Views: Compositional Reasoning for Concurrent Programs

Thomas Dinsdale-Young[1], Lars Birkedal[2], Philippa Gardner[1], Matthew J. Parkinson[3], and Hongseok Yang[4]

[1] Imperial College London [2] IT University of Copenhagen
[3] Microsoft Research Cambridge [4] University of Oxford

Abstract. We present a framework for reasoning compositionally about concurrent programs. At its core is the notion of a *view*: an abstraction of the state that takes account of the possible interference due to other threads. Threads' views are composable, and an update to the state by one thread must preserve the views of other threads. We prove soundness for our framework, and demonstrate its utility by studying examples, which include concurrent separation logic and type systems. Our framework is already being used to develop new reasoning systems.

1 Introduction

There has been a recent flurry of research activity on type systems and program logics for reasoning modularly about programs with dynamically allocated, shared mutable state. Type systems have been extended with linear types [23,1,30] and related capability systems [8] that enforce a mixture of local and global properties. Program logics, extending separation logic [25], have been developed to reason about various notions of sharing: for sequential languages, by combining with types [32] and various frame rules [26,29,2]; for concurrent languages by adding invariant [24,17,18,5] or relational reasoning [15,33,13,11].

These developments have led to increasingly elaborate reasoning systems, each introducing new features to tackle specific applications of modular reasoning and new metatheory to justify these features. Despite their ad hoc nature, these systems employ a common approach to compositionality. They provide thread-specific abstractions of the state, which embody enough information to prove the behaviour of a thread whilst allowing for the possible behaviours of other threads. In this paper, we provide a general framework for these abstractions, identifying simple properties necessary for sound, compositional reasoning.

Our fundamental idea is that threads have different *views* of the machine. Intuitively, a thread's view consists of information about the current state of the machine, the *right* of the thread to modify the state as long as the environment's view is stable (invariant) with respect to such changes, and the thread's *right* to the stability of its own view with respect to changes being made by the environment. Threads' views can be *composed*, which ensures that the rights and information held by different threads are compatible with each other.

A thread's view provides a *partial, abstract* description of the state of the machine. It is partial in that it only describes the state relevant to the thread.

It is abstract in that the verifier can use additional information to help with the reasoning, such as types, ghost state or permissions. Such instrumentation has no representation in the concrete state but is a useful fiction for the verifier. To relate the program logic with the operational semantics, we require that a view can be *reified* to a set of concrete machine states. Using reification, we prove a general soundness result, which we have formally verified in Coq.

To illustrate the essential compositionality of views, consider a command C that updates the view p to the view q. For compositional reasoning, we would require that C updates $p * r$ to $q * r$, where r represents any view held by the environment and $*$ is the composition operation on views. Traditional approaches in separation logic have achieved this by enforcing that commands satisfy so-called locality conditions [7]. We take the alternative approach of embedding compositionality into the *meaning* of "C updates p to q": for all r, it must update $p * r$ to $q * r$. This interpretation has been used for extending sequential separation logic for higher-order languages, where it is otherwise difficult to characterise locality [2,3]. We show that the interpretation also gives a simpler and more general metatheory for logics for concurrent programs.

A crucial implication of this interpretation is that views should be stable with respect to any operation that a thread with a consistent view could perform. At one extreme, stability can be enforced by disjointness between views: one thread can access variable $\boldsymbol{x}$, say, while the other cannot have anything to do with $\boldsymbol{x}$. At the other, stability can be enforced by invariant properties: both threads may agree that $\boldsymbol{x}$ always has type bool, for instance. In the middle ground lie many logics that allow controlled sharing. Views capture this whole spectrum.

The views framework embodies the essential ingredients for sound compositional reasoning. This minimality reveals that aspects that seem integral to complex proof systems can in fact be understood as separate concerns. For example, the logic of concurrent abstract predicates [11] models interference with rely and guarantee relations, which pervade the soundness proof. When understood in terms of views, however, interference relations are just a means of constructing the set of views, and have no special significance in the soundness proof.

