A Case for Behavior-Preserving Actions in Separation Logic

David Costanzo and Zhong Shao

Yale University

Abstract. Separation Logic is a widely-used tool that allows for local reasoning about imperative programs with pointers. A straightforward definition of this "local reasoning" is that, whenever a program runs safely on some state, adding more state would have no effect on the program's behavior. However, for a mix of technical and historical reasons, local reasoning is defined in a more subtle way, allowing a program to lose some behaviors when extra state is added. In this paper, we propose strengthening local reasoning to match the straightforward definition mentioned above. We argue that such a strengthening does not have any negative effect on the usability of Separation Logic, and we present four examples that illustrate how this strengthening simplifies some of the metatheoretical reasoning regarding Separation Logic. In one example, our change even results in a more powerful metatheory.

1 Introduction

Separation Logic [8, 13] is widely used for verifying the correctness of C-like imperative programs [9] that manipulate mutable data structures. It supports *local reasoning* [15]: if we know a program's behavior on some heap, then we can automatically infer something about its behavior on any larger heap. The concept of local reasoning is embodied as a logical inference rule, known as the *frame rule*. The frame rule allows us to extend a specification of a program's execution on a small heap to a specification of execution on a larger heap.

For the purpose of making Separation Logic extensible, it is common practice to abstract over the primitive commands of the programming language being used. By "primitive commands" here, we mean commands that are not defined in terms of other commands. Typical examples of primitive commands include variable assignment $x := E$ and heap update $[E] := E'$. One example of a non-primitive command is $\texttt{while}\, B\, \texttt{do}\, C$.

When we abstract over primitive commands, we need to make sure that we still have a sound logic. Specifically, it is possible for the frame rule to become unsound for certain primitive commands. In order to guarantee that this does not happen, certain "healthiness" conditions are required of primitive commands. We refer to these conditions together as "locality," since they guarantee soundness of the frame rule, and the frame rule is the embodiment of local reasoning.

As one might expect, locality in Separation Logic is defined in such a way that it is *precisely* strong enough to guarantee soundness of the frame rule. In other

words, the frame rule is sound *if and only if* all primitive commands are local. In this paper, we consider a strengthening of locality. Clearly, any strengthening will still guarantee soundness of the frame rule. The tradeoff, then, is that the stronger we make locality, the fewer primitive commands there will be that satisfy locality. We claim that we can strengthen locality to the point where: (1) the usage of the logic is unaffected — specifically, we do not lose the ability to model any primitive commands that are normally modeled in Separation Logic; (2) our strong locality is precisely the property that one would intuitively expect it to be — that the behavior of a program is completely independent from any unused state; and (3) we significantly simplify various technical work in the literature relating to metatheoretical facts about Separation Logic. We refer to our stronger notion of locality as "behavior preservation," because the behavior of a program is preserved when moving from a small state to a larger one.

We justify statement (1) above, that the usage of the logic is unaffected, in Section 3 by demonstrating a version of Separation Logic using the same primitive commands as the standard one presented in [13], for which our strong locality holds. We show that, even though we need to alter the state model of standard Separation Logic, we do not need to change any of the inference rules. We justify the second statement, that our strong locality preserves program behavior, in Section 2. We will also show that the standard, weaker notion of locality is not behavior-preserving. We provide some justification of the third statement, that behavior preservation significantly simplifies Separation Logic metatheory, in Section 5 by considering four specific examples in detail. As a primer, we will say a little bit about each example here.

The first simplification that we show is in regard to *program footprints*, as defined and analyzed in [12]. Informally, a footprint of a program is a set of states such that, given the program's behavior on those states, it is possible to infer all of the program's behavior on all other states. Footprints are useful for giving complete specifications of programs in a concise way. Intuitively, locality should tell us that the set of *smallest safe states*, or states containing the minimal amount of resources required for the program to safely execute, should always be a footprint. However, this is not the case in standard Separation Logic. To quote the authors in [12], the intuition that the smallest safe states should form a footprint "fails due to the subtle nature of the locality condition." We show that in the context of behavior-preserving locality, the set of smallest safe states does indeed form a footprint.

The second simplification regards the theory of data refinement, as defined in [6]. Data refinement is a formalism of the common programming paradigm in which an abstract module, or interface, is implemented by a concrete instantiation. In the context of [6], our programming language consists of a standard one, plus abstract module operations that are guaranteed to satisfy some specification. We wish to show that, given concrete and abstract modules, and a relation relating their equivalent states, any execution of the program that can happen when using the concrete module can also happen when using the abstract one.

We simplify the data refinement theory by eliminating the need for two somewhat unintuitive requirements used in [6], called contents independence and growing relations. Contents independence is a strengthening of locality that is implied by the stronger behavior preservation. A growing relation is a technical requirement guaranteeing that the area of memory used by the abstract module is a subset of that used by the concrete one. It turns out that behavior preservation is strong enough to completely eliminate the need to require growing relations, *without* automatically implying that any relations are growing. Therefore, we can prove refinement between some modules (e.g., ones that use completely disjoint areas of memory) that the system of [6] cannot handle.

Our third metatheoretical simplification is in the context of Relational Separation Logic, defined in [14]. Relational Separation Logic is a tool for reasoning about the relationship between two executions on different programs. In [14], soundness of the relational frame rule is initially shown to be dependent on programs being deterministic. The author presents a reasonable solution for making the frame rule sound in the presence of nondeterminism, but the solution is somewhat unintuitive and, more importantly, a significant chunk of the paper (about 9 pages out of 41) is devoted to developing the technical details of the solution. We show that under the context of behavior preservation, the relational frame rule as initially defined is already sound in the presence of nondeterminism, so that section of the paper is no longer needed.

The fourth simplification is minor, but still worth noting. For technical reasons, the standard definition of locality does not play well with a model in which the total amount of available memory is finite. Separation Logic generally avoids this issue by simply using an infinite space of memory. This works fine, but there may be situations in which we wish to use a model that more closely represents what is actually going on inside our computer. While Separation Logic can be made to work in the presence of finite memory, doing so is not a trivial matter. We will show that under our stronger notion of locality, no special treatment is required for finite-sized models.

The remainder of this paper is structured as follows: Section 2 describes the notion of locality employed by Separation Logic, as well as our stronger, behavior-preserving notion; Section 3 presents a version of Separation Logic in which all programs are behavior-preserving; Section 4 places behavior preservation in an abstract setting in preparation for discussing Separation Logic metatheory; Section 5 discusses all of our metatheoretical simplifications in detail; and finally Section 6 discusses related work and concludes the paper.

All proofs in Sections 3 and 4 have been fully mechanized in the Coq proof assistant [7]. The Coq source files, along with their conversions to pdf, can be found at the link to the technical report for this paper [5].

2 Locality and Behavior Preservation

In standard Separation Logic [8, 13, 15, 4], there are two locality properties, known as Safety Monotonicity and the Frame Property, that together imply

soundness of the frame rule. Safety Monotonicity says that any time a program executes safely in a certain state, the same program must also execute safely in any larger state — in other words, unused resources cannot cause a program to crash. The Frame Property says that if a program executes safely on a small state, then any terminating execution of the program on a larger state can be tracked back to some terminating execution on the small state by assuming that the extra added state has no effect and is unchanged. Furthermore, there is a third property, called Termination Monotonicity, that is required whenever we are interested in reasoning about divergence (nontermination). This property says that if a program executes safely and never diverges on a small state, then it cannot diverge on any larger state.

To describe these properties formally, we first formalize the idea of program state. We will describe the theory somewhat informally here; full formal detail will be described later in Section 4. We define states σ to be members of an abstract set Σ. We assume that whenever two states σ_0 and σ_1 are "disjoint," written $\sigma_0 \# \sigma_1$, they can be combined to form the larger state $\sigma_0 \cdot \sigma_1$. Intuitively, two states are disjoint when they occupy disjoint areas of memory.

We represent the semantic meaning of a program C by a binary relation $[\![C]\!]$. We use the common notational convention aRb for a binary relation R to denote $(a, b) \in R$. Intuitively, $\sigma [\![C]\!] \sigma'$ means that, when executing C on initial state σ, it is possible to terminate in state σ'. Note that if σ is related by $[\![C]\!]$ to more than one state, this simply means that C is a nondeterministic program.

We also define two special behaviors `bad` and `div`:

- The notation $\sigma [\![C]\!] \texttt{bad}$ means that C can crash or get stuck when executed on σ, while
- The notation $\sigma [\![C]\!] \texttt{div}$ means that C can diverge (execute forever) when executed on σ.

As a notational convention, we use τ to range over elements of $\Sigma \cup \{\texttt{bad}, \texttt{div}\}$. We require that for any state σ and program C, there is always at least one τ such that $\sigma [\![C]\!] \tau$. In other words, every execution must either crash, go on forever, or terminate in some state.

Now we can define the properties described above more formally. Following are definitions of Safety Monotonicity, the Frame Property, and Termination Monotonicity, respectively:

1.) $\neg \sigma_0 [\![C]\!] \texttt{bad} \wedge \sigma_0 \# \sigma_1 \implies \neg (\sigma_0 \cdot \sigma_1) [\![C]\!] \texttt{bad}$

2.) $\neg \sigma_0 [\![C]\!] \texttt{bad} \wedge (\sigma_0 \cdot \sigma_1) [\![C]\!] \sigma' \implies \exists \sigma_0' \,.\, \sigma' = \sigma_0' \cdot \sigma_1 \wedge \sigma_0 [\![C]\!] \sigma_0'$

3.) $\neg \sigma_0 [\![C]\!] \texttt{bad} \wedge \neg \sigma_0 [\![C]\!] \texttt{div} \wedge \sigma_0 \# \sigma_1 \implies \neg (\sigma_0 \cdot \sigma_1) [\![C]\!] \texttt{div}$

The standard definition of locality was defined in this way because it is the minimum requirement needed to make the frame rule sound — it is as weak as it can possibly be without breaking the logic. It was not defined to correspond with any intuitive notion of locality. As a result, there are two subtleties in the definition that might seem a bit odd. We will now describe these subtleties and

the changes we make to get rid of them. Note that we are not arguing in this section that there is any benefit to changing locality in this way (other than the arguably vacuous benefit of corresponding to our "intuition" of locality) — the benefit will become clear when we discuss how our change simplifies the metatheory in Section 5.

