The Relationship Between Separation Logic and Implicit Dynamic Frames

Matthew J. Parkinson[1] and Alexander J. Summers[2]

[1] Microsoft Research Cambridge, mattpark@microsoft.com
[2] ETH Zurich, alexander.summers@inf.ethz.ch

Abstract. Separation logic is a concise method for specifying programs that manipulate dynamically allocated storage. Partially inspired by separation logic, Implicit Dynamic Frames has recently been proposed, aiming at first-order tool support. In this paper, we provide a total heap semantics for a standard separation logic, and prove it equivalent to the standard model. With small adaptations, we then show how to give a direct semantics to implicit dynamic frames and show this semantics correctly captures the existing definitions. This precisely connects the two logics. As a consequence of this connection, we show that a fragment of separation logic can be faithfully encoded in a first-order automatic verification tool (Chalice).

1 Introduction

Separation logic (SL) [5, 11] is a popular approach to specifying the behaviour of programs, as it naturally deals with the issues of aliasing. Separation logic assertions extend classical logic with extra connectives and predicates to describe memory layout. This makes it difficult to reuse current tool support for verification. Implicit Dynamic Frames (IDF) [15] was developed to give the benefits of separation logic specifications while leveraging existing tool support for first-order logic.

Although IDF was partially inspired by separation logic, there are many differences between SL and IDF that make understanding their relationship difficult. SL does not allow expressions that refer to the heap, while IDF does. SL is defined on partial heaps, while IDF is defined using total heaps and permission masks. The semantics of IDF are only defined by its translation to first-order verification conditions, while SL has a direct Kripke semantics for its assertions. These differences make it challenging to understand the relationship between the two approaches.

In this paper, we develop an extended separation logic that both captures the original semantics of separation logic, and correctly captures the semantics of IDF. To achieve this we provide a separation logic based on total heaps and a permission mask. The permission mask specifies the locations in the heap which are safe to access. Our formulation allows expressions that access the heap to be defined, however it complicates the definition of the separation logic

"magic wand" connective. In order to faithfully capture the original semantics of separation logic, and thus use magic wand to give the weakest pre-condition of commands, we present a non-standard definition of magic wand that includes changes to the total heap.

We also show that this extended separation logic correctly captures the semantics of the IDF formulas, we focus on the form of IDF found in the concurrent verification tool Chalice [9]. As the semantics of IDF formulas are only defined indirectly via weakest pre-condition calculations for a language using them, we show that the verification conditions (VCs) generated by the existing Boogie2 [8] encoding and the VCs generated from the separation logic proof rules are logically equivalent. This shows that our model directly captures the semantics of IDF.

This strong correspondence enables us to encode a fragment of separation logic containing separating conjunction, points to assertions, equalities and conditionals on pure assertions (a typical fragment used in verification tools), into Chalice - a tool based on first-order theorem proving.

Outline The paper is structured as follows. We begin by presenting the background definitions of both separation logic and implicit dynamic frames (§2); we then develop our extended separation logic (§3). We prove the correspondence between VCs in the two approaches (§4). Finally, we discuss related work (§5) and consider possible extensions and conclude (§6).

The contributions of this paper are as follows:

- We define a total heap semantics for separation logic, and prove it equivalent with the standard (partial heaps) semantics for the logic.
- We define a direct semantics for the implicit dynamic frames logic (the specification logic of the Chalice tool), which has so far only been given a semantics implicitly, via verification conditions.
- We show how to encode a standard fragment of separation logic into an implicit dynamic frames setting, preserving its semantics.
- We show that verification conditions as computed for separation logic coincide via our translation and semantics with the verification conditions computed by Chalice.

2 Background and Motivation

2.1 Standard Separation Logic

Separation logic [5, 11] is a verification logic which was originally introduced to handle the verification of sequential programs in languages with manual memory management such as C. The key feature of the logic is the ability to describe the behaviour of commands in terms of disjoint heap fragments, greatly simplifying the work required when "framing on" extra properties in a modular setting. Since its inception, separation logic has evolved in a variety of ways. In particular,

variants of separation logic are now used for the verification of object-oriented languages with garbage collection, such as Java and C♯ [12].