Our framework provides a recipe for designing new reasoning systems. We demonstrate this by presenting several examples of type systems and program logics in our views framework. Further examples with advanced features can be found in our technical report [10]. The views framework has already proved useful to researchers creating new type systems and program logics, which we discuss in §5.

2 Views Framework

We present the views framework for developing sound program logics, an overview of which is given in Fig. 1. We use a concurrent programming language, parame-

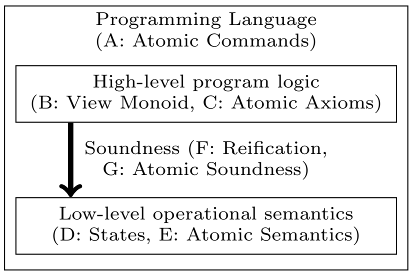
Programming Language
(A: Atomic Commands)
High-level program logic
(B: View Monoid, C: Atomic Axioms)
Soundness (F: Reification,
G: Atomic Soundness)
Low-level operational semantics
(D: States, E: Atomic Semantics)

Fig. 1. Overview of framework

terised by atomic commands (A), with a low-level operational semantics that is parameterised by the notion of state (D) and how the atomic commands manipulate that state (E). We provide a high-level program logic to reason about this system, which is parameterised by its form of assertions — *views* (B) — and its axioms for atomic commands (C). We provide a soundness result for the logic with respect to the operational semantics, which is parameterised by a relationship connecting the high-level assertions to states (F) and a property each axiom must satisfy with respect to the operational semantics (G). Simply defining the Parameters A–F and proving property G will ensure a sound logic.

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

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

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

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

Views form assertions in our program logic. We have one basic requirement for views: that they form a commutative monoid.

Parameter B (View Monoid). *Assume a commutative monoid* $(\mathsf{View}, *, u)$*. The variables* p, q, r *are used to denote elements of* View*.*

Intuitively, views are resources that embody knowledge and rights; combining two views with $*$ produces a view with the knowledge and rights of its components. The resource monoid underlies the logic of Bunched Implications [28], whose models are monoids with additional logical structure.

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

Definition 2 (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:*

$$\frac{(p,a,q) \in \mathsf{Axiom}}{\vdash \{p\}\ a\ \{q\}} \qquad \frac{\vdash \{p\}\ C\ \{q\}}{\vdash \{p * r\}\ C\ \{q * r\}} \qquad \frac{}{\vdash \{p\}\ \texttt{skip}\ \{p\}} \qquad \frac{\vdash \{p\}\ C_1\ \{q\} \quad \vdash \{p\}\ C_2\ \{q\}}{\vdash \{p\}\ C_1 + C_2\ \{q\}}$$

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

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.

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

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

Parameter E (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 transformers. [Where necessary, we lift non-deterministic state transformers to sets of states: for* $S \in \mathcal{P}(\mathcal{S})$*,* $\alpha(S) = \bigcup \{\alpha(s) \mid s \in S\}$*.]*

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

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

Definition 3 (Labelled Transition System and Operational Semantics). *The labelled transition relation* $- \xrightarrow{-} - : \mathsf{Comm} \times (\mathsf{Atom} \uplus \{\mathsf{id}\}) \times \mathsf{Comm}$ *is defined by the following rules, where* α *ranges over* $\mathsf{Atom} \uplus \{\mathsf{id}\}$*:*

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

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

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

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

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

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

Soundness requires that the axioms concerning atomic commands are satisfied by the operational interpretation of the commands. For each axiom (p, a, q), the interpretation of a must update view p to q while preserving any environment view. This is captured by the following property:

$$\textbf{Consequence:}\quad \frac{p \vDash p' \quad \vdash \{p'\}\ C\ \{q\}}{\vdash \{p\}\ C\ \{q\}} \qquad \frac{\vdash \{p\}\ C\ \{q'\} \quad q' \vDash q}{\vdash \{p\}\ C\ \{q\}}$$

$$\textbf{Disjunction:}\quad \frac{\forall i \in I.\vdash \{p_i\}\ C\ \{q\}}{\vdash \left\{\bigvee \{p_i\}_{i\in I}\right\}\ C\ \{q\}} \qquad \textbf{Conjunction:}\quad \frac{\forall i \in I.\vdash \{p\}\ C\ \{q_i\}}{\vdash \{p\}\ C\ \left\{\bigwedge \{q_i\}_{i\in I}\right\}}$$

Fig. 2. Additional proof rules.