The first subtlety is that Termination Monotonicity only applies in one direction. This means that we could have a program C that runs forever on a state σ, but when we add unused state, we suddenly lose the ability for that infinite execution to occur. We can easily get rid of this subtlety by replacing Termination Monoticity with the following Termination Equivalence property:

$$\neg\sigma_0[\![C]\!]\texttt{bad} \wedge \sigma_0\#\sigma_1 \implies (\sigma_0[\![C]\!]\texttt{div} \iff (\sigma_0 \cdot \sigma_1)[\![C]\!]\texttt{div})$$

The second subtlety is that locality gives us a way of tracking an execution on a large state back to a small one, but it does not allow for the other way around. This means that there can be an execution on a state σ that becomes invalid when we add unused state. This subtlety is a little trickier to remedy than the other. If we think of the Frame Property as really being a "Backwards Frame Property," in the sense that it only works in the direction from large state to small state, then we clearly need to require a corresponding Forwards Frame Property. We would like to say that if C takes σ_0 to σ_0' and we add the unused state σ_1, then C takes $\sigma_0 \cdot \sigma_1$ to $\sigma_0' \cdot \sigma_1$:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0\#\sigma_1 \implies (\sigma_0 \cdot \sigma_1)[\![C]\!](\sigma_0' \cdot \sigma_1)$$

Unfortunately, there is no guarantee that $\sigma_0' \cdot \sigma_1$ is defined, as the states might not occupy disjoint areas of memory. In fact, if C causes our initial state to grow, say by allocating memory, then there will always be some σ_1 that is disjoint from σ_0 but not from σ_0' (e.g., take σ_1 to be exactly that allocated memory). Therefore, it seems as if we are doomed to lose behavior in such a situation upon adding unused state.

There is, however, a solution worth considering: we could disallow programs from ever increasing state. In other words, we can require that whenever C takes σ_0 to σ_0', the area of memory occupied by σ_0' must be a subset of that occupied by σ_0. In this way, anything that is disjoint from σ_0 must also be disjoint from σ_0', so we will not lose any behavior. Formally, we express this property as:

$$\sigma_0[\![C]\!]\sigma_0' \implies (\forall\sigma_1 \,.\, \sigma_0\#\sigma_1 \Rightarrow \sigma_0'\#\sigma_1)$$

We can conveniently combine this property with the previous one to express the Forwards Frame Property as the following condition:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0\#\sigma_1 \implies \sigma_0'\#\sigma_1 \wedge (\sigma_0 \cdot \sigma_1)[\![C]\!](\sigma_0' \cdot \sigma_1)$$

At first glance, it may seem imprudent to impose this requirement, as it apparently disallows memory allocation. However, it is in fact still possible to model memory allocation — we just have to be a little clever about it. Specifically, we can include a set of memory locations in our state that we designate to be the "free list[1]." When memory is allocated, all allocated cells must be

[1] The free list is actually a set rather than a list; we use the term "free list" because it is commonly used in the context of memory allocation.

$$
\begin{array}{rl}
E & ::= E + E' \mid E - E' \mid E \times E' \mid \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid y \mid \ldots \\
B & ::= E = E' \mid \texttt{false} \mid B \Rightarrow B' \\
P, Q & ::= B \mid \texttt{false} \mid \texttt{emp} \mid E \mapsto E' \mid P \Rightarrow Q \mid \forall x.P \mid P * Q \\
C & ::= \texttt{skip} \mid x := E \mid x := [E] \mid [E] := E' \\
 & \quad \mid x := \texttt{cons}(E_1, \ldots, E_n) \mid \texttt{free}(E) \mid C; C' \\
 & \quad \mid \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C' \mid \texttt{while}\, B\, \texttt{do}\, C
\end{array}
$$

Fig. 1. Assertion and Program Syntax

taken from the free list. Contrast this to standard Separation Logic, in which newly-allocated heap cells are taken from outside the state. In the next section, we will show that we can add a free list in this way to the model of Separation Logic without requiring a change to any of the inference rules.

We conclude this section with a brief justification of the term "behavior preservation." Given that C runs safely on a state σ_0, we think of a behavior of C on σ_0 as a particular execution, which can either diverge or terminate at some state σ_0'. The Forwards Frame Property tells us that execution on a larger state $\sigma_0 \cdot \sigma_1$ simulates execution on the smaller state σ_0, while the Backwards (standard) Frame Property says that execution on the smaller state simulates execution on the larger one. Since standard locality only requires simulation in one direction, it is possible for a program to have fewer valid executions, or behaviors, when executing on $\sigma_0 \cdot \sigma_1$ as opposed to just σ_0. Our stronger locality disallows this from happening, enforcing a bisimulation under which all behaviors are preserved when extra resources are added.

3 Impact on a Concrete Separation Logic

We will now present one possible RAM model that enforces our stronger notion of locality without affecting the inference rules of standard Separation Logic. In the standard model of [13], a program state consists of two components: a variable store and a heap. When new memory is allocated, the memory is "magically" added to the heap. As shown in Section 2, we cannot allow allocation to increase the program state in this way. Instead, we will include an explicit free list, or a set of memory locations available for allocation, inside of the program state. Thus a state is now is a triple (s, h, f) consisting of a store, a heap, and a free list, with the heap and free list occupying disjoint areas of memory. Newly-allocated memory will always come from the free list, while deallocated memory goes back into the free list. Since the standard formulation of Separation Logic assumes that memory is infinite and hence that allocation never fails, we similarly require that the free list be infinite. More specifically, we require that there is some location n such that all locations above n are in the free list.

Formally, states are defined as follows:

$$\text{Var } V \triangleq \{x, y, z, \ldots\} \quad \text{Store } S \triangleq V \to \mathbb{Z} \quad \text{Heap } H \triangleq \mathbb{N} \underset{\text{fin}}{\rightharpoonup} \mathbb{Z}$$

$$\text{Free List } F \triangleq \{N \in \mathcal{P}(\mathbb{N}) \mid \exists n \,.\, \forall k \geq n \,.\, k \in N\}$$

$$\text{State } \Sigma \triangleq \{(s, h, f) \in S \times H \times F \mid \text{dom}(h) \cap f = \emptyset\}$$

As a point of clarification, we are not claiming here that including the free list in the state model is a novel idea. Other systems (e.g., [12]) have made use of a very similar idea. The two novel contributions that we will show in this section are: (1) that a state model which includes an explicit free list can provide a behavior-preserving semantics, and (2) that the corresponding program logic can be made to be completely backwards-compatible with standard Separation Logic (meaning that any valid Separation Logic derivation is also a valid derivation in our logic).

We adopt the following standard notations: $\lfloor h \rfloor$ is the domain of the heap h; $s[x \mapsto v]$ is the store which is identical to s, except that the value of variable x is updated to v; $h[l \mapsto v]$ is the heap which is identical to h, except that location l is either added to h with value v if it does not exist in h, or updated with value v if it does exist; $h \backslash l$ is the heap resulting from removing location l from h; $h_0 \# h_1$ is true just when $\lfloor h_0 \rfloor$ and $\lfloor h_1 \rfloor$ do not overlap; $h_0 \cdot h_1$ is equal to the union of h_0 and h_1 if $h_0 \# h_1$, and is undefined otherwise. We also overload the disjointness ($\#$) operator to work with free lists — e.g., $h \# f$ says that $\lfloor h \rfloor$ and $\lfloor f \rfloor$ are disjoint.

Assertion syntax and program syntax are given in Figure 1, and are exactly the same as in the standard model for Separation Logic. This syntax includes expressions E and boolean expressions B, both of which can be evaluated under a given variable store, without any knowledge of the heap. These valuations are denoted by $[\![E]\!]s$ and $[\![B]\!]s$ for a given store s; the former evaluates to an integer, while the latter evaluates to a boolean. These valuations are straightforward and standard in the literature, so we omit their definitions here.

Our satisfaction judgement $(s, h, f) \models P$ for an assertion P is defined by ignoring the free list and only considering whether (s, h) satisfies P. Our definition of $(s, h) \models P$ is identical to that of standard Separation Logic.

For those readers who are not entirely familiar with Separation Logic, as shown in Figure 2, the key assertions to understand are $E \mapsto E'$ and $P * Q$. $E \mapsto E'$ says that the current heap consists *only* of the memory cell at address $[\![E]\!]s$, and that the cell at that address maps to the value $[\![E']\!]s$. $P * Q$ says that we can separate the current heap into two disjoint subheaps h_0 and h_1, with h_0 satisfying P and h_1 satisfying Q. We also define the standard syntactic sugars $E \mapsto E_0, \ldots, E_n$ to be $(E \mapsto E_0) * \ldots * (E + n \mapsto E_n)$, and $E \mapsto -$ to be $\exists x. E \mapsto x$ (where x is not free in E).

Figure 3 defines the small-step operational semantics for our machine. $x := [E]$ and $[E] := E'$ correspond to reading from and writing to the heap, respectively. $x := \texttt{cons}(E_1, \ldots, E_n)$ allocates a nondeterministically-chosen contiguous block

$$
\begin{aligned}
(s,h) \models B &\iff [\![B]\!]s = \texttt{true} \\
(s,h) \models \texttt{false} &\iff \text{never} \\
(s,h) \models E \mapsto E' &\iff \lfloor h \rfloor = \{[\![E]\!]s\} \wedge h([\![E]\!]s) = [\![E']\!]s \\
(s,h) \models P \Rightarrow Q &\iff \text{if } (s,h) \models P, \text{ then } (s,h) \models Q \\
(s,h) \models \forall x.P &\iff \forall v \in \mathbb{Z} \,.\, (s[x \mapsto v], h) \models P \\
(s,h) \models P * Q &\iff \left(\begin{array}{c} \exists h_0, h_1 \,.\, h_0 \# h_1 \wedge h_0 \cdot h_1 = h \wedge \\ (s,h_0) \models P \wedge (s,h_1) \models Q \end{array} \right)
\end{aligned}
$$

Fig. 2. Satisfaction of Assertions

of n heap cells from the free list. The most interesting rules are those for allocation and deallocation, since they make use of the free list. Note that none of the operations make use of any memory existing outside the program state — this is the key for obtaining behavior-preservation.

