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:*

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

The operational semantics is parameterised by a model of machine states and an interpretation of the atomic commands as state transformers.

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

Parameter C (Interpretation of Atomic Commands)**.** *Assume a function* $[\![-]\!] : \mathsf{Atom} \to \mathcal{S} \to \mathcal{P}(\mathcal{S})$ *that associates each atomic command with a non-deterministic state transformer. (Where necessary, we lift non-deterministic state transformers to sets of states: for* $S \in \mathcal{P}(\mathcal{S})$*,* $[\![a]\!](S) = \bigcup \{[\![a]\!](s) \mid s \in S\}$*.)*

For 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 executions that deadlock — that is, reach a state (other than `skip`) in which every thread is blocked. If we wish to guarantee against some exceptional behaviour, it should be modelled with an exception state, rather than by blocking.

We define the operational semantics of the language using a labelled transition system. Transitions are between commands, and are labelled by atomic commands or id. id labels computation steps in which the state is not changed.

Definition 2 (Transition Labels)**.** *The set of* transition labels*,* $\mathsf{Label} \stackrel{\text{def}}{=} \mathsf{Atom} \uplus \{\mathsf{id}\}$*, extends the set of atomic commands with a designated identity label,* id*. Labels are ranged over by* α*.* $[\![-]\!]$ *is extended to* Label *by defining* $[\![\mathsf{id}]\!] \stackrel{\text{def}}{=} \lambda s.\, \{s\}$*.*

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 3 (Labelled Transition System and Operational Semantics)**.** *The labelled transition relation* $- \xrightarrow{-} - : \mathsf{Comm} \times \mathsf{Label} \times \mathsf{Comm}$ *is defined by the following rules:*

$$\frac{C_1 \xrightarrow{\alpha} C_1'}{C_1; C_2 \xrightarrow{\alpha} C_1'; C_2} \qquad \frac{}{\mathtt{skip}; C_2 \xrightarrow{\mathsf{id}} C_2} \qquad \frac{}{C_1 + C_2 \xrightarrow{\mathsf{id}} C_i}\, i \in \{1,2\}$$

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

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

The multi-step operational transition relation $-, - \rightarrow^* -, - : (\mathsf{Comm} \times \mathcal{S}) \times (\mathsf{Comm} \times \mathcal{S})$ *is defined by the following rules:*

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

Heap Update Language

For our examples, we use a language with simple atomic primitives for manipulating a heap. We therefore define instances of Parameters A, B and C accordingly.

Definition 4 (Atomic Heap Commands) **: Parameter A.** *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,* $\mathsf{Atom}_{\mathsf{H}}$*, is defined by the grammar:*

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

Definition 5 (Heap States) **: Parameter B.** *Machine states 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 6 (Heap Command Interpretation) **: Parameter C.** *The interpretation of the atomic heap-update commands is given by:*

$$\begin{aligned}
[\![\boldsymbol{x} := \boldsymbol{y}]\!](s) &\stackrel{\text{def}}{=} \boldsymbol{y} \in \mathrm{dom}(s) \triangleright \{s[\boldsymbol{x} \mapsto s(\boldsymbol{y})]\} \\
[\![[\boldsymbol{x}] := v]\!](s) &\stackrel{\text{def}}{=} \boldsymbol{x}, s(\boldsymbol{x}) \in \mathrm{dom}(s) \triangleright \{s[s(\boldsymbol{x}) \mapsto v]\} \\
[\![[\boldsymbol{x}] := \boldsymbol{y}]\!](s) &\stackrel{\text{def}}{=} \boldsymbol{x}, \boldsymbol{y}, s(\boldsymbol{x}) \in \mathrm{dom}(s) \triangleright \{s[s(\boldsymbol{x}) \mapsto s(\boldsymbol{y})]\} \\
[\![\boldsymbol{x} := [\boldsymbol{y}]]\!](s) &\stackrel{\text{def}}{=} \boldsymbol{y}, s(\boldsymbol{y}) \in \mathrm{dom}(s) \triangleright \{s[\boldsymbol{x} \mapsto s(s(\boldsymbol{y}))]\} \\
[\![\boldsymbol{x} := \mathsf{ref}\ \boldsymbol{y}]\!](s) &\stackrel{\text{def}}{=} \boldsymbol{y} \in \mathrm{dom}(s) \triangleright \\
&\qquad \{s[\boldsymbol{x} \mapsto l, l \mapsto s(\boldsymbol{y})] \mid l \in \mathsf{Loc} \setminus \mathrm{dom}(s)\}
\end{aligned}$$

where $b \triangleright s =$ *if* b *then* s *else* $\lightning$*. Here we extend* dom *to* $\mathcal{S}_{\mathsf{H}}$ *by taking* $\mathrm{dom}(\lightning) = \emptyset$*.*

2.2 Views and Program Logic

Reasoning systems typically do not require the user to reason directly about the state; they provide an abstract representation of the state that supports a particular form of reasoning. For example, in a simple type system the state is abstracted by a type context, Γ, which maps variables, $\boldsymbol{x}$, to types, τ. A type represents a property about the variable, *e.g.* that it is a pointer to a cell in the heap. In a program logic like separation logic, the state is abstracted by assertions, P, Q, that describe parts of the memory. Complex assertions may be used to represent interesting structures, such as a lock x with an invariant P: $\mathsf{Lock}(x, P)$. Such assertions may not merely describe the state, but also enforce a protocol that threads must obey — for instance, only releasing a lock after establishing its invariant. These abstract representations do not precisely describe the memory, just certain properties it satisfies. They also can contain additional information — about typing, ownership and protocols, for instance — that is not present in the underlying state.

These abstractions form the key parameter of the framework: the *views*. We have one basic requirement for views, which is that they form a commutative semigroup.

Parameter D (View Commutative Semigroup)**.** *Assume a commutative semigroup* $(\mathsf{View}, *)$*. The variables* p, q, r *are used to denote elements of* View*.*

If the semigroup has a unit, u, then we call it a *view monoid*, written $(\mathsf{View}, *, u)$. The unit is not required for the theory, but simplifies its use.

Intuitively, views are resources that embody *knowledge* about the state and the protocol threads must obey, and *rights* to modify the state in accordance with the implied protocol. The semigroup operation $*$ (*view composition*) combines the knowledge and rights of two views.

Since composition is used to combine the views of different threads, it must ensure consistency between these views. For example, to combine two typing contexts, they must agree on the type of any variables they have in common. Since threads only maintain the types in their view, if agreement was not enforced then one thread might violate another's expectations.

In separation logic, composition (separating conjunction) enforces that ownership is exclusive: no two views can simultaneously have ownership of a heap cell. In variants that allow assertions about shared state, such as deny-guarantee, composition also forces consistency between such assertions.

This consistency is typically implemented by a special inconsistent view that is the result of composing two views that are not consistent. The inconsistent view is usually a (unique) zero element of the semigroup: when composed with any other view, the result is the inconsistent view.

We define a *program logic* for our programming language, in which views provide the pre- and postconditions of the commands.

The program logic is parameterised by the set of axioms for atomic commands.

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

Definition 7 (Entailment)**.** *The* entailment relation $\models \subseteq \mathsf{View} \times \mathsf{View}$ *is defined by:*

$$p \models q \stackrel{\text{def}}{\iff} (p, \mathsf{id}, q) \in \mathsf{Axiom}.$$

Definition 8 (Program Logic)**.** *The program logic's judgements are of the form* $\vdash \{p\}\, C\, \{q\}$*, where* $p, q \in \mathsf{View}$ *provide the precondition and postcondition of command* $C \in \mathsf{Comm}$*. The proof rules for these judgements are as follows:*

$$\frac{(p, a, q) \in \mathsf{Axiom}}{\vdash \{p\}\, a\, \{q\}} \qquad \frac{\vdash \{p\}\, C\, \{q\}}{\vdash \{p * r\}\, C\, \{q * r\}} \qquad \frac{}{\vdash \{p\}\, \mathtt{skip}\, \{p\}}$$

$$\frac{\vdash \{p\}\, C_1\, \{q\} \quad \vdash \{p\}\, C_2\, \{q\}}{\vdash \{p\}\, C_1 + C_2\, \{q\}} \qquad \frac{\vdash \{p\}\, C\, \{p\}}{\vdash \{p\}\, C^*\, \{p\}}$$

$$\frac{\vdash \{p\}\, C_1\, \{r\} \quad \vdash \{r\}\, C_2\, \{q\}}{\vdash \{p\}\, C_1; C_2\, \{q\}} \qquad \frac{\vdash \{p_1\}\, C_1\, \{q_1\} \quad \vdash \{p_2\}\, C_2\, \{q_2\}}{\vdash \{p_1 * p_2\}\, C_1 \parallel C_2\, \{q_1 * q_2\}}$$

$$\frac{p \models p' \quad \vdash \{p'\}\, C\, \{q\}}{\vdash \{p\}\, C\, \{q\}} \qquad \frac{\vdash \{p\}\, C\, \{q'\} \quad q' \models q}{\vdash \{p\}\, C\, \{q\}}$$

The intended semantics of $\vdash \{p\}\, C\, \{q\}$ 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.