Property G (Atomic Soundness). *For every* $(p, a, q) \in \mathsf{Axiom}$*, and every* $r \in \mathsf{View}$ *then* $[\![a]\!]\lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor$*.*

This property is both necessary and sufficient for the soundness of the program logic. We state the soundness result here; proof details, including Coq proof scripts, are available [10].

Theorem 1 (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* $s' \in \lfloor q \rfloor$*.*

Remark 1. The views framework is more general than existing axiomatisations of separation logic [7], in that it does not restrict views to be sets of (machine) states but allows them to be elements in any monoid. In fact, choosing this views monoid and an accompanying reification function well is the most important step of using our framework. A good choice of the views monoid 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 Atomic Soundness property, and gives an effective set of axioms for atomic commands. This influence of the views monoid on Atomic 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.

2.1 Additional Rules

The rules in Definition 2 form the core of our proof system; however, views are often equipped with additional structure, which gives rise to additional rules.

Consequence. When views are equipped with an entailment relation, we can add rules of consequence, given in Fig. 2. (When $\vDash$ is reflexive, these rules can be condensed into a single rule.)

Parameter H (Entailment). *Assume a relation* $\vDash \subseteq \mathsf{View} \times \mathsf{View}$*.*

We can give a semantic notion entailment that comes directly from the composition and reification parameters:

Definition 4 (Semantic Entailment). $p \preceq q \overset{\text{def}}{\iff} \forall r \in \mathsf{View}. \lfloor p*r \rfloor \subseteq \lfloor q*r \rfloor$.

The soundness of the rules of consequence may be justified by the following necessary and sufficient property.

Property I (Entailment Locality). $p \vDash q \implies p \preceq q$*.*

Disjunction. When views are equipped with a notion of disjunction, we can add a rule of disjunction given in Fig. 2.

Parameter J (Disjunction). *Assume a function* $\bigvee : \mathcal{P}(\mathsf{View}) \to \mathsf{View}$.

The soundness of the rule of disjunction is justified when the following two properties hold:

Property K (Join Distributivity). $p * \bigvee \{q_i\}_{i \in I} = \bigvee \{p * q_i\}_{i \in I}$.

Property L (Join-semilattice Morphism). $\lfloor \bigvee \{p_i\}_{i \in I} \rfloor = \bigcup \{\lfloor p_i \rfloor\}_{i \in I}$.

Conjunction. When views are equipped with a notion of conjunction, we can add a rule of conjunction given in Fig. 2.

Parameter M (Conjunction). *Assume a function* $\bigwedge : \mathcal{P}(\mathsf{View}) \to \mathsf{View}$.

The soundness of the rule of conjunction is justified when the following two properties hold:

Property N (Primitive Conjunctivity). *If* $\forall r \in \mathsf{View}.\ [\![a]\!] \lfloor p * r \rfloor \subseteq \lfloor q_i * r \rfloor$ *for all i, then* $\forall r \in \mathsf{View}.\ [\![a]\!] \lfloor p * r \rfloor \subseteq \lfloor (\bigwedge \{q_i\}_{i \in I}) * r \rfloor$.

Property O (Meet Supremum). $\bigwedge \{p_i\}_{i \in I}$ *is the supremum (least upper bound) of* $\{p_i\}_{i \in I}$ *with respect to* $\preceq$.

3 Examples

For illustrative purposes, we use some simple atomic primitives for manipulating the heap. These are standard commands from separation logic [19].