We define *safety* of a configuration (σ, C) in the standard way, saying that we never get stuck in a non-halting state:

$$
\begin{aligned}
\text{safe}(\sigma, C) \stackrel{\triangle}{=} \forall \sigma', C' \,.\, \sigma, C \stackrel{*}{\longrightarrow} \sigma', C' \wedge C' \neq \texttt{skip} \\
\Longrightarrow \exists \sigma'', C'' \,.\, \sigma', C' \longrightarrow \sigma'', C''
\end{aligned}
$$

In order to formally compare our logic to "standard" Separation Logic, we need to provide the standard version of the small-step operational semantics, denoted as $(s,h), C \rightsquigarrow (s',h'), C'$. We use the notation $(s,h), C \downarrow (s',h')$ to denote big steps. Given this notation, the operational semantics for Separation Logic is nearly identical to ours (with all free lists removed from states, of course). The only difference is in the rule for allocation. For this rule, all we need to do is remove the free lists from the states and change the precondition from $\forall i \in [1,n] \,.\, l + i - 1 \in f$ to $\forall i \in [1,n] \,.\, l + i - 1 \notin h$. It is then possible to show the following relationship between the two operational semantics:

$$\frac{}{\sigma, \texttt{skip}; C \longrightarrow \sigma, C} \text{ (SKIP)}$$

$$\frac{}{(s,h,f), x := E \longrightarrow (s[x \mapsto [\![E]\!]s], h, f), \texttt{skip}} \text{ (ASSGN)}$$

$$\frac{[\![E]\!]s \in \lfloor h \rfloor}{(s,h,f), x := [E] \longrightarrow (s[x \mapsto h([\![E]\!]s)], h, f), \texttt{skip}} \text{ (HEAP-READ)}$$

$$\frac{[\![E]\!]s \in \lfloor h \rfloor}{(s,h,f), [E] := E' \longrightarrow (s, h[[\![E]\!]s \mapsto [\![E']\!]s], f), \texttt{skip}} \text{ (HEAP-WRITE)}$$

$$\frac{\forall i \in [1,n] \,.\, l + i - 1 \in f}{\begin{array}{c}(s,h,f), x := \texttt{cons}(E_1, \ldots, E_n) \longrightarrow \\ (s[x \mapsto l], h[l \mapsto [\![E_1]\!]s] \ldots [l+n-1 \mapsto [\![E_n]\!]s], f - \{l, \ldots, l+n-1\}), \texttt{skip}\end{array}} \text{ (CONS)}$$

$$\frac{[\![E]\!]s \in \lfloor h \rfloor}{(s,h,f), \texttt{free}(E) \longrightarrow (s, h \backslash [\![E]\!]s, f \cup \{[\![E]\!]s\}), \texttt{skip}} \text{ (FREE)}$$

$$\frac{\sigma, C \longrightarrow \sigma', C'}{\sigma, C; C'' \longrightarrow \sigma', C'; C''} \text{ (SEQ)} \qquad \frac{[\![B]\!]s = \texttt{true}}{\sigma, \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2 \longrightarrow \sigma, C_1} \text{ (IF-TRUE)}$$

$$\frac{[\![B]\!]s = \texttt{false}}{\sigma, \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2 \longrightarrow \sigma, C_2} \text{ (IF-FALSE)}$$

$$\frac{[\![B]\!]s = \texttt{true}}{\sigma, \texttt{while}\, B\, \texttt{do}\, C \longrightarrow \sigma, C; \texttt{while}\, B\, \texttt{do}\, C} \text{ (WHILE-TRUE)}$$

$$\frac{[\![B]\!]s = \texttt{false}}{\sigma, \texttt{while}\, B\, \texttt{do}\, C \longrightarrow \sigma, \texttt{skip}} \text{ (WHILE-FALSE)} \qquad \frac{}{\sigma, C \xrightarrow{0} \sigma, C} \text{ (STEPN-ZERO)}$$

$$\frac{\sigma, C \longrightarrow \sigma', C' \qquad \sigma', C' \xrightarrow{n} \sigma'', C''}{\sigma, C \xrightarrow{n+1} \sigma'', C''} \text{ (STEPN-SUCC)}$$

$$\frac{\sigma, C \xrightarrow{n} \sigma', C'}{\sigma, C \xrightarrow{*} \sigma', C'} \text{ (MULTI-STEP)} \qquad \frac{\sigma, C \xrightarrow{*} \sigma', \texttt{skip}}{\sigma, C \Downarrow \sigma'} \text{ (BIG-STEP)}$$

Fig. 3. Small-Step Operational Semantics

Lemma 1.

$$(s,h), C \overset{n}{\rightsquigarrow} (s',h'), C' \iff \exists f, f' \,.\, (s,h,f), C \xrightarrow{n} (s',h',f'), C'$$

Proof. The backwards direction is a straightforward proof by induction. For the forwards direction, we actually prove a stronger statement by picking our f and f' to be exactly $\mathbb{N} - \lfloor h \rfloor$ and $\mathbb{N} - \lfloor h' \rfloor$, respectively. The proof of this stronger statement is then straightforward by induction. Picking the free lists in this way showcases how the Separation Logic model can be interpreted as having an implicit free list containing everything not in the heap.

For more details on this proof, see our Coq implementation.

The inference rules in the form $\vdash \{P\}\, C\, \{Q\}$ for our logic are exactly the same as those used in standard Separation Logic. We give most of these inference rules in Figure 4. The reader may refer to [13] for more inference rules.

We say that a triple $\models \{P\}\, C\, \{Q\}$ is *semantically valid* when, for all σ, σ':

1.) if $\sigma \models P$, then $\text{safe}(\sigma, C)$
2.) if $\sigma \models P$ and $\sigma, C \Downarrow \sigma'$, then $\sigma' \models Q$

Semantic validity of standard Separation Logic triples is defined in the same way, but using the operational semantics for Separation Logic. We will write this as $\models_{SL} \{P\}\, C\, \{Q\}$. Note that we are only considering a *partial correctness* definition of validity here, meaning that programs are not required to terminate.

We are now in a position to prove soundness and completeness of our logic. We first prove a minor technical lemma:

Lemma 2.

$$(s,h), C \rightsquigarrow (s',h'), C' \Longrightarrow \forall f \,.\, (f \# h \Rightarrow \exists \sigma \,.\, (s,h,f), C \longrightarrow \sigma, C')$$

Proof. Straightforward by induction on the rules for stepping. See the Coq implementation for more details.

Theorem 1 (Soundness and Completeness).

$$\vdash \{P\}\, C\, \{Q\} \iff \models \{P\}\, C\, \{Q\}$$

Proof. Note that $\vdash \{P\}\, C\, \{Q\}$ has the same definition in both our logic and in Separation Logic, since we use the same assertion language and inference rules. Therefore, because Separation Logic is known to be sound and complete, we have that $\vdash \{P\}\, C\, \{Q\} \iff \models_{SL} \{P\}\, C\, \{Q\}$. We thus need only show that $\models_{SL} \{P\}\, C\, \{Q\} \iff \models \{P\}\, C\, \{Q\}$.

First, suppose that $\models_{SL} \{P\}\, C\, \{Q\}$. To prove the first property of semantic validity, suppose that $(s,h,f) \models P$, and consider some execution $(s,h,f), C \xrightarrow{*} (s',h',f'), C'$ with $C' \neq \texttt{skip}$. Then we need to show that $(s',h',f'), C'$ can take

$$\frac{}{\vdash \{\mathtt{emp}\}\,\mathtt{skip}\,\{\mathtt{emp}\}}\ (\text{SKIP}) \qquad \frac{}{\vdash \{x = y \wedge \mathtt{emp}\}\,x := E\,\{x = E[y/x] \wedge \mathtt{emp}\}}\ (\text{ASSGN})$$

$$\frac{}{\vdash \{x = y \wedge E \mapsto z\}\,x := [E]\,\{x = z \wedge E[y/x] \mapsto z\}}\ (\text{HEAP-READ})$$

$$\frac{}{\vdash \{E \mapsto -\}\,[E] := E'\,\{E \mapsto E'\}}\ (\text{HEAP-WRITE})$$

$$\frac{}{\vdash \{x = y \wedge \mathtt{emp}\}\,x := \mathtt{cons}(E_1, \ldots, E_k)\,\{x \mapsto E_1[y/x], \ldots, E_k[y/x]\}}\ (\text{CONS})$$

$$\frac{}{\vdash \{E \mapsto -\}\,\mathtt{free}(E)\,\{\mathtt{emp}\}}\ (\text{FREE}) \qquad \frac{\vdash \{P\}\,C_1\,\{Q\} \qquad \vdash \{Q\}\,C_2\,\{R\}}{\vdash \{P\}\,C_1; C_2\,\{R\}}\ (\text{SEQ})$$

$$\frac{\vdash \{B \wedge P\}\,C_1\,\{Q\} \qquad \vdash \{\neg B \wedge P\}\,C_2\,\{Q\}}{\vdash \{P\}\,\mathtt{if}\,B\,\mathtt{then}\,C_1\,\mathtt{else}\,C_2\,\{Q\}}\ (\text{IF})$$

$$\frac{\vdash \{B \wedge P\}\,C\,\{P\}}{\vdash \{P\}\,\mathtt{while}\,B\,\mathtt{do}\,C\,\{\neg B \wedge P\}}\ (\text{WHILE})$$

$$\frac{P' \Rightarrow P \qquad Q \Rightarrow Q' \qquad \vdash \{P\}\,C\,\{Q\}}{\vdash \{P'\}\,C\,\{Q'\}}\ (\text{CONSEQ})$$

$$\frac{\vdash \{P_1\}\,C\,\{Q_1\} \qquad \vdash \{P_2\}\,C\,\{Q_2\}}{\vdash \{P_1 \wedge P_2\}\,C\,\{Q_1 \wedge Q_2\}}\ (\text{CONJ})$$

$$\frac{\vdash \{P_1\}\,C\,\{Q_1\} \qquad \vdash \{P_2\}\,C\,\{Q_2\}}{\vdash \{P_1 \vee P_2\}\,C\,\{Q_1 \vee Q_2\}}\ (\text{DISJ})$$

$$\frac{\vdash \{P\}\,C\,\{Q\} \qquad \mathtt{modifies}(C) \cap \mathtt{vars}(R) = \emptyset}{\vdash \{P * R\}\,C\,\{Q * R\}}\ (\text{FRAME})$$

Fig. 4. Some Separation Logic Inference Rules

another step. By Lemma 1, we have that $(s,h), C \overset{*}{\rightsquigarrow} (s',h'), C'$. Since $(s,h) \models P$, we know that safe$((s,h),C)$, and so $(s',h'), C' \rightsquigarrow (s'',h''), C''$ for some s'', h'', C''. Therefore Lemma 2 tells us that $(s',h',f'), C'$ can indeed take a step. For the second property, suppose that $(s,h,f) \models P$ and $(s,h,f), C \longrightarrow (s',h',f'), \texttt{skip}$. Then Lemma 1 tells us that $(s,h), C \overset{*}{\rightsquigarrow} (s',h'), \texttt{skip}$, meaning that $(s',h') \models Q$, and so $(s',h',f') \models Q$.

Now suppose that $\models \{P\}\, C\, \{Q\}$. For the first property, suppose that $(s,h) \models P$ and $(s,h), C \overset{*}{\rightsquigarrow} (s',h'), C'$ with $C' \neq \texttt{skip}$. Lemma 1 gives us $(s,h,f), C \overset{*}{\longrightarrow} (s',h',f'), C'$ for some f and f', which means that $(s',h',f'), C' \longrightarrow (s'',h'',f''), C''$ for some s'', h'', f'', C'' (since $(s,h,f) \models P$). Therefore Lemma 1 gives us $(s',h'), C' \rightsquigarrow (s'',h''), C''$, as desired. For the second property, suppose $(s,h) \models P$ and $(s,h), C \overset{*}{\rightsquigarrow} (s',h'), \texttt{skip}$. By Lemma 1, we have

$$(s,h,f), C \overset{*}{\rightsquigarrow} (s',h',f'), \texttt{skip}$$

for some f and f'. Since $(s,h,f) \models P$, this means that $(s',h',f') \models Q$, and so $(s',h') \models Q$.