In order to handle concurrency, separation logic has been extended to consider its basic points-to assertions as *permissions* [10], determining which thread is allowed to read and write the corresponding state. To gain flexibility, *fractional permissions* [4, 3] were introduced, allowing the permissions governed by points-to assertions to be split and combined. A fractional permission is a rational number $0 < \pi \leq 1$, where 1 denotes full and exclusive (read/write) permission, and any other permission denotes read-only permission. In this paper we focus on the following core fragment of separation logic with fractional permissions.

Definition 1 (Separation Logic Assertions (*SL*)). *We assume a set of* object identifiers[3]*, ranged over by* ι*. We also assume a set of* field identifiers*, ranged over by* f*.* Values*, ranged over by* v *are either object identifiers, integers, or the special value* `null`*.*

The syntaxes of separation logic expressions (ranged over by e*) and assertions (ranged over by* a, b*) are defined as follows*[4]*. In this definition,* n *ranges over integer constants, and* $0 < \pi \leq 1$*.*

$$
\begin{array}{rl}
e & ::= x \mid \texttt{null} \mid n \\
a & ::= e = e \mid e.f \overset{\pi}{\mapsto} e \mid a * a \mid a \mathrel{-\!\!*} a \mid a \wedge a \mid a \vee a \mid a \rightarrow a
\end{array}
$$

We will refer to this separation logic simply as SL hereafter.

The key feature of separation logic is the facility to reason locally about separate heap portions. As such, the standard semantics for separation logic is formulated in terms of judgements parameterised by partial heaps (sometimes called *heap fragments*), which can be split and combined together as required. The critical new connectives are the *separating conjunction* $*$, and the *magic wand* $\mathrel{-\!\!*}$. The separating conjunction $a * b$ expresses that a and b are true and depend on disjoint fragments of the heap. The magic wand $a \mathrel{-\!\!*} b$ expresses that if any extra partial heap satisfying a is combined with the current partial heap, then the resulting heap is guaranteed to satisfy b.

Fractional permissions[5] are employed to manage shared memory concurrency in the usual way - a thread may only read from a heap location if it has a non-zero permission to the location, and it may only write to a location if it has the whole (full) permission to it. By careful permission accounting, it can then be guaranteed that a thread can never modify a heap location while another thread can read it. Note that permissions are handled (via points-to predicates $e.f \overset{\pi}{\mapsto} e'$) on a per-field basis: it is possible for an assertion to provide permission for only one field of an object. This fine granularity of permissions allows for greater

[3] These could be considered to be addresses, but we choose to be parametric with the concrete implementation of the heap.

[4] Note that variables x need not be program variables, but can also be specification-only variables (sometimes called *logical*, *ghost* or *specification variables*)

[5] Chalice, described in the next subsection, actually uses a slight variation on fractional permissions to make automatic theorem proving easier.

flexibility in the resulting logic - it can be specified that different threads have access to different fields of an object at the same time, for example. Combination of partial heaps includes combination of their permissions, where they overlap.

Definition 2 (Partial Fractional Heaps).

- *A* partial fractional heap h *is a partial function from pairs* (ι, f) *of object-identifier and field-identifier to pairs* (v, π) *of value and* non-zero *permission* π. *Partial heap lookup is written* $h[\iota, f]$, *and is only defined when* $(\iota, f) \in dom(h)$.
- *Partial heap extension:* $h_1 \subseteq h_2$, *iff* $\forall (\iota, f) \in dom(h_1).\ h_2[\iota, f] = h_1[\iota, f]$.
- *Partial heap compatible:* $h_1 \perp h_2$ *iff* $\forall (\iota, f) \in dom(h_1) \cap dom(h_2).\ \downarrow_1(h_1[\iota, f]) = \downarrow_1(h_2[\iota, f]) \ \wedge\ \downarrow_2(h_1[\iota, f]) + \downarrow_2(h_2[\iota, f]) \leq 1$.
- *The combination of two partial heaps, written* $h_1 * h_2$, *is defined only when* $h_1 \perp h_2$ *holds, by the following equations:*

$$dom(h_1 * h_2) = dom(h_1) \cup dom(h_2)$$
$$\forall (\iota, f) \in dom(h_1 * h_2).$$
$$(h_1 * h_2)[\iota, f] = \begin{cases} (\downarrow_1(h_1[\iota, f]), \downarrow_2(h_1[\iota, f])) & \text{if } (\iota, f) \notin dom(h_2) \\ (\downarrow_1(h_2[\iota, f]), \downarrow_2(h_2[\iota, f])) & \text{if } (\iota, f) \notin dom(h_1) \\ (\downarrow_1(h_1[\iota, f]), (\downarrow_2(h_1[\iota, f]) + \downarrow_2(h_2[\iota, f]))) & \text{otherwise} \end{cases}$$