Definition 5 (Atomic Heap Commands). *Assume a set of variable names* Var, *ranged over by* $\boldsymbol{x}$ *and* $\boldsymbol{y}$, *and a set of values* Val, *ranged over by* v, *of which a subset* $\mathsf{Loc} \subseteq \mathsf{Val}$ *represents heap addresses, ranged over by* l. *The syntax of atomic heap commands,* Atom_H, *is defined by the grammar:*

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

Definition 6 (Heap States). *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 7 (Heap Command Semantics). *The semantics of the atomic heap-update commands are given by:*

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

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

Separation Algebras. Calcagno *et al.* [7] introduced the concept of *separation algebras* to generalise separation logic. For many examples, we use a generalisation of separation algebras with multiple units [6,12] (and without the cancellativity requirement) to construct a view monoid.

Definition 8 (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 9 (Separation View Monoid). *Each separation algebra* $(\mathcal{M}, \bullet, I)$ *induces a* separation view monoid $(\mathcal{P}(\mathcal{M}), *, I)$*, where* $p_1 * p_2$ *is defined to be* $\{m_1 \bullet m_2 \mid m_1 \in p_1, m_2 \in p_2\}$.

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. With this flexibility, we can use union (rather than disjoint union) as a separation operator, which leads to views that express global invariant properties.

Remark 2. For Separation View Monoids, subset inclusion ($\subseteq$), union ($\bigcup$) and intersection ($\bigcap$) are natural choices for entailment, disjunction and conjunction respectively for the additional rules.

Disjoint Concurrent Separation Logic. Judgements of disjoint concurrent separation logic are, as in the views framework, triples of the form $\vdash \{p\}\ C\ \{q\}$. Abstractly, the state is treated as a resource, which is divided up by individual variables and heap addresses. Thus, p and q describe resources, which hold information about part of the state. Formally, p and q are views from the separation view monoid induced by the separation algebra $(\mathcal{M}_{\mathsf{DCSL}}, \uplus, \{\emptyset\})$, where $\mathcal{M}_{\mathsf{DCSL}} \stackrel{\text{def}}{=} (\mathsf{Var} \uplus \mathsf{Loc}) \rightharpoonup_{\text{fin}} \mathsf{Val}$. That is, $\mathcal{M}_{\mathsf{DCSL}}$ is the set of finite partial functions from variables and heap addresses to values, with the partial monoid operation given by the union of partial functions with disjoint domains $\uplus$, and the unit consisting of only the partial function with the empty domain, $\emptyset$.

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

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

The axiomatisation for separation logic is given by the schemas:

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

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 10 (DCSL Reification). $\lfloor p \rfloor_{\mathsf{DCSL}} \stackrel{\text{def}}{=} p$.

Our axioms for atomic commands are sound in the sense of Property G. We can also justify the soundness of the additional rules. Taking entailment to be $\subseteq$, it is easy to establish Property I (Entailment Locality). Taking disjunction to be $\bigcup$, Property K (Join Distributivity) holds by construction (as it does whenever views are constructed from separation algebras), and Property L is trivial given the definition of reification. Taking conjunction to be $\bigcap$, it is possible (although not trivial) to establish Property N (Primitive Conjunctivity), while Property O (Meet Supremum) holds trivially.

Remark 3. Although mathematically they may be defined the same way, intuitively a separation-logic state and a heap state represent different things. In a separation-logic state, the partiality of the function means that the rest of the state is unknown and not accessible by the thread. In a heap state, the partiality of the function means that the undefined parts are unallocated.

Reification can be seen as completing the state by treating the unknown regions as unallocated. We could define reification differently, by completing the state in *every possible way*. For example $\lfloor \boldsymbol{x} \Rightarrow 5 \rfloor$ would be the set of all states in which $\boldsymbol{x}$ has value 5. By changing the reification like this, we obtain a slightly different notion of soundness. This reification admits a more permissive notion of entailment where $p * q \vDash p$. Use of this weakening in a proof can be thought of as a thread renouncing ownership of some resource (without ownership transferring elsewhere). This gives “intuitionistic” separation logic [19].

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 separation-logic triples do not permit memory faults to occur, which is part of the standard interpretation.