We have thus shown that our new model does not cause any complications in the usage of Separation Logic. Any specification that can be proved using the standard model can also be proved using our model. We now only need to show that our model enjoys the stronger, behavior-preserving notion of locality. As described in Section 2, this locality is composed of Safety Monotonicity, Termination Equivalence, and the Forward and Backwards Frame Properties. We first prove that the two frame properties hold:

Theorem 2 (Frame Properties).

$$\begin{aligned}
&\textit{1.)}\ (s,h_0,f), C \xrightarrow{n} (s',h_0',f'), C' \wedge h_0 \# h_1 \wedge f \# h_1 \implies \\
&\qquad h_0' \# h_1 \wedge (s, h_0 \cdot h_1, f), C \xrightarrow{n} (s', h_0' \cdot h_1, f'), C' \\
&\textit{2.)}\ \text{safe}((s,h_0,f),C) \wedge (s,h_0 \cdot h_1, f), C \xrightarrow{n} (s',h',f'), C' \implies \\
&\qquad \exists h_0' \,.\, h' = h_0' \cdot h_1 \wedge (s,h_0,f), C \xrightarrow{n} (s',h_0',f'), C'
\end{aligned}$$

Proof. Straightforward by induction on the derivation rules for stepping. For details, see the Coq implementation.

It is easy to show that these Frame Properties imply both Safety Monotonicity and Termination Equivalence.

Lemma 3 (Safety Monotonicity).

$$\text{safe}((s, h_0, f), C) \land h_0 \# h_1 \land f \# h_1 \implies \text{safe}((s, h_0 \cdot h_1, f), C)$$

Proof. Suppose that $\text{safe}((s, h_0, f), C)$, and consider an execution on the large state $(s, h_0 \cdot h_1, f), C \xrightarrow{n} (s', h', f'), C'$ with $C' \neq \texttt{skip}$. Then the Backwards Frame Property tells us that $h' = h'_0 \cdot h_1$ and $(s, h_0, f), C \xrightarrow{n} (s', h'_0, f'), C'$. Since $\text{safe}((s, h_0, f), C)$ and $C' \neq \texttt{skip}$, we see that $(s', h'_0, f'), C' \longrightarrow (s'', h''_0, f''), C''$ for some s'', h''_0, f'', C''. Thus we can now use the Forwards Frame Property (clearly $h_1 \# f'$ since $(s', h'_0 \cdot h_1, f')$ is a well-typed state) to obtain $(s', h'_0 \cdot h_1, f'), C' \longrightarrow (s'', h''_0 \cdot h_1, f''), C''$, and so $\text{safe}((s, h_0 \cdot h_1, f), C)$ does indeed hold.

In order to define Termination Equivalence, we first need to define divergence. We say that σ diverges on C, written $\sigma, C \uparrow$, if there exists an infinite path of steps starting from σ, C. More formally:

$$\sigma, C \uparrow \triangleq \forall n \,.\, \exists \sigma', C' \,.\, \sigma, C \xrightarrow{n} \sigma', C'$$

Lemma 4 (Termination Equivalence).

$$\text{safe}((s, h_0, f), C) \land h_0 \# h_1 \land f \# h_1 \implies (s, h_0, f), C \uparrow \iff (s, h_0 \cdot h_1, f), C \uparrow$$

Proof. First, suppose $(s, h_0, f), C \uparrow$, and pick any n. Then there are some s', h'_0, f', C' such that $(s, h_0, f), C \xrightarrow{n} (s', h'_0, f'), C'$. Thus the Forwards Frame Property tells us that $h'_0 \# h_1$ and $(s, h_0 \cdot h_1, f), C \xrightarrow{n} (s', h'_0 \cdot h_1, f'), C'$, as desired. For the other direction, suppose $(s, h_0 \cdot h_1, f), C$ and pick any n. Then $(s, h_0 \cdot h_1, f), C \xrightarrow{n} (s', h', f'), C'$ for some s', h', f', C'. Since $\text{safe}((s, h_0, f), C)$, the Backwards Frame Property tells us that $h' = h'_0 \cdot h_1$ for some h'_0, and $(s, h_0, f), C \xrightarrow{n} (s', h'_0, f'), C'$, as desired.

We conclude this section with a quick note on reasoning about the free list. We presented our logic with the purpose of showing that, at the level of inference rules and derivations, it works exactly the same as standard Separation Logic. However, at the level of the underlying model, we now have this free list within the state. Therefore, if we so desire, we could define additional assertions and inference rules allowing for more precise reasoning involving the free list. One idea might be to have a separate, free list section of assertions in which we write, for example, $E * \texttt{true}$ to claim that E is a part of the free list. Then the axiom for `free` would look something like:

$$\{E \mapsto -; \texttt{true}\}\, \texttt{free}(E)\, \{\texttt{emp}; E * \texttt{true}\}$$

4 The Abstract Logic

In order to clearly explain how our stronger notion of locality resolves the metatheoretical issues described in Section 1, we will first formally describe how

our locality fits into a context similar to that of Abstract Separation Logic [4]. With a minor amount of work, the logic of Section 3 can be molded into a particular instance of the abstract logic presented here.

We define a *separation algebra* to be a set of states Σ, along with a partial associative and commutative operator $\cdot : \Sigma \to \Sigma \rightharpoonup \Sigma$. The disjointness relation $\sigma_0 \# \sigma_1$ holds iff $\sigma_0 \cdot \sigma_1$ is defined, and the substate relation $\sigma_0 \preceq \sigma_1$ holds iff there is some σ_0' such that $\sigma_0 \cdot \sigma_0' = \sigma_1$. A particular element of Σ is designated as a unit state, denoted u, with the property that for any σ, $\sigma \# u$ and $\sigma \cdot u = \sigma$. We require the $\cdot$ operator to be cancellative, meaning that $\sigma \cdot \sigma_0 = \sigma \cdot \sigma_1 \Rightarrow \sigma_0 = \sigma_1$.

Our concrete model can be represented as a separation algebra by defining $(s_0, h_0, f_0) \cdot (s_1, h_1, f_1)$ to be $(s_0, h_0 \cdot h_1, f_0)$ if $s_0 = s_1$, $f_0 = f_1$, and $h_0 \# h_1$; otherwise, it is undefined. Associativity, commutativity, and cancellativity are simple to verify. We can create a special state, denoted by (), to be the unit state — thus $(s, h, f) \cdot () = () \cdot (s, h, f) = (s, h, f)$.

An *action* is a set of pairs of type $\Sigma \cup \{\texttt{bad}, \texttt{div}\} \times \Sigma \cup \{\texttt{bad}, \texttt{div}\}$. We require the following two properties: (1) actions always relate `bad` to `bad` and `div` to `div`, and never relate `bad` or `div` to anything else; and (2) actions are total, in the sense that for any τ, there exists some τ' such that $\tau A \tau'$ (recall from Section 2 that we use τ to range over elements of $\Sigma \cup \{\texttt{bad}, \texttt{div}\}$). Note that these two requirements are preserved over the standard composition of relations, as well as over both finitary and infinite unions. We write `Id` to represent the identity action $\{(\tau, \tau) \mid \tau \in \Sigma \cup \{\texttt{bad}, \texttt{div}\}\}$.

Note that it is more standard in the literature to have the domain of actions range only over Σ — we use $\Sigma \cup \{\texttt{bad}, \texttt{div}\}$ here because it has the pleasant effect of making $[\![C_1; C_2]\!]$ correspond precisely to standard composition. Intuitively, once an execution goes wrong, it continues to go wrong, and once an execution diverges, it continues to diverge.

A *local action* is an action A that satisfies the following four properties, which respectively correspond to Safety Monotonicity, Termination Equivalence, the Forwards Frame Property, and the Backwards Frame Property from Section 2:

$$\text{1.)}\ \neg \sigma_0 A \texttt{bad} \wedge \sigma_0 \# \sigma_1 \Longrightarrow \neg (\sigma_0 \cdot \sigma_1) A \texttt{bad}$$
$$\text{2.)}\ \neg \sigma_0 A \texttt{bad} \wedge \sigma_0 \# \sigma_1 \Longrightarrow (\sigma_0 A \texttt{div} \iff (\sigma_0 \cdot \sigma_1) A \texttt{div})$$
$$\text{3.)}\ \sigma_0 A \sigma_0' \wedge \sigma_0 \# \sigma_1 \Longrightarrow \sigma_0' \# \sigma_1 \wedge (\sigma_0 \cdot \sigma_1) A (\sigma_0' \cdot \sigma_1)$$
$$\text{4.)}\ \neg \sigma_0 A \texttt{bad} \wedge (\sigma_0 \cdot \sigma_1) A \sigma' \Longrightarrow \exists \sigma_0' \,.\, \sigma' = \sigma_0' \cdot \sigma_1 \wedge \sigma_0 A \sigma_0'$$

We denote the set of all local actions by **LocAct**. We now show that the set of local actions is closed under composition and (possibly infinite) union. We use the notation $A_1; A_2$ to denote composition, and $\bigcup \mathcal{A}$ to denote union (where $\mathcal{A}$ is a possibly infinite set of actions). The formal definitions of these operations follow. Note that we require that $\mathcal{A}$ be non-empty. This is necessary because $\bigcup \emptyset$ is $\emptyset$, which is not a valid action. Unless otherwise stated, whenever we write

$\bigcup \mathcal{A}$, there will always be an implicit assumption that $\mathcal{A} \neq \emptyset$.

$$\tau A_1; A_2 \tau' \iff \exists \tau'' \,.\, \tau A_1 \tau'' \wedge \tau'' A_2 \tau'$$
$$\tau \bigcup \mathcal{A} \tau' \iff \exists A \in \mathcal{A} \,.\, \tau A \tau' \qquad (\mathcal{A} \neq \emptyset)$$

Lemma 5. *If A_1 and A_2 are local actions, then $A_1; A_2$ is a local action.*

Proof. It will be useful to first note that $\sigma A_1; A_2 \mathtt{bad}$ iff either $\sigma A_1 \mathtt{bad}$ or there exists some σ' such that $\sigma A_1 \sigma'$ and $\sigma' A_2 \mathtt{bad}$. This is due to the fact that we know $\mathtt{bad} A_2 \mathtt{bad}$ and $\neg \mathtt{div} A_2 \mathtt{bad}$. Similarly, it also the case that $\sigma A_1; A_2 \mathtt{div}$ iff either $\sigma A_1 \mathtt{div}$ or there exists some σ' such that $\sigma A_1 \sigma'$ and $\sigma' A_2 \mathtt{div}$.