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. Note that, although in separation logic the concurrency rule imposes disjointness, in Views it does not; it just imposes composability. The last two rules are rules of consequence: they allow the precondition to be strengthened or the postcondition weakened with respect to the axiomatically-specified relation $\models$. If $\models$ is reflexive, these rules may be combined into a single rule.

Simple Type System

Consider a simple type system for our heap update language, where variables and heap cells are typed from the set Type, ranged over by τ, and defined by:

$$\tau \;::=\; \mathsf{val} \;\mid\; \mathsf{ref}\ \tau.$$

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.

Within the Views Framework, typing contexts are views. Composition, $\cup_\perp$, is the union of contexts (as relations), with a special *inconsistent typing context*, $\perp$, used to denote the composition of typing contexts that disagree on the type of some variable.

Definition 9 (Simple Type Views) **: Parameter D.** *The view monoid for the simple type system is defined as:*

$$\left((\mathsf{Var} \rightharpoonup \mathsf{Type}) \uplus \{\perp\}, \cup_\perp, \emptyset\right)$$

Judgements of the type system have the form $\Gamma \vdash C$. We treat this as syntax for $\vdash \{\Gamma\}\, C\, \{\Gamma\}$ in the Views Framework. It remains to give axioms for the type system.

Definition 10 (Simple Type Axioms) **: Parameter E.** *The axioms for the simple type system are defined schematically as:*

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

The inference rules of the type system are as follows:

$$\frac{}{\Gamma \vdash \mathtt{skip}} \qquad \frac{\Gamma \vdash C_1 \quad \Gamma \vdash C_2 \quad \mathsf{op} \in \{;, +\}}{\Gamma \vdash C_1\ \mathsf{op}\ C_2}$$

$$\frac{\Gamma_1 \vdash C_1 \quad \Gamma_2 \vdash C_2}{\Gamma_1, \Gamma_2 \vdash C_1 \| C_2} \qquad \frac{\Gamma \vdash C}{\Gamma \vdash C^*} \qquad \frac{\Gamma \vdash C}{\Gamma, \Gamma' \vdash C}$$

Each of these rules is justified by the rules of the Views program logic, where we interpret Γ_1, Γ_2 as $\Gamma_1 \cup_\perp \Gamma_2$. The most interesting of these is the last, the weakening rule, which is an instance of the frame rule, with the frame Γ'.

2.3 Reification and Soundness

In order to relate the program logic to the operational semantics, it is necessary to define a relationship between the abstract views and the concrete machine states. In a type system, this would be given by a judgement $\Gamma \vdash s$ asserting that state s is well-typed with respect to the context Γ. In separation logic, it would be given by a Kripke model, where the judgement $s \models P$ asserts that formula P holds in state s. In the Views Framework, we model this relationship with a *reification function*.

Parameter F (Reification)**.** *Assume a* reification function $\lfloor - \rfloor : \mathsf{View} \to \mathcal{P}(\mathcal{S})$ *which maps views to sets of machine states.*

Note that the reification function has very few restrictions. The semigroup structure of views is not reflected at the level of machine states: the notion of composition is purely an abstraction. The space of machine states need not be covered completely by the reification; this is useful for giving fault-avoiding interpretations to our proof judgements. Furthermore, inconsistent views may be mapped to the empty set of machine states.

Choosing an appropriate reification function is important. We could, for instance, map all views to the empty set and claim soundness of an arbitrary logic. However, judgements in the logic would convey no information about the operational semantics, as they would not describe the execution of a program from any initial state at all. Consequently, soundness is always with respect to the choice of reification function.

Soundness requires that the axioms concerning atomic commands are satisfied by the operational interpretation of the commands. We define an abstract notion of executing a small step of the program. To be able to make the step from p to q with the action α, we must ensure that the operational interpretation of the action satisfies the specification, and moreover it also preserves any possible environment view.

Definition 11 (Action Judgement)**.** *The* action judgement *is defined as:*

$$\alpha \Vdash \{p\}\{q\} \stackrel{\text{def}}{\iff}$$
$$[\![\alpha]\!](\lfloor p \rfloor) \subseteq \lfloor q \rfloor \;\wedge\; \forall r \in \mathsf{View}.\, [\![\alpha]\!](\lfloor p * r \rfloor) \subseteq \lfloor q * r \rfloor.$$

For a view monoid, this definition simplifies to:

$$\alpha \Vdash \{p\}\{q\} \iff \forall r \in \mathsf{View}.\, [\![\alpha]\!](\lfloor p * r \rfloor) \subseteq \lfloor q * r \rfloor. \quad (1)$$

Semantic entailment is a special case of the action judgement, for the label id.

Definition 12 (Semantic Entailment)**.** $p \preceq q \stackrel{\text{def}}{\iff} \mathsf{id} \Vdash \{p\}\{q\}$.

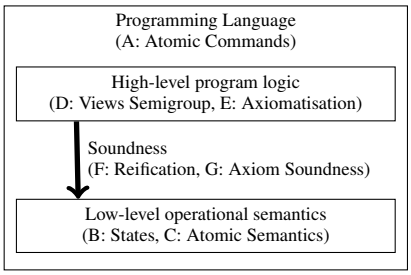

Figure 1. Overview of the Views Framework

For each axiom (p, α, q), the interpretation of α must update view p to q while preserving any environment view. This is captured by the following property:

Property G (Axiom Soundness)**.** *For every* $(p, \alpha, q) \in \mathsf{Axiom}$*,*

$$\alpha \Vdash \{p\}\{q\}\,.$$

This property is both necessary and sufficient for the soundness of the program logic. We state the soundness result here, and provide more details in §6.

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

Simple Type System: Soundness

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

Definition 13 (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:*

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

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

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 14 (Simple Type Reification) **: Parameter F.** *The simple type reification,* $\lfloor - \rfloor_{\mathsf{TS}} : \mathsf{View}_{\mathcal{M}_{\mathsf{TS}}} \to \mathcal{P}(\mathcal{S}_\mathsf{H})$*, is defined as follows:*

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

To establish soundness, we need only show Property G (Axiom Soundness). This is straightforward; for further details, consult [11]. Importantly, this works because we do not require locality at the low level of the semantics, only at the high level. Thus, standard separation algebra approaches [6, 12] would not work for this example.

2.4 Summary

The parameters of the Views Framework are summarised in Figure 1. The language (§2.1) is parameterised by a set of atomic commands, and its semantics is determined by a notion of state and the semantics of the atomic commands as state transformers. The Views program logic (§2.2) is parameterised by a semigroup of views and an axiomatisation of the atomic commands. The soundness of the logic (§2.3) is with respect to a reification of views into concrete states, and is established by proving the axiom soundness property.

In the following sections, we consider numerous instances of the Views Framework. In doing so, we elaborate the metatheory with common constructions and additional proof rules.

Remark. The Views Framework is more general than existing axiomatisations of separation logic [6], in that it does not restrict views to be sets of (machine) states but allows them to be elements in any semigroup. In fact, choosing this views semigroup and an accompanying reification function well is the most important step of using our framework. A good choice of the views semigroup leads to a program logic where a verifier works on the right level of abstraction of machine states. Also, it picks an appropriate scope for the universal quantification in the axiom soundness property, and gives an effective set of axioms for atomic commands. This influence of the views semigroup on axiom soundness corresponds to selecting a notion of locality properties for commands, which was usually done in a fixed manner in the work on separation logic.

3. Constructing Views

We can instantiate of the framework directly, but instantiations often have common structure which can be used to simplify the process. We present two general approaches to constructing views as sets, whose elements themselves belong to a (partial) monoid.