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

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

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

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

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

Typing contexts may be combined when they agree on the types of all variables that they have in common by taking their union (as relations). We introduce a new context, $\bot$, the *inconsistent typing context*, to represent the result of combining contexts that do not agree. We thus have a view monoid of typing contexts: $((\mathsf{Var} \rightharpoonup \mathsf{Type}) \uplus \{\bot\}, \cup_\bot, \emptyset)$.

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

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 11 (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 12 (Weak Type Reification). $\lfloor \Gamma \rfloor_{\mathsf{WTS}} \overset{\text{def}}{=} \{s \in \mathcal{S}_\mathsf{H} \mid \exists \Theta.\, \Gamma; \Theta \vdash s\}$.

To establish soundness, we need only show Property G (Atomic Soundness). This is straightforward; for further details, consult [10]. 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 [7,12] would not work for this example.

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

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

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

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

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

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

Here, we combine typing contexts only when the variables they type are *disjoint*. This enforces the intended discipline of ownership: threads may modify the types and values of variables that they own, but have no knowledge or rights to any other variables. Whenever there is an overlap between contexts, their combination is $\bot$. This gives a view monoid: $((\mathsf{Var} \rightharpoonup \mathsf{Type}) \uplus \{\bot\}, \uplus_\bot, \emptyset)$.

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

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

Fig. 3. Subtyping rule

We can add the subtyping rule in Fig. 3 to the system, which is an instance of the (left) rule of consequence if we define an entailment relation corresponding to subtyping: $\Gamma, \boldsymbol{x} : \mathsf{ref}\ \tau \vDash \Gamma, \boldsymbol{x} : \mathsf{val}$.

For the strong-update type system, we use an analogous reification to that of Definition 12 for weak update. It is straightforward to prove atomic soundness. For details, see [10]. To justify subtyping, we must establish Property I (Entailment Locality). This amounts to showing that $\lfloor \Gamma, \boldsymbol{x} : \mathsf{ref}\ \tau \rfloor \subseteq \lfloor \Gamma, \boldsymbol{x} : \mathsf{val} \rfloor$, which is the case since in any state in which $\boldsymbol{x}$ holds a reference, $\boldsymbol{x}$ also holds a value.

The framework also tells us what cannot be used as a subtyping judgement, as Property I is necessary. For instance, one might wish to allow subtyping in the heap: $\boldsymbol{x} : \mathsf{ref\ ref}\ \tau \vDash \boldsymbol{x} : \mathsf{ref\ val}$. Although $\lfloor \boldsymbol{x} : \mathsf{ref\ ref}\ \tau \rfloor \subseteq \lfloor \boldsymbol{x} : \mathsf{ref\ val} \rfloor$ does hold, we are required to consider all contexts. However, $\lfloor \boldsymbol{x} : \mathsf{ref\ ref}\ \tau, \boldsymbol{y} : \mathsf{ref\ ref}\ \tau \rfloor \not\subseteq \lfloor \boldsymbol{x} : \mathsf{ref\ val}, \boldsymbol{y} : \mathsf{ref\ ref}\ \tau \rfloor$, because on the left $\boldsymbol{x}$ can equal $\boldsymbol{y}$ but on the right it cannot. Thus Property I would not hold, and this subtyping would be unsound.

Typed Separation Logic. Separation logic has been used to reason about programming languages with types. However, separation-logic reasoning typically ignores the types, although the work of Tan *et al.* [32] is a notable exception. Since type information is shared and global, and separation-logic reasoning is local, it previously seemed difficult to integrate the two systems. However, with our framework, it is easy. The system we present here only allows weak updates of the heap. We also have an example with strong updates [10].

We can axiomatise the atomic commands by combining the axioms of the strong-update type system and separation logic; see the first rule in Fig. 4. If the atomic command is allowed by the type system and the separation logic, then it is allowed in the combined system. We can also add axioms, such as the second and third rules in Fig. 4, which derive from only one of the systems. Importantly, anything that is changed must be allowed by both systems, but the ability to access something only needs to be justified in one of the underlying systems.

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

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

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

Fig. 4. Combined typing and separation logic rules.