For Safety Monotonicity, suppose that $\sigma_0 \# \sigma_1$ and $\neg \sigma_0 A_1; A_2 \mathtt{bad}$. Suppose by way of contradiction that $(\sigma_0 \cdot \sigma_1) A_1; A_2 \mathtt{bad}$. Since $\neg \sigma_0 A_1; A_2 \mathtt{bad}$ and $\mathtt{bad} A_2 \mathtt{bad}$, we have $\neg \sigma_0 A_1 \mathtt{bad}$. Thus by Safety Monotonicity of A_1, $\neg(\sigma_0 \cdot \sigma_1) A_1 \mathtt{bad}$. By our note above, we see that there must be some σ such that $(\sigma_0 \cdot \sigma_1) A_1 \sigma$ and $\sigma A_2 \mathtt{bad}$. By the Backwards Frame Property of A_1, there must be a σ_0' such that $\sigma = \sigma_0' \cdot \sigma_1$ and $\sigma_0 A_1 \sigma_0'$. Thus we have that $(\sigma_0' \cdot \sigma_1) A_2 \mathtt{bad}$, and so Safety Monotonicity of A_2 tells us that $\sigma_0' A_2 \mathtt{bad}$. Hence $\sigma_0 A_1; A_2 \mathtt{bad}$, which is a contradiction.

For Termination Equivalence, suppose that $\sigma_0 \# \sigma_1$ and $\neg \sigma_0 A_1; A_2 \mathtt{bad}$. Then we also have $\neg \sigma_0 A_1 \mathtt{bad}$, since we have $\mathtt{bad} A_2 \mathtt{bad}$.

For the forward direction, suppose that $\sigma_0 A_1; A_2 \mathtt{div}$. By the note above, there are two possible situations. In the first situation, we have $\sigma_0 A_1 \mathtt{div}$. By Termination Equivalence of A_1, this implies that $(\sigma_0 \cdot \sigma_1) A_1 \mathtt{div}$, and so $(\sigma_0 \cdot \sigma_1) A_1; A_2 \mathtt{div}$, as desired. In the second situation, there is a state σ such that $\sigma_0 A_1 \sigma$ and $\sigma A_2 \mathtt{div}$. By the Forwards Frame Property of A_1, we see that $\sigma \# \sigma_1$ and $(\sigma_0 \cdot \sigma_1) A_1 (\sigma \cdot \sigma_1)$. Now note that we must have $\neg \sigma A_2 \mathtt{bad}$, because otherwise we would be able to derive $\sigma_0 A_1; A_2 \mathtt{bad}$, which is a contradiction. Therefore, by Termination Equivalence of A_2, we have $(\sigma \cdot \sigma_1) A_2 \mathtt{div}$. Hence we get $(\sigma_0 \cdot \sigma_1) A_1; A_2 \mathtt{div}$, as desired.

For the backward direction, suppose that $(\sigma_0 \cdot \sigma_1) A_1; A_2 \mathtt{div}$. Again by the note above, there are two possible situations. In the first situation, we have $(\sigma_0 \cdot \sigma_1) A_1 \mathtt{div}$. By Termination Equivalence of A_1, this implies that $\sigma_0 A_1 \mathtt{div}$, and so $\sigma_0 A_1; A_2 \mathtt{div}$, as desired. In the second situation, there is a state σ such that $(\sigma_0 \cdot \sigma_1) A_1 \sigma$ and $\sigma A_2 \mathtt{div}$. By the Backwards Frame Property of A_1, there must be a σ_0' such that $\sigma = \sigma_0' \cdot \sigma_1$ and $\sigma_0 A_1 \sigma_0'$. Now note that we must have $\neg \sigma_0' A_2 \mathtt{bad}$, because otherwise we would be able to derive $\sigma_0 A_1; A_2 \mathtt{bad}$, which is a contradiction. Therefore, by Termination Equivalence of A_2, we have $\sigma_0' A_2 \mathtt{div}$. Hence we get $\sigma_0 A_1; A_2 \mathtt{div}$, as desired.

For the Forwards Frame Property, suppose that $\sigma_0 \# \sigma_1$ and $\sigma_0 A_1; A_2 \sigma_0'$. Then there exists a τ such that $\sigma_0 A_1 \tau$ and $\tau A_2 \sigma_0'$. Furthermore, τ cannot be $\mathtt{bad}$ or $\mathtt{div}$ since $\tau A_2 \sigma_0'$ — thus let τ be σ_0''. By the Forwards Frame Property of A_1, we have $\sigma_0'' \# \sigma_1$ and $(\sigma_0 \cdot \sigma_1) A_1 (\sigma_0'' \cdot \sigma_1)$. Therefore, by the Forwards Frame Property of A_2, we have $\sigma_0' \# \sigma_1$ and $(\sigma_0'' \cdot \sigma_1) A_2 (\sigma_0' \cdot \sigma_1)$. Hence $\sigma_0' \# \sigma_1$ and $(\sigma_0 \cdot \sigma_1) A_1; A_2 (\sigma_0' \cdot \sigma_1)$, as desired.

$$C \ ::= c \mid C_1; C_2 \mid C_1 + C_2 \mid C^*$$

$$\forall c \,.\, [\![c]\!] \in \mathbf{LocAct} \qquad [\![C_1; C_2]\!] \triangleq [\![C_1]\!]; [\![C_2]\!]$$

$$[\![C_1 + C_2]\!] \triangleq [\![C_1]\!] \cup [\![C_2]\!] \qquad [\![C^*]\!] \triangleq \bigcup_{n \in \mathbb{N}} [\![C]\!]^n$$

$$[\![C]\!]^0 \triangleq \mathtt{Id} \qquad [\![C]\!]^{n+1} \triangleq [\![C]\!]; [\![C]\!]^n$$

Fig. 5. Command Definition and Denotational Semantics

For the Backwards Frame Property, suppose that $\neg\sigma_0 A_1; A_2\mathtt{bad}$ and $(\sigma_0 \cdot \sigma_1)A_1; A_2\sigma'$. Then, repeating some reasoning from earlier in this proof, we have $\neg\sigma_0 A_1\mathtt{bad}$, and there exists a σ such that $(\sigma_0 \cdot \sigma_1)A_1\sigma$ and $\sigma A_2\sigma'$. By the Backwards Frame Property of A_1, we get $\sigma = \sigma_0' \cdot \sigma_1$ and $\sigma_0 A_1\sigma_0'$. Now note that $\neg\sigma_0' A_2\mathtt{bad}$, because otherwise we would be able to derive $\sigma_0 A_1; A_2\mathtt{bad}$, which is a contradiction. Therefore, by the Backwards Frame Property of A_2, we get $\sigma' = \sigma_0'' \cdot \sigma_1$ and $\sigma_0' A_2\sigma_0''$. Hence $\sigma' = \sigma_0'' \cdot \sigma_1$ and $\sigma_0 A_1; A_2\sigma_0''$, as desired.

Lemma 6. *If every A in the set $\mathcal{A}$ is a local action, then $\bigcup\mathcal{A}$ is a local action.*

Proof. For Safety Monotonicity, suppose $\sigma_0\#\sigma_1$ and $\neg\sigma_0\bigcup\mathcal{A}\mathtt{bad}$. Suppose by way of contradiction that $(\sigma_0 \cdot \sigma_1)\bigcup\mathcal{A}\mathtt{bad}$. Then there is some $A \in \mathcal{A}$ such that $(\sigma_0 \cdot \sigma_1)A\mathtt{bad}$. By Safety Monotonicity of A, we get $\sigma_0 A\mathtt{bad}$. But this means that $\sigma_0\bigcup\mathcal{A}\mathtt{bad}$, which is a contradiction.

For Termination Equivalence, suppose that $\sigma_0\#\sigma_1$ and $\neg\sigma_0\bigcup\mathcal{A}\mathtt{bad}$. This means that for every $A \in \mathcal{A}$, $\neg\sigma_0 A\mathtt{bad}$. For the forward direction, suppose that $\sigma_0\bigcup\mathcal{A}\mathtt{div}$. Then $\sigma_0 A\mathtt{div}$ for some $A \in \mathcal{A}$. Thus Termination Equivalence of A gives us $(\sigma_0 \cdot \sigma_1)A\mathtt{div}$, and so we get the desired $(\sigma_0 \cdot \sigma_1)\bigcup\mathcal{A}\mathtt{div}$. For the backward direction, suppose that $(\sigma_0 \cdot \sigma_1)\bigcup\mathcal{A}\mathtt{div}$. Then $(\sigma_0 \cdot \sigma_1)A\mathtt{div}$ for some $A \in \mathcal{A}$. Thus Termination Equivalence of A gives us $\sigma_0 A\mathtt{div}$, and so we get the desired $\sigma_0\bigcup\mathcal{A}\mathtt{div}$.

For the Forwards Frame Property, suppose that $\sigma_0\#\sigma_1$ and $\sigma_0\bigcup\mathcal{A}\sigma_0'$. Then $\sigma_0 A\sigma_0'$ for some $A \in \mathcal{A}$, and so by the Forwards Frame Property of A, we have $\sigma_0'\#\sigma_1$ and $(\sigma_0 \cdot \sigma_1)A(\sigma_0' \cdot \sigma_1)$, which in turn implies the desired result.

For the Backwards Frame Property, suppose that $\neg\sigma_0\bigcup\mathcal{A}\mathtt{bad}$ and $(\sigma_0 \cdot \sigma_1)\bigcup\mathcal{A}\sigma'$. Then $(\sigma_0 \cdot \sigma_1)A\sigma'$ for some $A \in \mathcal{A}$, and for all $A \in \mathcal{A}$ we have $\neg\sigma_0 A\mathtt{bad}$. Hence the Backwards Frame Property of A tells us that $\sigma' = \sigma_0' \cdot \sigma_1$ and $\sigma_0 A\sigma_0'$, which implies the desired result.