We use $\downarrow_n$ *to denote the* n*th component of a tuple.*

There are two main flavours of separation logic studied in the literature: *classical* separation logic, and *intuitionistic* separation logic. In this paper, we consider *intuitionistic* separation logic. In intuitionistic separation logic, truth of assertions is closed under heap extension, which is appropriate for a garbage-collected language such as Java/C♯, rather than a language with manual memory management, such as C. The standard intuitionistic separation logic semantics for our fragment *SL* is defined as follows.

Definition 3 (Standard Semantics for *SL*). *Environments* σ *are partial functions*[6] *from variable names to values. Separation logic expression semantics,* $[\![e]\!]_\sigma$ *are defined by* $[\![x]\!]_\sigma = \sigma(x)$, $[\![n]\!]_\sigma = n$ *and* $[\![\mathtt{null}]\!]_\sigma = \mathtt{null}$. *The semantics of assertions is then as follows:*

$$h, \sigma \vDash_{SL} e_1.f \overset{\pi}{\mapsto} e_2 \iff \downarrow_2(h[[\![e_1]\!]_\sigma, f]) \geq \pi \ \wedge\ \downarrow_1(h[[\![e_1]\!]_\sigma, f]) = [\![e_2]\!]_\sigma$$
$$h, \sigma \vDash_{SL} e = e' \iff [\![e]\!]_\sigma = [\![e']\!]_\sigma$$
$$h, \sigma \vDash_{SL} a * b \iff \exists h_1, h_2.(\ h = h_1 * h_2 \ \wedge\ h_1, \sigma \vDash_{SL} a \ \wedge\ h_2, \sigma \vDash_{SL} b)$$
$$h, \sigma \vDash_{SL} a \mathrel{-\!\!*} b \iff \forall h'.(\ h' \perp h \ \wedge\ h', \sigma \vDash_{SL} a \ \Rightarrow\ h * h', \sigma \vDash_{SL} b)$$
$$h, \sigma \vDash_{SL} a \wedge b \iff h, \sigma \vDash_{SL} a \ \wedge\ h, \sigma \vDash_{SL} b$$
$$h, \sigma \vDash_{SL} a \vee b \iff h, \sigma \vDash_{SL} a \ \vee\ h, \sigma \vDash_{SL} b$$
$$h, \sigma \vDash_{SL} a \to b \iff \forall h'.(\ h \subseteq h' \ \wedge\ h', \sigma \vDash_{SL} a \ \Rightarrow\ h', \sigma \vDash_{SL} b)$$

[6] However, we assume that all applications of environments are well-defined; i.e., whenever we write $\sigma(x)$, that $x \in dom(\sigma)$. This assumption is justified so long as the program and specifications are type-checked appropriately.

The semantics for the separating conjunction and magic wand express the required splitting and combination of partial heaps. The semantics for logical implication $\rightarrow$ considers all possible extensions of the current heap, so that assertion truth is closed under heap extension [5].

Assume/Assert Verification in Boogie2 and related technologies uses two commands commonly to encode verification: `assume` A and `assert` A. The first allows the verification to work forwards with the additional assumption of A, while the second requires A to hold otherwise it will be considered a fault. These can be given weakest precondition semantics of:

$$wp(\texttt{assert}\ A, B) = A \wedge B \qquad wp(\texttt{assume}\ A, B) = A \Rightarrow B$$

From a verification perspective, these primitives can be used to encode many advanced language features. For example, in a modular verification setting with a first-order assertion language, a method-call is encoded by a sequence `assert` *pre*; `havoc`(*Heap*); `assume` *post*, in which *pre* and *post* are the pre- and post-conditions of the method respectively, and `havoc`(.) is a Boogie command that causes the prover to forget all knowledge about a variable/expression.