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

Definition 13 (TSL reification). $\lfloor \Gamma, p \rfloor_{\mathsf{TSL}} \stackrel{\text{def}}{=} \lfloor \Gamma \rfloor_{\mathsf{STS}} \cap \lfloor p * \mathsf{true} \rfloor_{\mathsf{DCSL}}$.

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

The soundness of the combined axiom follows directly from the soundness of the underlying two axioms. The new axioms' soundness proofs are straightforward and given in the extended version [10].

4 Views and Interference

Views model partial, abstracted information about machine states that is stable — immune to interference from other threads. In logics with more fine-grained permissions such as those used in deny-guarantee (DG) [13] and concurrent abstract predicates (CAP) [11], 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.*

Definition 14 (Interference Relation). *An* interference relation $R \subseteq \mathcal{M} \times \mathcal{M}$ *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 $m \; R \; m'$, there exist $m'_1, m'_2 \in \mathcal{M}$ with $m_1 \; R \; m'_1$, $m_2 \; R \; m'_2$ and $m' = m'_1 \bullet m'_2$; and*
- *for all $i \in I$ and $m \in \mathcal{M}$ with $i \; R \; m$, $m \in I$.*

Definition 15 (Stabilised View Monoid). *An interference relation R on a separation algebra $(\mathcal{M}, \bullet, I)$ generates a* stabilised view monoid *$(R(\mathsf{View}_{\mathcal{M}}), *, I)$, where $R(\mathsf{View}_{\mathcal{M}}) = \{p \in \mathcal{P}(\mathcal{M}) \mid R(p) \subseteq p\}$[1] and $*$ is as in Definition 9. That $R(\mathsf{View}_{\mathcal{M}})$ is closed under $*$ and includes I follows from the conditions in Definition 14*

Remark 4. Unlike in CAP [11] and DG [13], we do not need to provide a guarantee relation to say what a thread can do. That is dealt with by atomic soundness.

We illustrate how interference can be used to construct views, by showing how separation logic can be constructed this way. Rather than the view monoid introduced in Section 3, we construct a 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". Our set of instrumented states is:

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

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

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

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. To complete the separation algebra $(\mathcal{M}_{\mathsf{MSL}}, \bullet, I)$, it remains to give the unit: $I \stackrel{\text{def}}{=} \{m \in \mathcal{M}_{\mathsf{MSL}} \mid \forall k \in \mathrm{dom}\,(m) \,.\, m(k){\downarrow_2} = 0\}$.

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:

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

[1] The notation $R(p)$ means $\{m \in \mathcal{M} \mid \exists m_p \in p.\, m_p \; R \; m\}$.

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, $R(\mathsf{View}_{\mathsf{MSL}})$, 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, since it enforces that the environment cannot also own variables and locations that belong to the thread.) It can be shown that the DCSL monoid is isomorphic to this monoid, so we can consider the separation logic model as being constructed in this way.

We can define a separation logic with fractional permissions [4] by using fractions from the interval $[0, 1]$ instead of a bit mask. This approach also extends to model the logics of CAP [11] and DG [13]; more details can be found in [10].

5 Conclusions and Related Work

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

The examples present in this paper have been intentionally elementary to illustrate the use of the framework. However, many further examples have been encoded into the framework. In the extended version of this paper [10], we extend the typing system to recursive types, to demonstrate that the type system can handle more realistic typing challenges. We have combined this extended type system with unique references to deal with separating allocation and initialisation of data structures, and also demonstrated that the unique references can be replaced by separation logic in a similar system to Tan *et al.* [32]. We also present an encoding of concurrent abstract predicates into the views framework using the interference relation discussed in Section 4.

Our views framework is already being used to develop logics for advanced language features. Concurrent abstract predicates has been extended with higher-order features and the soundness of this uses the views framework extended with step-indexing [31]. Views have also been extended to reason about C$^\sharp$ with interesting permission annotations to ensure isolation between threads [16]. Views have also inspired work on dependent types involving sharing [22].

In on-going work, views are being applied to help construct sound logics for local reasoning about abstract structured data, such as trees. These logics allow structured resource (such as a file system or XML data) to be decomposed into fragments that record how they connect together; this can be justified by the general notion of entailment in the views framework. Views are also helping to construct logics for reasoning about power-fault-tolerant programs and message passing concurrency.