Figure 5 defines our abstract program syntax and semantics. The language consists of primitive commands, sequencing $(C_1; C_2)$, nondeterministic choice $(C_1 + C_2)$, and finite iteration (C^*). The semantics of primitive commands are abstracted — the only requirement is that they are local actions. Therefore, from

$$\frac{\neg\sigma[\![c]\!]\texttt{bad}}{\vdash \{\{\sigma\}\}\, c\, \{\{\sigma' \mid \sigma[\![c]\!]\sigma'\}\}}\ (\text{PRIM}) \qquad \frac{\vdash \{P\}\, C_1\, \{Q\} \qquad \vdash \{Q\}\, C_2\, \{R\}}{\vdash \{P\}\, C_1; C_2\, \{R\}}\ (\text{SEQ})$$

$$\frac{\vdash \{P\}\, C_1\, \{Q\} \qquad \vdash \{P\}\, C_2\, \{Q\}}{\vdash \{P\}\, C_1 + C_2\, \{Q\}}\ (\text{PLUS}) \qquad \frac{\vdash \{P\}\, C\, \{P\}}{\vdash \{P\}\, C^*\, \{P\}}\ (\text{STAR})$$

$$\frac{\vdash \{P\}\, C\, \{Q\}}{\vdash \{P * R\}\, C\, \{Q * R\}}\ (\text{FRAME}) \qquad \frac{P' \subseteq P \qquad \vdash \{P\}\, C\, \{Q\} \qquad Q \subseteq Q'}{\vdash \{P'\}\, C\, \{Q'\}}\ (\text{CONSEQ})$$

$$\frac{\forall i \in I\ .\ \vdash \{P_i\}\, C\, \{Q_i\}}{\vdash \{\bigcup P_i\}\, C\, \{\bigcup Q_i\}}\ (\text{DISJ}) \qquad \frac{\forall i \in I\ .\ \vdash \{P_i\}\, C\, \{Q_i\} \qquad I \neq \emptyset}{\vdash \{\bigcap P_i\}\, C\, \{\bigcap Q_i\}}\ (\text{CONJ})$$

Fig. 6. Inference Rules

the two previous lemmas and the trivial fact that `Id` is a local action, it is clear that the semantics of *every* program is a local action.

Note that in our concrete language used if statements and while loops. As shownin [4], it is possible to represent if and while constructs with finite iteration and nondeterministic choice by including a primitive command $\texttt{assume}(B)$, which does nothing if the boolean expression B is true, and diverges otherwise. Given this setup, we can define the primitive command $\texttt{assume}(B)$ as follows:

$$\begin{aligned}[\![\texttt{assume}(B)]\!] \triangleq\ & \{(\texttt{bad}, \texttt{bad}), (\texttt{div}, \texttt{div})\} \cup \\ & \{(\sigma, \sigma) \mid [\![B]\!]\sigma = \texttt{true}\} \cup \{(\sigma, \texttt{div}) \mid [\![B]\!]\sigma = \texttt{false}\} \cup \\ & \{(\sigma, \texttt{bad}) \mid [\![B]\!]\sigma \text{ undefined}\}\end{aligned}$$

It is a simple matter to show that this is a local action. We can then syntactically define if and while statements as follows:

$$\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \triangleq (\texttt{assume}(B); C_1) + (\texttt{assume}(\neg B); C_2)$$

$$\texttt{while}\ B\ \texttt{do}\ C \triangleq (\texttt{assume}(B); C)^*; \texttt{assume}(\neg B)$$

Technically, these definitions only correctly implement if and while statements in terms of which states they can terminate at — they do not correctly implement divergence behavior since they allow for arbitrary divergence. Thus these definitions should only be used if we do not care about divergence behavior. It is certainly still possible to define fully correct if and while statements, but describing the technical details would venture too far beyond the scope of this paper.

Now that we have defined the interpretation of programs as local actions, we can talk about the meaning of a triple $\{P\}\, C\, \{Q\}$. We define an assertion P to be a set of states, and we say that a state σ satisfies P iff $\sigma \in P$. We can then

define the separating conjunction as follows:

$$P * Q \triangleq \{\sigma \in \Sigma \mid \exists \sigma_0 \in P, \sigma_1 \in Q \,.\, \sigma = \sigma_0 \cdot \sigma_1\}$$

Given an assignment of primitive commands to local actions, we say that a triple is valid, written $\models \{P\}\, C\, \{Q\}$, just when the following two properties hold for all states σ and σ':

$$\text{1.)}\ \sigma \in P \implies \neg \sigma [\![C]\!] \mathtt{bad}$$
$$\text{2.)}\ \sigma \in P \wedge \sigma [\![C]\!] \sigma' \implies \sigma' \in Q$$

The inference rules of the logic are given in Figure 6. Note that we are taking a significant presentation shortcut here in the inference rule for primitive commands. Specifically, we assume that we know the exact local action $[\![c]\!]$ of each primitive command c. This assumption makes sense when we define our own primitive commands, as we do in the logic of Section 3. However, in a more general setting, we might be provided with an opaque function along with a specification (precondition and postcondition) for the function. Since the function is opaque, we must consider it to be a primitive command in the abstract setting. Yet we do not know how it is implemented, so we do not know its precise local action. In [4], the authors provide a method for inferring a "best" local action from the function's specification. With a decent amount of technical development, we can do something similar here, using our stronger definition of locality.

Given this assumption, we prove soundness and completeness of our abstract logic. The details of the proof can be found in our Coq implementation [5].

Theorem 3 (Soundness and Completeness).

$$\vdash \{P\}\, C\, \{Q\} \iff \models \{P\}\, C\, \{Q\}$$

5 Simplifying Separation Logic Metatheory

Now that we have an abstracted formalism of our behavior-preserving local actions, we will resolve each of the four metatheoretical issues described in Sec 1.

5.1 Footprints and Smallest Safe States

Consider a situation in which we are handed a program C along with a specification of what this program does. The specification consists of a set of axioms; each axiom has the form $\{P\}\, C\, \{Q\}$ for some precondition P and postcondition Q. A common question to ask would be: is this specification *complete*? In other words, if the triple $\models \{P\}\, C\, \{Q\}$ is valid for some P and Q, then is it possible to derive $\vdash \{P\}\, C\, \{Q\}$ from the provided specification?

In standard Separation Logic, it can be extremely difficult to answer this question. In [12], the authors conduct an in-depth study of various conditions

and circumstances under which it is possible to prove that certain specifications are complete. However, in the general case, there is no easy way to prove this.

We can show that under our assumption of behavior preservation, there is a very easy way to guarantee that a specification is complete. In particular, a specification that describes the exact behavior of C on all of its *smallest safe states* is always complete. Formally, a smallest safe state is a state σ such that $\neg\sigma[\![C]\!]\texttt{bad}$ and, for all $\sigma' \prec \sigma$, $\sigma'[\![C]\!]\texttt{bad}$.

To see that such a specification may not be complete in standard Separation Logic, we borrow an example from [12]. Consider the program C, defined as $x := \texttt{cons}(0); \texttt{free}(x)$. This program simply allocates a single cell and then frees it. Under the standard model, the smallest safe states are those of the form $(s, \emptyset)$ for any store s. For simplicity, assume that the only variables in the store are x and y. Define the specification to be the infinite set of triples that have the following form, for any a, b in $\mathbb{Z}$, and any a' in $\mathbb{N}$:

$$\{x = a \wedge y = b \wedge \texttt{emp}\}\, C\, \{x = a' \wedge y = b \wedge \texttt{emp}\}$$

Note that a' must be in $\mathbb{N}$ because only valid unallocated memory addresses can be assigned into x. It should be clear that this specification describes the exact behavior on all smallest safe states of C. Now we claim that the following triple is valid, but there is no way to derive it from the specification.

$$\{x = a \wedge y = b \wedge y \mapsto -\}\, C\, \{x = a' \wedge y = b \wedge y \mapsto - \wedge a' \neq b\}$$

The triple is clearly valid because a' must be a memory address that was initially unallocated, while address b was initially allocated. Nevertheless, there will not be any way to derive this triple, even if we come up with new assertion syntax or inference rules. The behavior of C on the larger state is different from the behavior on the small one, but there is no way to recover this fact once we make C opaque. It can be shown (see [12]) that if we add triples of the above form to our specification, then we will obtain a complete specification for C. Yet there is no straightforward way to see that such a specification is complete.

We will now formally prove that, in our system, there is a canonical form for complete specification. We first note that we will need to assume that our set of states is well-founded with respect to the substate relation (i.e., there is no infinite strictly-decreasing chain of states). This assumption is true for most standard models of Separation Logic, and furthermore, there is no reason to intuitively believe that the smallest safe states should be able to provide a complete specification when the assumption is not true.

We say that a specification Ψ is *complete for* C if, whenever $\models \{P\}\, C\, \{Q\}$ is valid, the triple $\vdash \{P\}\, C\, \{Q\}$ is derivable using only the inference rules that are not specific to the structure of C (i.e., the frame, consequence, disjunction, and conjunction rules), plus the following axiom rule:

$$\frac{\{P\}\, C\, \{Q\} \in \Psi}{\vdash \{P\}\, C\, \{Q\}}$$

For any σ, let $\sigma[\![C]\!]$ denote the set of all σ' such that $\sigma[\![C]\!]\sigma'$. For any set of states S, we define a *canonical specification on* S as the set of triples of the form $\{\{\sigma\}\}\, C\, \{\sigma[\![C]\!]\}$ for any state $\sigma \in S$. If there exists a canonical specification on S that is complete for C, then we say that S forms a *footprint* for C.

Theorem 4. *For any program C, the set of all smallest safe states of C forms a footprint for C.*

Proof. Let Ψ be the canonical specification of C on the set of all smallest safe states S. Consider any valid triple $\models \{P\}\, C\, \{Q\}$. We will show that for any $\sigma \in P$, we can derive the triple $\vdash \{\{\sigma\}\}\, C\, \{Q\}$ using our restricted set of inference rules — an application of the disjunction rule then completes the proof.

Consider any state $\sigma \in P$. Since $\models \{P\}\, C\, \{Q\}$ is valid, $\neg\sigma[\![C]\!]\mathtt{bad}$. We will show with a simple induction on the subheap operator that $\sigma_0 \preceq \sigma$ for some smallest safe state σ_0. Note that we can perform such an induction because the subheap operator is well-founded.

Case 1. σ is a smallest safe state. Then $\sigma \preceq \sigma$, and we are done.

Case 2. σ is not a smallest safe state. Since $\neg\sigma[\![C]\!]\mathtt{bad}$, σ is by definition a safe state. Therefore, there must be some strictly smaller safe state $\sigma_0 \prec \sigma$. By our induction hypothesis, $\sigma_0' \preceq \sigma_0$ for some smallest safe state σ_0'. Hence we have $\sigma_0' \preceq \sigma_0 \preceq \sigma$.

Now let $\sigma = \sigma_0 \bullet \sigma_1$, where σ_0 is a smallest safe state. Then there is an axiom $\{\{\sigma_0\}\}\, C\, \{\sigma_0[\![C]\!]\} \in \Psi$. We use the axiom rule to get $\vdash \{\{\sigma_0\}\}\, C\, \{\sigma_0[\![C]\!]\}$, followed by the frame rule to get $\vdash \{\{\sigma\}\}\, C\, \{\sigma_0[\![C]\!] * \{\sigma_1\}\}$. Consider any $\sigma' \in \sigma_0[\![C]\!] * \{\sigma_1\}$. Then $\sigma' = \sigma_0' \cdot \sigma_1$ for some σ_0' such that $\sigma_0[\![C]\!]\sigma_0'$. By the Forwards Frame Property, $\sigma[\![C]\!]\sigma'$. Since the triple $\models \{P\}\, C\, \{Q\}$ is valid and $\sigma \in P$, we see that $\sigma' \in Q$. Thus we have shown that $\sigma_0[\![C]\!] * \{\sigma_1\} \subseteq Q$, and so an application of the consequence rule gives us the desired $\vdash \{\{\sigma\}\}\, C\, \{Q\}$.