3.1 Separation Algebras

Calcagno *et al.* [6] introduced the concept of *separation algebras* to generalise separation logic. For many examples, we use a generalisation of separation algebras with multiple units [5, 12] (and without the cancellativity requirement) to construct a view monoid.

Parameter H (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 operator* $\bullet : \mathcal{M} \times \mathcal{M} \rightharpoonup \mathcal{M}$ *and a 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 15 (Separation View Monoid) **: Parameter D.** *Each separation algebra* $(\mathcal{M}, \bullet, I)$ *induces a* separation view monoid $(\mathcal{P}(\mathcal{M}), *, I)$*, where* $p_1 * p_2 \overset{\text{def}}{=} \{m_1 \bullet m_2 \mid m_1 \in p_1, m_2 \in p_2\}$.

Separation algebras are typically constructed by adding instrumentation to machine states; this instrumentation determines how states may be composed, typically by recording ownership or invariant properties. While previous work has required cancellativity, we do not.

Remark. For Separation View Monoids, it is common to choose entailment ($\models$) to be subset inclusion ($\subseteq$).

Disjoint Concurrent Separation Logic

To illustrate the separation algebra, we present a simple separation logic for disjoint concurrency.

Definition 16 (DCSL Separation Algebra) **: Parameter H.** *The separation algebra for disjoint concurrent separation logic is* $(\mathcal{M}_{\mathsf{DCSL}}, \uplus, \{\emptyset\})$*, where* $\mathcal{M}_{\mathsf{DCSL}} \stackrel{\text{def}}{=} (\mathsf{Var} \uplus \mathsf{Loc}) \rightharpoonup_{\text{fin}} \mathsf{Val}$*,* $\uplus$ *is the union of partial functions with disjoint domains, and the unit,* $\emptyset$*, is the partial function with the empty domain.*

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. Thus, p and q describe resources, which hold information about part of the state.

Judgements of disjoint concurrent separation logic are, as in the Views Framework, triples of the form $\vdash \{p\}\, C\, \{q\}$. The view $\boldsymbol{x} \Rightarrow v$ denotes the singleton set of the partial function mapping variable $\boldsymbol{x}$ to value v, and $\boldsymbol{x} \Rightarrow \_$ denotes the set of all partial functions that map only 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.

Definition 17 (DCSL Axiomatisation) **: Parameter E.** *The axiomatisation for separation logic is given by the schemas:*

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

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

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

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

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

For the soundness of a view model based on separation algebra, we typically provide a reification function for elements of the separation algebra. This is then lifted to give a reification for views.

Parameter I (Separation Algebra Reification)**.** $\lfloor - \rfloor : \mathcal{M} \to \mathcal{P}(\mathcal{S})$.

Definition 18 : Parameter F. $\lfloor p \rfloor = \bigcup_{m \in p} \lfloor m \rfloor$.

Given this construction of a view monoid and reification function, Property G has an equivalent formulation that is typically simpler to check.

Property J (Axiom Soundness II)**.** *For every* $(p, \alpha, q) \in \mathsf{Axiom}$*, and every* $m \in \mathcal{M}$*,* $[\![\alpha]\!] \lfloor p * \{m\} \rfloor \subseteq \lfloor q * \{m\} \rfloor$.

Remark. The case where $\alpha = \mathsf{id}$ will always hold if $p \vDash q$ implies that $p \subseteq q$, justifying subset inclusion as a natural choice for the entailment relation.

Disjoint Concurrent Separation Logic: Soundness

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 reification.

Definition 19 (DCSL Reification) **: Parameter I.** *The DCSL reification,* $\lfloor - \rfloor_{\mathsf{DCSL}} : \mathcal{M}_{\mathsf{DCSL}} \to \mathcal{P}(\mathcal{S})$*, is defined as follows:*

$$\lfloor s \rfloor_{\mathsf{DCSL}} \stackrel{\text{def}}{=} \{s\}.$$

Our axioms for atomic commands are sound in the sense of Property J. Taking entailment to be $\subseteq$ gives the standard rule of consequence.

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

3.2 Separation Algebras with Interference

Views model partial, abstracted information about machine states that is stable — immune to interference from other threads. In logics with fine-grained permissions such as those used in deny-guarantee (DG) [13] and concurrent abstract predicates (CAP) [9], the elements of the separation algebra are not stable by construction, but an additional obligation is added to only mention stable sets of elements. This can simply be seen as another way of constructing a view monoid from a separation algebra and an *interference relation*.

Parameter K (Interference Relation)**.** *An* interference relation $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ *on a separation algebra* $(\mathcal{M}, \bullet, I)$ *is a preorder satisfying the properties:*

- *for all* $m_1, m_2, m, m' \in \mathcal{M}$ *with* $m = m_1 \bullet m_2$ *and* $m\ \mathcal{R}\ m'$*, there exist* $m_1', m_2' \in \mathcal{M}$ *with* $m_1\ \mathcal{R}\ m_1'$*,* $m_2\ \mathcal{R}\ m_2'$ *and* $m' = m_1' \bullet m_2'$*; and*
- *for all* $i \in I$ *and* $m \in \mathcal{M}$ *with* $i\ \mathcal{R}\ m$*,* $m \in I$*.*

Definition 20 (Stabilised View Monoid) **: Parameter D.** *An interference relation* $\mathcal{R}$ *on a separation algebra* $(\mathcal{M}, \bullet, I)$ *generates a* stabilised view monoid $(\mathcal{R}(\mathsf{View}_{\mathcal{M}}), *, I)$*, where*

$$\mathcal{R}(\mathsf{View}_{\mathcal{M}}) = \{p \in \mathcal{P}(\mathcal{M}) \mid \mathcal{R}(p) \subseteq p\}$$

The notation $\mathcal{R}(p)$ *means* $\{m \in \mathcal{M} \mid \exists m_p \in p.\, m_p\ \mathcal{R}\ m\}$*. The composition* $*$ *is as in Definition 15. That* $\mathcal{R}(\mathsf{View}_{\mathcal{M}})$ *is closed under* $*$ *and includes* I *follows from the conditions in Parameter K.*

Remark. Unlike in CAP [9] and DG [13], we do not need to provide a guarantee relation to say what a thread can do. That is dealt with by axiom soundness.

In this setting, Property G has an equivalent formulation that is typically simpler to check.

Property L (Axiom Soundness III)**.** *For every* $(p, \alpha, q) \in \mathsf{Axiom}$*, and every* $m \in \mathcal{M}$ *then* $[\![\alpha]\!](\lfloor p * \{m\} \rfloor) \subseteq \lfloor q * \mathcal{R}(\{m\}) \rfloor$.

Separation Logic with Ownership

We illustrate how interference can be used to construct views, by showing how separation logic can be constructed this way. Rather than using the separation algebra introduced in Section 3.1, we construct a view monoid 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".

Definition 21 (MSL Separation Algebra) **: Parameter H.** *The masked-separation-logic separation algebra is* $(\mathcal{M}_{\mathsf{MSL}}, \bullet_{\mathsf{MSL}}, I_{\mathsf{MSL}})$*, where*

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

$$\begin{aligned} m_1 \bullet m_2 = m \iff & \\ & \mathrm{dom}\,(m_1) = \mathrm{dom}\,(m_2) = \mathrm{dom}\,(m) \\ & \wedge \forall k \in \mathrm{dom}\,(m)\,.\, m_1(k)\!\downarrow_1 = m_2(k)\!\downarrow_1 = m(k)\!\downarrow_1 \\ & \qquad \wedge\ m_1(k)\!\downarrow_2 + m_2(k)\!\downarrow_2 = m(k)\!\downarrow_2 \end{aligned}$$

$$\textit{and } I \stackrel{\text{def}}{=} \{m \in \mathcal{M}_{\mathsf{MSL}} \mid \forall k \in \mathrm{dom}\,(m)\,.\, m(k)\!\downarrow_2 = 0\}\,.$$

where $\downarrow_i$ *accesses the* i*-th component of a tuple.*

For each variable (or address), the first component of the partial function represents its value, and the second component indicates whether or not the variable (or location) is owned. If a variable, or location, is undefined then it is not allocated. Composition requires that the state components 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.