Related Work. Our composition operator provides a logical notion of separation, which, as we have demonstrated by examples, need not be realized by physical separation in the concrete machine. This idea of fictional separation has been used in recent work on separation logics for concurrent languages [11,15,33,13]. 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 [9,21,20].

The soundness of Pottier's capability system [27] is based on an axiom that is similar to our definition of interference relation, and the soundness proof of concurrent abstract predicates [11] 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. LRG is the only combination of rely-guarantee with separation logic we have not encoded into our framework, and remains future work.

References

1. Ahmed, A., Fluet, M., Morrisett, G.: L^3: A linear language with locations. Fundam. Inform. 77(4), 397–449 (2007)
2. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. LMCS 2(5:1) (2006)
3. Birkedal, L., Reus, B., Schwinghammer, J., Yang, H.: A simple model of separation logic for higher-order store. In: ICALP (2008)
4. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL'05 (2005)
5. Buisse, A., Birkedal, L., Støvring, K.: A step-indexed Kripke model of separation logic for storable locks. In: MFPS (2011)
6. Calcagno, C., Gardner, P., Zarfaty, U.: Local reasoning about data update. ENTCS 172, 133–175 (2007)
7. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS. pp. 366–378. IEEE Computer Society (2007)
8. Charguéraud, A., Pottier, F.: Functional translation of a calculus of capabilities. In: ICFP. pp. 213–224 (2008)
9. Dinsdale-Young, T., Gardner, P., Wheelhouse, M.: Abstraction and refinement for local reasoning. In: VSTTE (2010)
10. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Adddi-tional material. `http://sites.google.com/site/viewsmodel/` (2012)
11. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP (2010)
12. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: APLAS (2009)

13. Dodds, M., Feng, X., Parkinson, M.J., Vafeiadis, V.: Deny-guarantee reasoning. In: ESOP. pp. 363–377 (2009)
14. Feng, X.: Local rely-guarantee reasoning. In: POPL (2009)
15. Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: ESOP. pp. 173–188 (2007)
16. Gordon, C., Parkinson, M., Parsons, J., Bromfield, A., Duffy, J.: A low level type system for convertible reference immutability and parallelism (2012), draft
17. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: APLAS (2007)
18. Hobor, A.: Oracle Semantics. Ph.D. thesis, Princeton University, Department of Computer Science, Princeton, NJ (October 2008)
19. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL. pp. 14–26 (2001)
20. Jensen, J.B., Birkedal, L.: Fictional separation logic. In: ESOP (2012)
21. Krishnaswami, N., Birkedal, L., Aldrich, J.: Verifying event-driven programs using ramified frame properties. In: TLDI (2010)
22. Krishnaswami, N., Turon, A., Dreyer, D., Garg, D.: Superficially substructural types (2012), draft
23. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. TOPLAS 21(3), 527–568 (1999)
24. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)
25. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL. pp. 1–19 (2001)
26. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: POPL. pp. 268–280 (2004)
27. Pottier, F.: Syntactic soundness proof of a type-and-capability system with hidden state. Tech. rep., INRIA (2011), (available from the author)
28. Pym, D.J.: The Semantics and Proof Theory of the Logic of Bunched Implications, Applied Logic Series, vol. 26. Springer (2002)
29. Schwinghammer, J., Yang, H., Birkedal, L., Pottier, F., Reus, B.: A semantic foundation for hidden state. In: FOSSACS. pp. 2–16 (2010)
30. Smith, F., Walker, D., Morrisett, J.G.: Alias types. In: ESOP (2000)
31. Svendsen, K., Birkedal, L., Parkinson, M.: Joined-up thinking: A specification of the joins library in higher-order separation logic (2012), draft
32. Tan, G., Shao, Z., Feng, X., Cai, H.: Weak updates and separation logic. In: APLAS. pp. 178–193. Springer-Verlag (2009)
33. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. pp. 256–271 (2007)