Note that while this theorem guarantees that the canonical specification is complete, we may not actually be able to write down the specification simply because the assertion language is not expressive enough. This would be the case for the behavior-preserving nondeterministic memory allocator if we used the assertion language presented in Section 3. We could, however, express canonical specifications in that system by extending the assertion language to talk about the free list (as briefly discussed at the end of Section 3).

5.2 Data Refinement

In [6], the goal is to formalize the concept of having a concrete module correctly implement an abstract one, within the context of Separation Logic. Specifically, the authors prove that as long as a client program "behaves nicely," any execution of the program using the concrete module can be tracked to a corresponding

execution using the abstract module. The client states in the corresponding executions are identical, so the proof shows that a well-behaved client cannot tell the difference between the concrete and abstract modules.

To get their proof to work out, the authors require two somewhat odd properties to hold. The first is called *contents independence*, and is an extra condition on top of the standard locality conditions. The second is called a *growing relation* — it refers to the relation connecting a state of the abstract module to its logically equivalent state(s) in the concrete module. All relations connecting the abstract and concrete modules in this way are required to be growing, which means that the domain of memory used by the abstract state must be a subset of that used by the concrete state. This is a somewhat unintuitive and restrictive requirement which is needed for purely technical reasons. We will show that behavior preservation completely eliminates the need for both contents independence and growing relations.

We now provide a formal setting for the data refinement theory. This formal setting is similar to the one in [6], but we will make some minor alterations to simplify the presentation. The programming language is defined as:

$$\begin{array}{rl} C & ::= \texttt{skip} \mid c \mid \texttt{m} \mid C_1; C_2 \mid \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2 \\ & \quad \mid \texttt{while}\, B\, \texttt{do}\, C \end{array}$$

c is a primitive command (sometimes referred to as "client operation" in this context). $\texttt{m}$ is a *module command* taken from an abstracted set **MOp** (e.g., a memory manager might implement the two module commands `cons` and `free`).

The abstracted client and module commands are assumed to have a semantics mapping them to particular local actions. We of course use our behavior-preserving notion of "local" here, whereas in [6], the authors use the three properties of safety monotonicity, the (backwards) frame property, and a new property called contents independence. It is trivial to show that behavior preservation implies contents independence, as contents independence is essentially a forwards frame property that can only be applied under special circumstances.

A *module* is a pair (p, η) representing a particular implementation of the module commands in **MOp**; the state predicate p describes the module's *invariant* (e.g., that a valid free list is stored starting at a location pointed to by a particular head pointer), while η is a function mapping each module command to a particular local action. The predicate p is required to be *precise* [11], meaning that no state can have more than one substate satisfying p (if a state σ does have a substate satisfying p, then we refer to that uniquely-defined state as σ_p). Additionally, all module operations are required to preserve the invariant p:

$$\neg\sigma(\eta\texttt{m})\texttt{bad} \wedge \sigma \in p * \texttt{true} \wedge \sigma(\eta\texttt{m})\sigma' \Longrightarrow \sigma' \in p * \texttt{true}$$

We define a big-step operational semantics parameterized by a module (p, η). This semantics is fundamentally the same as the one defined in [6]; the extended TR contains the full details. The only aspect that is important to mention here is that the semantics is equipped with a special kind of faulting called "access

violation." Intuitively, an access violation occurs when a client operation's execution depends on the module's portion of memory. More formally, it occurs when the client operation executes safely on a state where the module's memory is present (i.e., a state satisfying $p * \texttt{true}$), but faults when that memory is removed from the state.

The main theorem that we get out of this setup is a refinement simulation between a program being run in the presence of an abstract module (p, η), and the same program being run in the presence of a concrete module (q, μ) that implements the same module commands (i.e., $\lfloor \eta \rfloor = \lfloor \mu \rfloor$, where the floor notation indicates domain). Suppose we have a binary relation R relating states of the abstract module to those of the concrete module. For example, if our modules are memory managers, then R might relate a particular set of memory locations available for allocation to all lists containing that set of locations in some order. To represent that R relates abstract module states to concrete module states, we require that whenever $\sigma_1 R \sigma_2$, $\sigma_1 \in p$ and $\sigma_2 \in q$. Given this relation R, we can make use of the separating conjunction of Relational Separation Logic [14] and write $R * \texttt{Id}$ to indicate the relation relating any two states of the form $\sigma_p \cdot \sigma_c$ and $\sigma_q \cdot \sigma_c$, where $\sigma_p R \sigma_q$.

Now, for any module (p, η), let $C[(p, \eta)]$ be notation for the program C whose semantics have (p, η) filled in for the parameter module. Then our main theorem says that, if $\eta(f)$ simulates $\mu(f)$ under relation $R * \texttt{Id}$ for all $f \in \lfloor \eta \rfloor$, then for any program C, $C[(p, \eta)]$ also simulates $C[(q, \mu)]$ under relation $R * \texttt{Id}$. More formally, say that C_1 simulates C_2 under relation R (written $R; C_2 \subseteq C_1; R$) when, for all σ_1, σ_2 such that $\sigma_1 R \sigma_2$:

1.) $\sigma_1 [\![C_1]\!] \texttt{bad} \iff \sigma_2 [\![C_2]\!] \texttt{bad}$, and
2.) $\neg\, \sigma_1 [\![C_1]\!] \texttt{bad} \implies (\forall \sigma_2' \,.\, \sigma_2 [\![C_2]\!] \sigma_2' \Rightarrow \exists \sigma_1' \,.\, \sigma_1 [\![C_1]\!] \sigma_1' \wedge \sigma_1' R \sigma_2')$

Theorem 5. *Suppose we have modules (p, η) and (q, μ) with $\lfloor \eta \rfloor = \lfloor \mu \rfloor$ and a refinement relation R as described above, such that $R * \texttt{Id}; \mu(f) \subseteq \eta(f); R * \texttt{Id}$ for all $f \in \lfloor \eta \rfloor$. Then, for any program C, $R * \texttt{Id}; C[(q, \mu)] \subseteq C[(p, \eta)]; R * \texttt{Id}$.*

Proof. Straightforward by induction on C. We make use of behavior preservation in the base case when C is a primitive client command c. Note that this base case corresponds to Lemma 4 in [6], and it is the only place where contents independence and growing relations are used. Hence behavior preservation does indeed eliminate the need for these two concepts.

While the full proof can be found in the extended TR, we will semi-formally describe here the one case that highlights why behavior preservation eliminates the need for contents independence and growing relations: when C is simply a client command c. We wish to prove that $C[(p, \eta)]$ simulates $C[(q, \mu)]$, so suppose we have related states σ_1 and σ_2, and executing c on σ_2 results in σ_2'. Since σ_1 and σ_2 are related by $R * \texttt{Id}$, we have that $\sigma_1 = \sigma_p \cdot \sigma_c$ and $\sigma_2 = \sigma_q \cdot \sigma_c$. We know that (1) $\sigma_q \cdot \sigma_c \xrightarrow{c} \sigma_2'$, (2) c is local, and (3) c runs safely on σ_c because the

client operation's execution must be independent of the module state σ_q; thus the backwards frame property tells us that $\sigma_2' = \sigma_q \cdot \sigma_c'$ and $\sigma_c \xrightarrow{c} \sigma_c'$. Now, if c is behavior-preserving, then we can simply apply the forwards frame property, framing on the state σ_p, to get that $\sigma_p \# \sigma_c'$ and $\sigma_p \cdot \sigma_c \xrightarrow{c} \sigma_p \cdot \sigma_c'$, completing the proof for this case. However, without behavior preservation, contents independence and growing relations are used in [6] to finish the proof. Specifically, because we know that $\sigma_q \cdot \sigma_c \xrightarrow{c} \sigma_q \cdot \sigma_c'$ and that c runs safely on σ_c, contents independence says that $\sigma \cdot \sigma_c \xrightarrow{c} \sigma \cdot \sigma_c'$ for any σ whose domain is a subset of the domain of σ_q. Therefore, we can choose $\sigma = \sigma_p$ because R is a growing relation.

For example, suppose we have two memory manager modules that implement a free list in exactly the same way, except that one module stores its head pointer at memory location 100, while the other stores its pointer at location 200. It is very simple to prove that these two modules are equivalent using our system, but impossible using theirs (since neither module uses a subset of the other module's memory footprint).

5.3 Relational Separation Logic

Relational Separation Logic [14] allows for simple reasoning about the relationship between two executions. Instead of deriving triples $\{P\}\, C\, \{Q\}$, a user of the logic derives *quadruples* of the form:

$$\{R\} \begin{matrix} C \\ C' \end{matrix} \{S\}$$

R and S are binary relations on states, rather than unary predicates. Semantically, a quadruple says that if we execute the two programs in states that are related by R, then both executions are safe, and any termination states will be related by S. Furthermore, we want to be able to use this logic to prove program equivalence, so we also require that initial states related by R have the same divergence behavior. Formally, we say that the above quadruple is valid if, for any states σ_1, σ_2, σ_1', σ_2':

$$\begin{aligned}
&1.)\ \sigma_1 R \sigma_2 \implies \neg \sigma_1 [\![C]\!] \texttt{bad} \wedge \neg \sigma_2 [\![C']\!] \texttt{bad} \\
&2.)\ \sigma_1 R \sigma_2 \implies (\sigma_1 [\![C]\!] \texttt{div} \iff \sigma_2 [\![C']\!] \texttt{div}) \\
&3.)\ \sigma_1 R \sigma_2 \wedge \sigma_1 [\![C]\!] \sigma_1' \wedge \sigma_2 [\![C']\!] \sigma_2' \implies \sigma_1' S \sigma_2'
\end{aligned}$$

Relational Separation Logic extends the separating conjunction to work for relations, breaking related states into disjoint, correspondingly-related pieces:

$$\begin{aligned}
\sigma_1 (R * S) \sigma_2 \iff \exists\ \sigma_{1r},\ \sigma_{1s},\ \sigma_{2r},\ \sigma_{2s}\ .& \\
\sigma_1 = \sigma_{1r} \cdot \sigma_{1s} \wedge \sigma_2 = \sigma_{2r} \cdot \sigma_{2s} \wedge \sigma_{1r} R \sigma_{2r} \wedge \sigma_{1s} S \sigma_{2s}&
\end{aligned}$$

Just as Separation Logic has a frame rule for enabling local reasoning, Relational Separation Logic has a frame rule with the same purpose. This frame rule

says that, given that we can derive the quadruple above involving R, S, C, and C', we can also derive the following quadruple for any relation T:

$$\{R * T\} \begin{matrix} C \\ C' \end{matrix} \{S * T\}$$

In [14], it is shown that the frame rule is sound when all programs are deterministic but it is unsound if nondeterministic programs are allowed, so this frame rule cannot be used when we have a nondeterministic memory allocator.