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

Definition 22 (MSL Interference Relation) **: Parameter K.**

$$m \mathrel{\mathcal{R}} m' \stackrel{\text{def}}{\iff} \forall k \in \mathrm{dom}\,(m)\,.\, m(k)\!\downarrow_2 > 0 \implies m'(k) = m(k)$$

This 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 consider the view monoid induced by the separation algebra under this interference relation, $\mathcal{R}(\mathsf{View}_{\mathsf{MSL}})$, 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. Thus, 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: 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:

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

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

This construction of separation logic parallels the approach of implicit dynamic frames [28]. There, assertions about the state and about ownership are decoupled. Assertions are required to be *self framing*, a property which corresponds directly to stability.

Separation logic with fractional permissions [4] can be defined by using fractions from the interval $[0, 1]$ instead of a bit mask. In §7.5, we discuss how the more complex permissions model of CAP can also be expressed using separation algebras with interference.

4. Generalised Frame Rule

When reasoning uses a context, it is common to have weakening rules on that context, such as the weakening rule of the type system in §2.2. In the views representation, weakening rules become rules that perform the same manipulation on the pre- and postcondition of the specification. In the simple type system, the weakening rule corresponded to the frame rule. However, for certain examples the frame rule is not powerful enough to represent the required context manipulation directly. We therefore introduce a generalised version of the frame rule.

Rely-Guarantee

We motivate the need for generalized frame rules by considering the rely-guarantee method [20]. Rely-guarantee judgements take the form $R, G \vdash \{P\}\ C\ \{Q\}$. Here, P and Q are assertions (interpreted as sets of states). R and G are relations on states — *rely* and *guarantee* relations — that describe how the environment and the program C respectively are allowed to update the state. The assertions P and Q are required to be stable under R.

The atomic step rule takes the following form:

$$\frac{[\![a]\!]\!\downarrow_P \subseteq G \quad [\![a]\!](P) \subseteq Q}{R, G \vdash \{P\}\ a\ \{Q\}} \tag{2}$$

where $A\!\downarrow_P \stackrel{\text{def}}{=} \{(s, s') \in A \mid s \in P\}$. The first condition ensures that G describes the possible behaviours of the program.

The rely-guarantee method is epitomised by the following rule for parallel composition:

$$\frac{\begin{array}{cc} G_1 \subseteq R_2 & R_1, G_1 \vdash \{P_1\}\ C_1\ \{Q_1\} \\ G_2 \subseteq R_1 & R_2, G_2 \vdash \{P_2\}\ C_2\ \{Q_2\} \end{array}}{R_1 \cap R_2, G_1 \cup G_2 \vdash \{P_1 \cap P_2\}\ C_1 \| C_2\ \{Q_1 \cap Q_2\}}$$

This rule requires that the behaviours of each thread, abstracted by G_1 and G_2, are contained in the interference expected by the other thread, represented by R_2 and R_1. The context of the composition relies on the environment doing no more than both threads expect, while the guarantee must account for the behaviours of each thread.

The above rules suggest encoding rely-guarantee into the Views Framework with views consisting of triples (P, R, G) of assertions and rely and guarantee relations, where P is stable under R. Composition would then follow the parallel rule, resulting in the special inconsistent view $\bot$ when the relies and guarantees do not meet the side-conditions.

Definition 23 (RG View Monoid) **: Parameter D.** *The set* View_{RG} *of rely-guarantee views is defined as:*

$$\left\{(P, R, G) \in \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{S}^2) \times \mathcal{P}(\mathcal{S}^2) \;\middle|\; R(P) \subseteq P\right\} \uplus \{\bot\}\,.$$

Composition is defined as:

$$(P_1, R_1, G_1) * (P_2, R_2, G_2) = (P_1 \cap P_2, R_1 \cap R_2, G_1 \cup G_2) \quad \text{provided } G_1 \subseteq R_2 \text{ and } G_2 \subseteq R_1;$$

with all other cases resulting in $\bot$*. The unit is* $(\mathcal{S}, \mathcal{S}^2, \emptyset)$.

We encode the rules by placing the rely and guarantee both into the pre- and postcondition.

$$R, G \vdash \{P\}\ C\ \{Q\} \stackrel{\text{def}}{\iff} \vdash \{(P, R, G)\}\ C\ \{(Q, R, G)\}\,.$$

For all rely-guarantee judgements we assume the pre- and postcondition are stable with respect to the rely, and thus (P, R, G) and (Q, R, G) are both valid views. Using this encoding, the structural rules (including parallel) follow trivially from the rules of the Views program logic, and the atomic rule (2) can be considered to define a set of atomic axioms (Parameter E).

Views are reified to the set of states satisfying their assertions.

Definition 24 (RG Reification) **: Parameter F.**

$$\lfloor (P, R, G) \rfloor = P \qquad \lfloor \bot \rfloor = \emptyset.$$

Lemma 1 (RG Axiom Soundness) **: Property G.** *If* $[\![a]\!]\!\downarrow_P \subseteq G$ *and* $[\![a]\!](P) \subseteq Q$ *then* $a \Vdash \{(P, R, G)\}\{(Q, R, G)\}$.

Proof. We must show that, for every $(P_1, R_1, G_1) \in \mathsf{View}_{RG}$,

$$[\![a]\!]\lfloor (P, R, G) * (P_1, R_1, G_1) \rfloor \subseteq \lfloor (Q, R, G) * (P_1, R_1, G_1) \rfloor.$$

We can assume composition is defined as the undefined case is trivial, and as reification ignores the rely-guarantee components, we can simplify this to:

$$(G \subseteq R_1 \ \wedge\ G_1 \subseteq R \ \wedge\ R(P) \subseteq P \ \wedge\ R_1(P_1) \subseteq P_1) \implies [\![a]\!](P \cap P_1) \subseteq Q \cap P_1.$$

Showing that the result is in Q follows directly from the second premise, and that it is in P_1 from the first premise using routine relational reasoning. □

While we have established the soundness of most of the rules, we have not yet considered rely-guarantee's weakening rule:

$$\frac{R_1 \subseteq R_2 \quad G_2 \subseteq G_1 \quad R_2, G_2 \vdash \{P\}\ C\ \{Q\}}{R_1, G_1 \vdash \{P\}\ C\ \{Q\}}$$

This rule simply increases the restrictions on the environment: it may do less ($R_1 \subseteq R_2$) but must tolerate more ($G_2 \subseteq G_1$).

If we try to encode the context weakening rule using the frame rule induced by the composition above, we require:

$$R_1 \subseteq R_2 \ \wedge\ G_2 \subseteq G_1 \stackrel{?}{\Longrightarrow} \exists R_3, G_3.\ (P, R_1, G_1) = (P, R_2, G_2) * (P, R_3, G_3).$$

However, this does not hold when $R_2 \not\subseteq G_1$. We could use a more elaborate structure to represent the pair of relations such as the deny-guarantee structure [13], which can then be encoded into our framework. However, this loses the original elegance of rely-guarantee, and does not capture the intuition behind it. Instead, we provide a new generalisation of the frame rule that can capture this reasoning directly.

We introduce a generalised frame rule, which applies a function to the pre- and postconditions of a specification. The rule is parameterised by a function $f : \mathsf{View} \to \mathsf{View}$:

$$\frac{\vdash \{p\}\ C\ \{q\}}{\vdash \{f(p)\}\ C\ \{f(q)\}}$$

By choosing particular functions, f, we can encode a range of reasoning rules, in particular the standard frame rule: for the frame r, pick the function

$$f_{*r} \stackrel{\text{def}}{=} \lambda p.\ p * r.$$

Now, for the rule to be sound we must prove the following property for the f function and the action judgement used in axiom soundness.

Property M (f-step Preservation)**.**

$$\forall \alpha, p, q.\ \ \alpha \Vdash \{p\}\{q\} \implies \alpha \Vdash \{f(p)\}\{f(q)\}.$$

This property is sufficient for the soundness of the generalised frame rule. The proof of soundness is essentially the same as for the frame rule, and has been checked in Coq [11].

Remark. The generalised frame rule is analogous to the frame rule of context logic [5], where contexts are applied by a non-commutative operation to the pre- and postconditions. One could construct a context algebra by taking views as the data and the functions satisfying Property M as the contexts, and thus see the generalised frame rule as an instance of the context logic rule. Conversely, one could view the application of a context as a function satisfying Property M, and hence see the context logic frame rule as an instance of the generalised frame rule.