To deal with nondeterministic programs, a solution is proposed in [14], in which the interpretation of quadruples is strengthened. The new interpretation for a quadruple containing R, S, C, and C' is that, for any σ_1, σ_2, σ_1', σ_2', σ, σ':

1.) $\sigma_1 R \sigma_2 \implies \neg\sigma_1[\![C]\!]\texttt{bad} \wedge \neg\sigma_2[\![C']\!]\texttt{bad}$
2.) $\sigma_1 R \sigma_2 \wedge \sigma_1 \# \sigma \wedge \sigma_2 \# \sigma' \implies ((\sigma_1 \cdot \sigma)[\![C]\!]\texttt{div} \iff (\sigma_2 \cdot \sigma')[\![C']\!]\texttt{div})$
3.) $\sigma_1 R \sigma_2 \wedge \sigma_1[\![C]\!]\sigma_1' \wedge \sigma_2[\![C']\!]\sigma_2' \implies \sigma_1' S \sigma_2'$

Note that this interpretation is the same as before, except that the second property is strengthened to say that divergence behavior must be equivalent not only on the initial states, but also on any larger states. It can be shown that the frame rule becomes sound under this stronger interpretation of quadruples.

In our behavior-preserving setting, it is possible to use the simpler interpretation of quadruples without breaking soundness of the frame rule. We could prove this by directly proving frame rule soundness, but instead we will take a shorter route in which we show that, when actions are behavior-preserving, a quadruple is valid under the first interpretation above if and only if it is valid under the second interpretation — i.e., the two interpretations are the same in our setting. Since the frame rule is sound under the second interpretation, this implies that it will also be sound under the first interpretation.

Clearly, validity under the second interpretation implies validity under the first, since it is a direct strengthening. To prove the inverse, suppose we have a quadruple (involving R, S, C, and C') that is valid under the first interpretation. Properties 1 and 3 of the second interpretation are identical to those of the first, so all we need to show is that Property 2 holds. Suppose that $\sigma_1 R \sigma_2$, $\sigma_1 \# \sigma$, and $\sigma_2 \# \sigma'$. By Property 1 of the first interpretation, we know that $\neg\sigma_1[\![C]\!]\texttt{bad}$ and $\neg\sigma_2[\![C']\!]\texttt{bad}$. Therefore, Termination Equivalence tells us that $\sigma_1[\![C]\!]\texttt{div} \iff (\sigma_1 \cdot \sigma)[\![C]\!]\texttt{div}$, and that $\sigma_2[\![C']\!]\texttt{div} \iff (\sigma_2 \cdot \sigma')[\![C']\!]\texttt{div}$. Furthermore, we know by Property 2 of the first interpretation that $\sigma_1[\![C]\!]\texttt{div} \iff \sigma_2[\![C']\!]\texttt{div}$. Hence we obtain our desired result.

Note In case the reader is curious, the reason that the frame rule under the first interpretation is sound when all programs are deterministic is simply that determinism (along with standard locality) implies Termination Equivalence. To see this, we only need to check the forwards direction, since standard locality requires the backwards one. Consider a situation where $\sigma_0 A \texttt{div}$, $\sigma_0 \# \sigma_1$, and

$\neg\sigma_0 A\texttt{bad}$. By Safety Monotonicity, we have $\neg(\sigma_0 \cdot \sigma_1)A\texttt{bad}$. Furthermore, suppose there is some σ such that $(\sigma_0 \cdot \sigma_1)A\sigma$. Then by the Backwards Frame Property, we have $\sigma = \sigma_0' \cdot \sigma_1$ and $\sigma_0 A\sigma_0'$. But we already know that $\sigma_0 A\texttt{div}$, so this contradicts the fact that A is deterministic. Therefore, A does not relate $\sigma_0 \cdot \sigma_1$ to `bad` or to any state σ. Since A is required to be total, we conclude that $(\sigma_0 \cdot \sigma_1)A\texttt{div}$.

5.4 Finite Memory

Since standard locality allows the program state to increase during execution, it does not play nicely with a model in which memory is finite. Consider any command that grows the program state in some way. Such a command is safe on the empty state but, if we extend this empty state to the larger state consisting of all available memory, then the command becomes unsafe. Hence such a command violates Safety Monotonicity.

There is one commonly-used solution for supporting finite memory without enforcing behavior preservation: say that, instead of faulting on the state consisting of all of memory, a state-growing command diverges. Furthermore, to satisfy Termination Monotonicity, we also need to allow the command to diverge on *any* state. The downside of this solution, therefore, is that it is only reasonable when we are not interested in the termination behavior of programs.

When behavior preservation is enforced, we no longer have any issues with finite memory models because program state cannot increase during execution. The initial state is obviously contained within the finite memory, so all states reachable through execution must also be contained within memory.

6 Related Work and Conclusions

The definition of locality (or local action), which enables the frame rule, plays a critical role in Separation Logic [8, 13, 15]. Almost all versions of Separation Logic — including their concurrent [3, 10, 4], higher-order [2], and relational [14] variants, as well as mechanized implementation (e.g., [1]) — have always used the same locality definition that matches the well-known Safety and Termination Monotonicity properties and the Frame Property [15].

In this paper, we argued a case for strengthening the definition of locality to enforce *behavior preservation*. This means that the behavior of a program when executed on a small state is identical to the behavior when executed on a larger state — put another way, excess, unused state cannot have any effect on program behavior. We showed that this change can be made to have no effect on the usage of Separation Logic, and we gave multiple examples of how it simplifies reasoning about metatheoretical properties. Behavior preservation has a particularly large impact on the theory of data refinement, as it opens up the possibility for proving module equivalence.

Determinism Constancy One related work that calls for comparison is the property of "Determinism Constancy" presented by Raza and Gardner [12], which is also a strengthening of locality. While they use a slightly different notion of action than we do, it can be shown that Determinism Constancy, when translated into our context (and ignoring divergence behaviors), is logically equivalent to:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0' \# \sigma_1 \Longrightarrow \sigma_0 \# \sigma_1 \wedge (\sigma_0 \cdot \sigma_1)[\![C]\!](\sigma_0' \cdot \sigma_1)$$

For comparison, we repeat our Forwards Frame Property here:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0 \# \sigma_1 \Longrightarrow \sigma_0' \# \sigma_1 \wedge (\sigma_0 \cdot \sigma_1)[\![C]\!](\sigma_0' \cdot \sigma_1)$$

While our strengthening of locality prevents programs from increasing state during execution, Determinism Constancy prevents programs from *decreasing* state. The authors use Determinism Constancy to prove the same property regarding footprints that we proved in Section 5.1. Note that, while behavior preservation does not imply Determinism Constancy, our concrete logic of Section 3 does have the property since it never decreases state (we chose to have the `free` command put the deallocated cell back onto the free list, rather than get rid of it entirely).

While Determinism Constancy is strong enough to prove the footprint property, it does not provide behavior preservation — an execution on a small state can still become invalid on a larger state. Thus it will not, for example, help in resolving the dilemma of growing relations in the data refinement theory. Due to the lack of behavior preservation, we do not expect the property to have a significant impact on the metatheory as a whole. Note, however, that there does not seem to be any harm in using *both* behavior preservation and Determinism Constancy. The two properties together enforce that the area of memory accessible to a program be constant throughout execution.

Module Reasoning Besides our discussion of data refinement in Section 5.2, there has been some previous work on reasoning about modules and their implementations. In [11], a "Hypothetical Frame Rule" is used to allow modular reasoning when a module's implementation is hidden from the rest of the code. In [2], a higher-order frame rule is used to allow reasoning in a higher-order language with hidden module or function code. However, neither of these works discuss relational reasoning between different modules. We are not aware of any relational logic for reasoning about modules.

For future work, it would be desirable to find more situations in the literature in which behavior preservation simplifies the theory or opens up new ideas to explore. One area that we are currently exploring is using behavior preservation in a security context. A corollary of behavior preservation is that there is no way for a program to determine any information about unused state (if there were, then this would imply a difference in behavior between executing the program with the unused state and executing without it). Therefore it would be perfectly safe to run the program in a state containing top secret data, as long as the program were known to execute safely on a state without that secret data.

Another direction for future work would be to define a behavior-preserving version of Concurrent Separation Logic that has the same inference rules as

standard CSL. The commands that acquire and release locks should be able to be expressed in a behavior-preserving fashion by including both local and shared state in the underlying state model. A lock acquire will move memory from the shared state into the local state, while a lock release will move it from local into shared. Neither command requires an increase in total state. The model could get quite interesting if we allow threads to allocate memory. One possible way to implement this might be to assign a separate free list to each thread. Another way might be to use a single free list, and, at any point in execution, we consider the free list to be owned by the currently-executing thread.

Acknowledgements. We thank Xinyu Feng and anonymous referees for suggestions and comments on an earlier version of this paper. This material is based on research sponsored by DARPA under agreement numbers FA8750-10-2-0254 and FA8750-12-2-0293, and by NSF grants CNS-0910670, CNS-0915888, and CNS-1065451. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these agencies.

References

1. A. W. Appel and S. Blazy. Separation logic for small-step cminor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 5–21, 2007.
2. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proc. 20th IEEE Symp. on Logic in Computer Science*, pages 260–269, 2005.
3. S. Brookes. A semantics for concurrent separation logic. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, 2004.
4. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 366–378, July 2007.
5. D. Costanzo and Z. Shao. A case for behavior-preserving actions in separation logic. Technical report, Dept. of Computer Science, Yale University, New Haven, CT, June 2012. `http://flint.cs.yale.edu/publications/bpsl.html`.
6. I. Filipovic, P. W. O'Hearn, N. Torp-Smith, and H. Yang. Blaming the client: on data refinement in the presence of pointers. *Formal Asp. Comput.*, 22(5):547–583, 2010.
7. G. Huet, C. Paulin-Mohring, et al. The Coq proof assistant reference manual. The Coq release v6.3.1, May 2000.
8. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symposium on Principles of Programming Languages*, pages 14–26, Jan. 2001.
9. B. W. Kernighan and D. M. Ritchie. *The C Programming Language (Second Edition)*. Prentice Hall, 1988.
10. P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67, 2004.

11. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3):1–50, 2009.
12. M. Raza and P. Gardner. Footprints in local reasoning. *Journal of Logical Methods in Computer Science*, 5(2), 2009.
13. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symp. on Logic in Computer Science*, pages 55–74, July 2002.
14. H. Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, 2007.
15. H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Proc. 5th Int'l Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002.