Rely-Guarantee: Weakening

Returning to our motivating example, we can now show how to encode rely-guarantee weakening rule using the generalised frame rule. The weakening rule can be restated in an equivalent form as:

$$\frac{R, G \vdash \{P\}\ C\ \{Q\}}{R \cap R', G \cup G' \vdash \{P\}\ C\ \{Q\}}$$

Thus we justify this instance of the rule using the generalised frame rule choosing f to be the function that restricts the rely by R', and extends the guarantee by G':

$$f_{(R',G')} \stackrel{\text{def}}{=} \lambda_\perp (P, R, G).\ (P, R \cap R', G \cup G')$$

where $(\lambda_\perp x.\ f(x))\ v$ is $f(v)$ if v is not $\perp$, and is $\perp$ otherwise.

Thus we are required to show the following.

Lemma 2 ($f_{(R',G')}$-step Preservation) **: Property M.**

$$\alpha \Vdash \{(P_1, R_1, G_1)\}\{(P_2, R_2, G_2)\} \implies \alpha \Vdash \{(P_1, R_1 \cap R', G_1 \cup G')\}\{(P_2, R_2 \cap R', G_2 \cup G')\}.$$

This holds because the changes to the rely and guarantee only serve to further restrict the set of environment views that we must consider in the action judgement, and the assumption ensures $R_1 \subseteq R_2$ and $G_1 \supseteq G_2$.

5. Combination Rule

The views proof system we have presented so far has omitted two common rules: disjunction and conjunction. One key reason for this is that such rules are not applicable to every instantiation of the framework. For example, the simple type system of §2.2 does not naturally support a disjunction or conjunction rule.

Both of these rules may be expressed as special cases of a general rule for combining multiple specifications of a single program. This general rule is parameterised by a notion of view combination (such as disjunction or conjunction), which maps indexed families of views into single views. By choosing the index set, I, we can define combination operations at different cardinalities.

Parameter N (View I-Combination)**.** *Assume a function* $\bigoplus_I : (I \to \mathsf{View}) \to \mathsf{View}$.

The combination rule is as follows:

$$\frac{\forall i \in I. \vdash \{p_i\}\ C\ \{q_i\}}{\vdash \left\{\bigoplus_{i \in I} p_i\right\}\ C\ \left\{\bigoplus_{i \in I} q_i\right\}}$$

This rule is justified sound by the following property:

Property O (Primitive Combination)**.** *For all* $\alpha \in \mathsf{Label}$*,*

$$(\forall i \in I.\, \alpha \Vdash \{p_i\}\{q_i\}) \implies \alpha \Vdash \left\{\bigoplus_{i \in I} p_i\right\}\left\{\bigoplus_{i \in I} q_i\right\}.$$

Remark. This combination rule generalises the *generalised frame rule* from the previous section, which is simply a unary instance of the combination rule.

Disjunction. In the case of disjunction, there is often a natural operator $\bigvee$ that has the following properties:

Property P (Join Distributivity)**.** $p * \bigvee_{i \in I} q_i = \bigvee_{i \in I} (p * q_i)$.

Property Q (Join Morphism)**.** $\left\lfloor \bigvee_{i \in I} p_i \right\rfloor = \bigcup_{i \in I} \lfloor p_i \rfloor$.

Together, these two properties imply Property O.

Separation Algebras

When views are constructed from a separation algebra, as in §3.1, the views themselves are sets, and so it is natural to consider set union ($\bigcup$) as a notion of disjunction. By the definition of a separation view monoid (Definition 15), it follows that $\bigcup$ satisfies the join distributivity property (Property P). The join morphism property (Property Q) likewise follows directly from the element-wise definition of reification (Definition 18). Consequently, the obvious rule of disjunction is sound for logics built from separation algebras in this way, such as separation logic.

For a stabilised view monoid (Definition 20), note that the interference relation preserves unions (since it is applied element-wise). From this, it follows that $\bigcup$ again satisfies the required properties to provide a rule of disjunction.

Conjunction. It is possible to give properties analogous to Properties P and Q, relating a conjunction $\bigwedge$ with $\bigcap$. Such properties are sufficient to establish Property O, however, it is not often that an instantiation of Views is equipped with a $\bigwedge$ operator with these properties. For instance, in separation logic, $*$ does not distribute over conjunction. In cases, such as separation logic, where a conjunction rule is sound, Property O can typically used to directly justify it.

It is less common for instantiations to have a conjunction rule than a disjunction rule. One explanation for this is that views typically add instrumentation that is not present in the concrete state (such as type information). This information can evolve independently of the concrete state, leading to different specifications for the same program that make incompatible changes to the instrumentation, which cannot be combined with a conjunction rule.

Disjoint Concurrent Separation Logic

For disjoint concurrent separation logic, consider the binary conjunction operation given by set intersection, $\cap$. We can prove that this gives a sound rule of conjunction by establishing Property O, using two useful facts. Firstly, for views constructed from separation algebras, the action judgement need only account for singleton environment views:

$$\alpha \Vdash \{p\}\{q\} \iff \forall m.\, [\![\alpha]\!]\,(\lfloor p * \{m\}\rfloor) \subseteq \lfloor q * \{m\}\rfloor.$$

(This fact justifies the sufficiency of axiom soundness II (Property J).) Secondly, singleton views are precise:

$$(p_1 \cap p_2) * \{m\} = (p_1 * \{m\}) \cap (p_2 * \{m\}).$$

(Note that this is not a general property of separation algebras, but can be established for separation logic.)

Suppose that $\alpha \Vdash \{p_1\}\{q_1\}$ and $\alpha \Vdash \{p_2\}\{q_2\}$. Using the above facts, and Definition I, we can establish that

$$\begin{aligned}[\![\alpha]\!]\,(\lfloor (p_1 \cap p_2) * \{m\}\rfloor) &\subseteq [\![\alpha]\!]\,(\lfloor p_1 * \{m\}\rfloor) \cap [\![\alpha]\!]\,(\lfloor p_2 * \{m\}\rfloor)\\ &\subseteq \lfloor (q_1 * \{m\})\rfloor \cap \lfloor (q_2 * \{m\})\rfloor\\ &= \lfloor (q_1 \cap q_2) * \{m\}\rfloor.\end{aligned}$$

This is sufficient for Property O to hold for binary intersection.

This argument extends to infinitary intersection. Notably, however, it does not apply to the nullary case, which would give $\vdash \{\top\}\ C\ \{\top\}$ for every program C (where $\top = \mathcal{M}_{\mathsf{DCSL}}$). This would be unsound, since $\lightning \notin \lfloor \top \rfloor$, yet there are programs which may fault when run from an arbitrary initial state.

6. Soundness

In this section, we sketch the soundness of the framework. The following results have been machine checked with Coq, and the proof scripts are available [11].

There are two key properties of the action judgement. The first is that it is closed under the composition of a view, and the second that it is closed under the semantic entailment order (Definition 12).

Lemma 3 (Basic Locality)**.** *For all $p, q, r \in \mathsf{View}$, $\alpha \in \mathsf{Label}$, if $\alpha \Vdash \{p\}\{q\}$ then $\alpha \Vdash \{p * r\}\{q * r\}$.*

Lemma 4 (Basic $\preceq$-closure)**.** *For all $p, p', q, q' \in \mathsf{View}$, $\alpha \in \mathsf{Label}$, if $p \preceq p'$, $\alpha \Vdash \{p'\}\{q'\}$ and $q' \preceq q$ then $\alpha \Vdash \{p\}\{q\}$.*

To prove the soundness, we use a semantic judgment that ensures every step of the machine correctly preserves the context using the action judgment. The semantic judgement is defined coinductively: it is a greatest fixed point.

Definition 25 (Semantic Judgement)**.** *For command $C \in \mathsf{Comm}$ and views $p, q \in \mathsf{View}$, the semantic judgement $\Vdash \{p\}\ C\ \{q\}$ is defined to be the maximal relation such that $\Vdash \{p\}\ C\ \{q\}$ holds if and only if the following two conditions are satisfied:*

1. *for all $\alpha \in \mathsf{Label}$, $C' \in \mathsf{Comm}$, 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 $C = \mathtt{skip}$ then $p \preceq q$.*

We can understand the semantic judgment, $\Vdash \{p\}\ C\ \{q\}$, as creating a trace of views for each possible interleaving of the commands in C, where each trace starts in p, and if it is a finite trace, then it ends in q. That is, if $\alpha_1\alpha_2\ldots\alpha_n$ is a finite trace of C, then there exists a sequence $p_1 \ldots p_{n+1}$, such that $p = p_1$, $p_{n+1} \preceq q$ and $\alpha_i \Vdash \{p_i\}\{p_{i+1}\}$. If $\alpha_1\alpha_2\ldots$ is an infinite trace of C, then there exists a sequence $p_1p_2\ldots$, such that $p = p_1$ and $\alpha_i \Vdash \{p_i\}\{p_{i+1}\}$.

For each of the proof rules there is a corresponding lemma establishing that it holds for the semantic judgement. We present two particularly interesting ones here.

Lemma 5 (Soundness of Frame)**.** *If $\Vdash \{p\}\ C\ \{q\}$ then*

$$\Vdash \{p * r\}\ C\ \{q * r\}.$$

Proof. By coinduction. The case $C = \mathtt{skip}$ is trivial. Assume that $\Vdash \{p\}\ C\ \{q\}$ and $C \xrightarrow{\alpha} C'$. By these assumptions, there exists p' such that $\alpha \Vdash \{p\}\{p'\}$ and $\Vdash \{p'\}\ C'\ \{q\}$. It suffices to show that $\alpha \Vdash \{p * r\}\{p' * r\}$ and $\Vdash \{p' * r\}\ C'\ \{q * r\}$. The first follows from Lemma 3 and the second from the coinductive hypothesis. □

Remark. The soundness of the generalised frame rule and the combination rule are proved by a similar argument to the frame rule. For the combination rule, we require the axiom of choice for the indexing set; this allows us to select a "next" view (p' in the above proof) for each index.

Lemma 6 (Soundness of Parallel)**.** *If $\Vdash \{p_1\}\ C_1\ \{q_1\}$ and $\Vdash \{p_2\}\ C_2\ \{q_2\}$ then $\Vdash \{p_1 * p_2\}\ C_1 \parallel C_2\ \{q_1 * q_2\}$.*

Proof. By coinduction. Assume that $C_1 \parallel C_2 \xrightarrow{\alpha} C'$. Either one thread takes a step or one thread has finished executing.

If the latter, $C' = C_i$, $C_j = \mathtt{skip}$ and $\alpha = \mathsf{id}$, where $\{i, j\} = \{1, 2\}$. It must be that $p_j \preceq q_j$, so $\mathsf{id} \Vdash \{p_i * p_j\}\{p_i * q_j\}$ by definition of $\preceq$. Furthermore, $\Vdash \{p_i * q_j\}\ C_i\ \{q_i * q_j\}$ follows from Lemmas 5.

If the former, assume that $C' = C_1' \parallel C_2$ and $C_1 \xrightarrow{\alpha} C_1'$. (The case where C_2 reduces is similar.) By assumption, there exists p_1' such that $\alpha \Vdash \{p_1\}\{p_1'\}$ and $\Vdash \{p_1'\}\ C_1\ \{q_1\}$. It suffices to show that $\alpha \Vdash \{p_1 * p_2\}\{p_1' * p_2\}$ and $\Vdash \{p_1' * p_2\}\ C_1' \parallel C_2\ \{q_1 * q_2\}$. The first follows from Lemma 3 and the second from the coinductive hypothesis. □

By similar lemmas for the other proof rules, we establish that the proof judgement of the logic implies the semantic judgement.

Lemma 7. *If $\vdash \{p\}\ C\ \{q\}$ 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 8. *If $\Vdash \{p\}\ C\ \{q\}$, then for all $s \in \lfloor p \rfloor$ and $s' \in \mathcal{S}$, if $(C, s) \to^* (\mathtt{skip}, s')$ then $s' \in \lfloor q \rfloor$.*

This lemma follows from the definitions. Soundness now follows immediately from the preceding two lemmas.

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

7. Further Examples

We provide five additional instantiations to illustrate the flexibility of our metatheory: recursive types, unique types, Owicki-Gries method, separation logic with resource invariants, and Concurrent Abstract Predicates. The recursive types example demonstrates that Views captures more realistic types. It also illustrates how the rule of consequence can be used for subtyping. The connection between strong update type systems and separation logic has been an open problem for around ten years; the unique types example makes tentative steps towards connecting them using the Views Framework. The Owicki-Gries example illustrates that even underlying traditionally non-compositional reasoning is a compositional core. The penultimate example shows how resource invariants can be added to the views formulation of separation logic by using the generalised frame rule. The final example, CAP, illustrates that the metatheory scales to recent complex logics.

7.1 Recursive Types

In this example, we give a type system for references to pairs and recursive types, which allows us to represent simple lists and trees. The syntax of types is

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

The type null represents the type inhabited just by the value 0. The reference type, $\mathsf{ref}\ \tau_1\ \tau_2$, is a pointer to a pair of locations with types τ_1 and τ_2 respectively. The nullable type $\tau?$ allows null, as well as values of type τ. Note that ref cannot be a null pointer Finally, the fixed-point operator and type variables make it possible to represent recursive data types. For example, we can define lists as:

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

As with the simple type system, views consist of type contexts and the inconsistent context. Unlike in that example, however, we compose typing contexts with disjoint union. This allows us to consider variables as uniquely owned by threads, rather than shared, and consequently reason about strong (type-changing) updates of variables.

Definition 26 (Recursive Type View Monoid) **: Parameter D.** *The view monoid for recursive types is* $((\mathsf{Var} \rightharpoonup \mathsf{Type})_\perp, \uplus_\perp, \emptyset)$.

We extend the heap-update language introduced in §2.1 to work with pairs instead of single heap cells. We assume that values are integers, and locations are positive integers. $x := \mathsf{ref}\ y\ z$ allocates a pair of consecutive locations in the heap, initialising them with values y and z. The first and second components of a pair x may be accessed independently by $x.\mathsf{fst}$ and $x.\mathsf{snd}$. To encode case splitting on a pointer being null we use two commands: $?x{=}0$, which blocks if x is not null, and performs the identity action otherwise; and $?x{\neq}0$ which blocks if x is null, and performs the identity action otherwise. These allow us to encode choice as

$$\mathsf{ifNull}\ x\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \stackrel{\text{def}}{=} (?x{=}0; C_1) + (?x{\neq}0; C_2)$$

Since we wish to allow type-changing updates, commands are specified with both a pre- and a post-type context: $\Gamma_1 \vdash C \dashv \Gamma_2$. In the Views Framework, we naturally interpret this as syntax for $\vdash \{\Gamma_1\}\ C\ \{\Gamma_2\}$.

In order to introduce nullable types, and to fold and unfold recursive types, we introduce a notion of subtyping. The subtyping relation, $\prec$, is defined as the least pre-order satisfying:

$$\frac{}{\tau \prec \tau?} \qquad \frac{}{\mathsf{null} \prec \tau?}$$

$$\frac{}{\mu X.\,\tau \prec \tau[\mu X.\,\tau/X]} \qquad \frac{}{\tau[\mu X.\,\tau/X] \prec \mu X.\,\tau}$$

Definition 27 (Context Subtyping)**.** *Subtyping is lifted to contexts in the obvious way:*

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

Definition 28 (Recursive Type Axiomatisation) **: Parameter E.** *The typing rules for atomic commands include the following:*

$$\frac{}{x : \tau? \vdash ?x{=}0 \dashv x : \mathsf{null}} \qquad \frac{}{x : \tau? \vdash ?x{\neq}0 \dashv x : \tau}$$

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

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

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

Typing rules for the snd *component are analogous to those for* fst. *We give the axioms for entailment as*

$$\Gamma_1 \vDash \Gamma_2 \iff \Gamma_1 \prec \Gamma_2$$

The challenging aspect of this instantiation is giving the reification, which requires us to define a notion of being well-typed. We define a relation $\Theta \vdash v : \tau$ that expresses that a value v has type τ in a heap typing Θ, that maps locations to types. Note that values range over integers.

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

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

We can then define reification as the set of states that are well typed with respect to a view. That is, those states for which there is a heap typing such that the value of each variable and heap location is correctly typed by the view and the heap typing.

Definition 29 (Recursive Type Reification) **: Parameter F.**

$$s \in \lfloor \Gamma \rfloor_\mu \iff \exists \Theta. \forall x \in \mathrm{dom}(\Gamma \cup \Theta). \Theta \vdash s(x) : (\Gamma \cup \Theta)(x)$$

To show the type system is sound we simply show axiom soundness (Property G) for the axioms and subtyping relation.

7.2 Unique Types

The previous example used an atomic allocate and instantiate instruction, $x := \mathsf{ref}\ y\ z$. But this is not 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

$$\frac{}{x : \_ \vdash x := \mathsf{ref} \dashv x : \mathsf{uref}\ \mathsf{val}\ \mathsf{val}}$$

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

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 use the subtyping relation to cast away the uniqueness by adding the following to the subtype relation:

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

We define a function that forgets the uniqueness information, so that we can reuse the previous definitions:

$$\lfloor\!\lfloor \Gamma \rfloor\!\rfloor \stackrel{\text{def}}{=} \lambda x.\, \text{if } \Gamma\, x = \mathsf{uref}\ \tau_1\ \tau_2 \text{ then } \mathsf{ref}\ \tau_1\ \tau_2 \text{ else } \Gamma\, x$$

Our notion of a memory satisfying a type context is the same as before with an additional constraint that each unique reference can be removed from the memory and the memory is well-typed without it in the smaller context.

$$s \in \lfloor \Gamma \rfloor_{u\mu} \stackrel{\text{def}}{=} s \in \lfloor \lfloor\!\lfloor \Gamma \rfloor\!\rfloor \rfloor_{\mu} \wedge \forall x : \mathsf{uref}\ \_\ \_ \in \Gamma.\, s\lfloor_{s(x),s(x+1)} \in \lfloor \lfloor\!\lfloor \Gamma \lfloor_x \rfloor\!\rfloor \rfloor_{\mu}$$

where $f\lfloor_x \stackrel{\text{def}}{=} \lambda y.$ if $x \neq y$ then $f\, y$ else $\bot$. Note this does not mean that the location of a unique reference cannot occur in another stack or heap 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 axiom 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 does not matter. If it is being treated as a location, then it would already be allocated and thus the allocator would not double allocate it.

The axioms and subtyping satisfy the required properties (Property G).

7.3 Owicki-Gries Method

We present a version of the Owicki-Gries method [24] that can be encoded trivially into the Views Framework. However, we omit the auxiliary variable rule, and leave this to future work.

Normally, the Owicki-Gries method has a non-compositional check on parallel composition: each command used by one thread, when restricted to its precondition, preserves all the assertions used in the proof of other thread. We explicitly account for the assertions used in a proof, and the commands and preconditions as contexts to our judgements. Specifically,

$$\mathbb{P}, \mathbb{A} \vdash \{P\}\ C\ \{Q\}$$

means that executing C in a state satisfying P, if it terminates, then the state will satisfy Q, and the proof only uses assertions in $\mathbb{P}$,[1] and its atomic commands are described by $\mathbb{A}$. Thus the rule for an atomic command is

$$\frac{P \in \mathbb{P} \quad Q \in \mathbb{P} \quad (P,a) \in \mathbb{A} \quad [\![a]\!](P) \subseteq Q}{\mathbb{P}, \mathbb{A} \vdash \{P\}\ a\ \{Q\}} \tag{3}$$

The parallel rule is

$$\frac{\mathbb{P}_1, \mathbb{A}_1 \vdash \{P_1\}\ C_1\ \{Q_1\} \quad \forall P \in \mathbb{P}_2, (P',a) \in \mathbb{A}_1.\, [\![a]\!](P \wedge P') \subseteq P \qquad \mathbb{P}_2, \mathbb{A}_2 \vdash \{P_2\}\ C_2\ \{Q_2\} \quad \forall P \in \mathbb{P}_1, (P',a) \in \mathbb{A}_2.\, [\![a]\!](P \wedge P') \subseteq P}{\mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{A}_1 \cup \mathbb{A}_2 \vdash \{P_1 \wedge P_2\}\ C_1 \| C_2\ \{Q_1 \wedge Q_2\}}$$

The two side-conditions (right premises) are the standard "non-compositional" interference freedom checks. Again, as with rely-guarantee, this parallel rule induces a composition operation that we can use to build a view model.

[1] The precise definition actually requires that the proof only uses assertions that are intersections of some elements of $\mathbb{P}$. Otherwise, we would require the parallel rule to additionally require its pre- and postconditions to be in $\mathbb{P}$. This requirement is only evident in the soundness proof, and does not alter the reasoning provided by the Owicki Gries method that we present.

Definition 30 (Owicki-Gries View Monoid) **: Parameter D.** *The Owicki-Gries view monoid* $(\mathsf{View}_{OG}, *, (\emptyset, \emptyset, \mathcal{S}))$ *is defined by*

$$\mathsf{View}_{OG} \stackrel{\text{def}}{=} \left\{ (\mathbb{P}, \mathbb{A}, P) \;\middle|\; \exists \mathbb{P}' \subseteq \mathbb{P}.\, P = \bigcap \mathbb{P}' \right\} \uplus \{\bot\}$$

with composition given by

$$(\mathbb{P}_1, \mathbb{A}_1, P_1) * (\mathbb{P}_2, \mathbb{A}_2, P_2) \stackrel{\text{def}}{=} (\mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{A}_1 \cup \mathbb{A}_2, P_1 \cap P_2)$$
$$\textit{provided } \forall P \in \mathbb{P}_1, (P',a) \in \mathbb{A}_2.\, [\![a]\!](P \wedge P') \subseteq P$$
$$\forall P \in \mathbb{P}_2, (P',a) \in \mathbb{A}_1.\, [\![a]\!](P \wedge P') \subseteq P$$

and otherwise defined to be $\bot$.

The proof rules follow from the Views Framework using the obvious encoding. The atomic rule (3) determines the atomic axioms (Parameter E). We can provide a weakening rule, that allows the context to be enlarged, as a simple application of the generalised frame rule.

For soundness, we define reification (Parameter F) as simply taking the assertion component of a view (and $\emptyset$ in the case of $\bot$). Axiom soundness (Property G) follows from the premises of (3) by the consistency requirement of composition. Step preservation (Property M) for weakening follows trivially from axiom soundness, given that the atomic rule (3) is closed under enlarging the context.

7.4 Atomic CSL

We consider a concurrent separation logic with a single resource invariant that must be preserved by each atomic command [25]. This logic has a simple rule for atomic commands:

$$\frac{\vdash_{\mathsf{DCSL}} \{I * P\}\ a\ \{I * Q\}}{I \vdash \{P\}\ a\ \{Q\}}$$

If the atomic command meets the specification in standard separation logic (§3.1) with the invariant, I, added to both the pre- and postcondition, then it meets the specification in this logic. We interpret this in the Views Framework by defining views to be pairs of shared invariants and local assertions. Composition on these views requires invariants to agree.

Definition 31 (Atomic CSL View Semigroup) **: Parameter D.** *The atomic CSL view semigroup is*

$$((\mathsf{View}_{\mathsf{DCSL}} \times \mathsf{View}_{\mathsf{DCSL}}) \uplus \{\bot\}, *_{\mathsf{ACSL}})$$

where $(I_1, P_1) *_{\mathsf{ACSL}} (I_2, P_2)$ *is defined to be* $(I_1, P_1 *_{\mathsf{DCSL}} P_2)$ *if* $I_1 = I_2$*, and* $\bot$ *otherwise.*

We encode the judgement $I \vdash \{P\}\ C\ \{Q\}$ as

$$\vdash \{(I,P)\}\ C\ \{(I,Q)\}\,.$$

There are two rules for manipulating the context (the resource invariant). The first is a frame rule for the resource invariant, and the second allows local state to be added to the resource invariant:

$$\frac{I \vdash \{P\}\ C\ \{Q\}}{I * I' \vdash \{P\}\ C\ \{Q\}} \qquad \frac{I * I' \vdash \{P\}\ C\ \{Q\}}{I \vdash \{P * I'\}\ C\ \{Q * I'\}}$$

These rules can be interpreted as instances of the generalised frame rule. For the invariant frame rule, the following function is suitable:

$$f_h(I') \stackrel{\text{def}}{=} \lambda_{\bot}(I,P).\, (I * I', P).$$

For the local invariant rule, a more complicated function is necessary:

$$f_s(I, I') \stackrel{\text{def}}{=} \lambda_{\bot}(I_1, P).\, \text{if } I_1 = I * I' \text{ then } (I, P * I') \text{ else } \bot.$$

The function ensures that the resource invariant in the premise is $I * I'$, if not then the function maps to the inconsistent view, thus making the rule and the property vacuous.

For soundness, we define the reification to take the composition of the invariant and local-state assertion in the sense of DCSL.

Definition 32 (Atomic CSL Reification) **: Parameter F.**

$$\lfloor (I, P) \rfloor \stackrel{\text{def}}{=} I *_{\mathsf{DCSL}} P \qquad\qquad \lfloor \bot \rfloor \stackrel{\text{def}}{=} \emptyset$$

This is the only example we present in the paper that does not have a unit. However, the model does have a zero, $\bot$, and satisfies

$$\forall x.\, \exists u.\, x * u = x \wedge \forall y.\, y * u = y \vee y * u = \bot.$$

This property is sufficient to show the simplification of $\alpha \Vdash \{p\}\{q\}$ given in (1), provided $\lfloor \bot \rfloor = \emptyset$.

Axiom soundness (Property G) can easily be established given the soundness of DCSL. The context rules require us to prove Property M. For the first, we can appeal to the frame rule of DCSL, with the frame I'. For the second, the function $f_s(I, I')$ either renders the judgement trivial or preserves the reification, so Property M holds.

Remark. If we wish to establish the conjunction rule for this logic, we can do so if the resource invariant is precise[2], given that assume DCSL has a conjunction rule. We must show (for the binary case)

$$\alpha \Vdash \{(I, P_1)\}\{(I, Q_1)\} \wedge \alpha \Vdash \{(I, P_2)\}\{(I, Q_2)\} \implies \alpha \Vdash \{(I, P_1 \wedge P_2)\}\{(I, Q_1 \wedge Q_2)\}.$$

Using the DCSL rule of conjunction, we know

$$\alpha \Vdash_{\mathsf{DCSL}} \{(I * P_1) \wedge (I * P_2)\}\{(I * Q_1) \wedge (I * Q_2)\}.$$

Using the DCSL rule of consequence, we know

$$\alpha \Vdash_{\mathsf{DCSL}} \{I * (P_1 \wedge P_2)\}\{(I * Q_1) \wedge (I * Q_2)\}.$$

And, moreover, if I is precise, then this implies

$$\alpha \Vdash_{\mathsf{DCSL}} \{I * (P_1 \wedge P_2)\}\{I * (Q_1 \wedge Q_2)\}.$$

This establishes primitive combination (Property O), which is sufficient for the conjunction rule to be sound.

7.5 Concurrent Abstract Predicates

Concurrent Abstract Predicates (CAP) [9] 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:

$$\exists r.\, [\textsc{Lock}]^r_\pi * \boxed{\left(x \mapsto 0 * [\textsc{Unlock}]^r_1\right) \vee x \mapsto 1}^r_{I(r)} \qquad (4)$$

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 $I(r)$:

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

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:

$$\exists r.\, [\textsc{Unlock}]^r_1 * [\textsc{Lock}]^r_\pi * \boxed{x \mapsto 1}^r_{I(r)} \qquad (5)$$

Whereas in (4) the thread does not know whether the mutex is locked or not, in (5) 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.)

In the rest of this section, we informally describe the parameters for the Views Framework. We do not give their formal details, and direct the reader to Dinsdale-Young's thesis [8] and our technical report [11] for the full definition and explanation.

CAP assertions are modelled by sets of states instrumented with ownership, capabilities and region information. Each instrumented state consists of three components. The first is the thread-local state, which consists of capability resources as well as separation-logic-style heap resources. The second is a mapping that associates region identifiers with their states. The third is an interference environment that associates the action identifiers used in capabilities with their interpretations as actions. Instrumented states form a separation algebra (Parameter H): composition of instrumented states is defined as requiring equality between the second and third components, while combining the resources in the first component. For the formal definition, see [8, §8.3.1].

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 (Parameter K), which allows the environment to update regions in any way for which it can have a suitable capability. The interference relation is constructed as the transitive closure of three relations, which model region update (in accordance with actions the environment may have capability to perform), region construction and region destruction. See [8, p319, Definition 8.14] for the formal definition and [8, p337, Lemma 112] for the proof of the decomposition property.

Reification (Parameter I) for CAP is given by combining the heap components of the local and shared states, dropping the capabilities and interference environment. The atomic rule of CAP effectively establishes axiom soundness (Property L) by means of a guarantee relation that obliges atomic operations to preserve stable frames [8, p345, Lemma 125; p338, Lemma 114].

Remark. In our MSL example, 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.

8. 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

[2] I is precise, iff $\forall p, q.\, (I * p) \wedge (I * q) \implies I * (p \wedge q)$.

for concurrent languages [9, 13, 15, 31]. 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 [10, 19, 21, 22].

The soundness of Pottier's capability system [26] is based on an axiom that is similar to our definition of interference relation, and the soundness proof of concurrent abstract predicates [9] 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 [14] 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.

9. 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, rely-guarantee, the Owicki-Gries method and type systems can be understood in an elegant, unifying setting. While the power of the Views Framework is in the variety of systems that it encapsulates, its virtue is in providing a common semantic footing for them: each of the approaches we have studied can be proved sound by appeal to the general soundness result of Views, without recourse to induction over the operational semantics.

We have shown how Views captures a broad selection of existing reasoning systems. Of course, the true test is whether Views can be used to develop new and interesting systems. We are already finding this to be the case. Concurrent abstract predicates has been extended with higher-order features, using Views extended with step-indexing to prove soundness [30]. The Views Framework has been extended to reason about C$^\sharp$ with interesting permission annotations to ensure isolation between threads [17]. Views are also helping in the development of sound logics for local reasoning about intrinsically structured data, such as file systems [16]; in this setting, decomposing data into fragments that record how they are connected is central, and is elegantly justified by the semantic entailment of Views.

In 2002, Reynolds noted the close relationship between separation logic and type systems and wondered "whether the dividing line between types and assertions can be erased" [27]. We have shown that, at least in the first-order case, types and assertions are simply different kinds of *view*. Ultimately, we have identified core principles underlying compositional reasoning about concurrent programs, and formalised them in a unifying framework: Views.

Acknowledgments We thank Tony Hoare, Peter O'Hearn, Azalea Raad, John Wickerson, Adam Wright and the anonymous referees of POPL'12, LICS'12, CONCUR'12 and POPL'13 for helpful feedback and comments on the paper. Dinsdale-Young, Gardner and Yang acknowledge support from EPSRC.

References

[1] A. Ahmed, M. Fluet, and G. Morrisett. L^3: A linear language with locations. *Fundam. Inform.*, 77(4):397–449, 2007.

[2] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *LMCS*, 2(5:1), 2006.

[3] L. Birkedal, B. Reus, J. Schwinghammer, and H. Yang. A simple model of separation logic for higher-order store. In *ICALP*, 2008.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05*, 2005.

[5] C. Calcagno, P. Gardner, and U. Zarfaty. Local reasoning about data update. *ENTCS*, 172:133–175, 2007.

[6] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.

[7] A. Charguéraud and F. Pottier. Functional translation of a calculus of capabilities. In *ICFP*, pages 213–224, 2008.

[8] T. Dinsdale-Young. *Abstract Data and Local Reasoning*. PhD thesis, Imperial College, Department of Computing, 2010.

[9] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[10] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, 2010.

[11] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs (technical report and addditional material). `http://sites.google.com/site/viewsmodel/`, 2012.

[12] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, 2009.

[13] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, pages 363–377, 2009.

[14] X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

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

[16] P. Gardner, G. Nzik, and A. Wright. Reasoning about POSIX file systems using structural separation logic. Draft, 2012.

[17] C. Gordon, M. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*, 2012.

[18] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[19] J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, 2012.

[20] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[21] N. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI*, 2010.

[22] N. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *ICFP*, 2012.

[23] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *TOPLAS*, 21(3):527–568, 1999.

[24] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[25] M. J. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, 2007.

[26] F. Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. Technical report, INRIA, 2011.

[27] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[28] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, 2009.

[29] F. Smith, D. Walker, and J. G. Morrisett. Alias types. In *ESOP*, 2000.

[30] K. Svendsen, L. Birkedal, and M. Parkinson. A specification of the joins library in higher-order separation logic. Technical report, IT University of Copenhagen, 2012.

[31] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.