With separation logic, there are two forms of conjunction and implication, the standard (additive) ones $\wedge$ and $\rightarrow$, and the separating (multiplicative) ones $*$ and $-\!\!*$. This naturally gives rise to a second form of assume and assert for the multiplicative connectives (`assume`* and `assert`*), with the following weakest precondition semantics:

$$wp(\texttt{assert}^*\ A,\ B) = A * B \qquad wp(\texttt{assume}^*\ A,\ B) = A -\!\!* B$$

These commands can be understood as follows: `assert`* A removes a heap fragment satisfying A, and `assume`* A adds a heap fragment satisfying A. In a verification setting where assertions express permissions as well as functional properties, these can be used to correctly model the transfer of permissions when encoding various constructs. In a separation logic setting, a method call is encoded as `assert`* *pre*; `assume`* *post*.

In Chalice, which handles an assertion logic based on Implicit Dynamic Frames, functional verification is based on two new commands: `inhale` A and `exhale` A, which are also given an intuitive semantics of adding and removing access to state. One outcome of this paper is to make this intuitive connection between `inhale`/`exhale` and `assume`*/`assert`* formal, by defining a concrete and common semantics which can correctly characterise both assertion languages.

2.2 Chalice and Implicit Dynamic Frames

The original concept of Dynamic Frames comes from the PhD thesis of Kassios [7, 6]. The idea is to tackle the frame problem by allowing method specifications to declare the portion of the heap they may modify (a "frame" for the method call) via functions of the heap. The computed frames are therefore dynamic, in the sense that the actual values determined by these functions may change as the heap itself gets modified. Implicit Dynamic Frames [15, 14] takes a different approach to computing frames - a first-order logic is extended with a new kind

of assertion called an *accessibility predicate* (written e.g., as $acc(x.f)$) whose role is to represent a permission to a heap location $x.f$. In a method pre-condition, such an accessibility predicate indicates that the method requires permission to $x.f$ in order to be called - usually because this location might be read or written to in the method implementation. By imposing the restriction that heap dereference expressions (whether in assertions or in method bodies) are only allowed if a corresponding permission has already been acquired, this specification style allows a method frame to be calculated implicitly from its pre-condition.

Chalice [9] is a tool written for the automatic verification of concurrent programs. It handles a fairly simple imperative language, with classes (but no inheritance), and several interesting concurrency features (locks, channels, fork/join of threads). The tool proves partial correctness of method specifications, as well as absence of deadlock. The core of the methodology is based on the Implicit Dynamic Frames specification logic, using accessibility predicates to handle the permissions necessary to avoid data races between threads.

In this paper we ignore the deadlock-avoidance aspects of Chalice, and focus on the aspects which guarantee functional correctness. Verification in Chalice is defined via an encoding into Boogie2, in which two auxiliary Chalice commands `inhale` p and `exhale` p are used. These commands reflect the addition and removal of permissions from the state, as well as expressing assumptions and assertions about the heap. For example, method calls are represented by `exhale` *pre*; `inhale` *post*. The command `exhale` *pre* has the effect of giving up any permissions mentioned in accessibility predicates in *pre*, and generating `assert` statements for any logical properties such as heap equalities. Dually, `inhale` *post* has the effect of adding any permissions mentioned in *post* and *assuming* any logical properties.

Definition 4 (Our Chalice Subsyntax). *Expressions E and assertions p in our fragment of Chalice are given by the following syntax definitions:*

$$E ::= x \mid n \mid \texttt{null} \mid E.f$$
$$p ::= E = E \mid acc(E.f, \pi) \mid p * p$$

Note that Chalice actually uses the symbol for logical conjunction ($\wedge$ or &&) where we write $*$ above. However, in terms the semantics of the logic this is misleading - in general it is not the case that $p \wedge p$ (as written in Chalice) is equivalent to p. Chalice's conjunction treats permissions multiplicatively, that is $acc(x.f, 1) \wedge acc(x.f, 1)$ is actually equivalent to falsity. As we will show, Chalice conjunction is actually directly related to the separating conjunction of separation logic, hence our choice of notation here. Where we use the symbol $\wedge$ later in the paper, we mean the usual (additive) conjunction, just as in SL.

Chalice performs verification condition generation via an encoding into Boogie2, which makes use of two special variables $\mathcal{P}$ and $\mathcal{H}$. The former maps object identifier and field name pairs to permissions, in this instance a fractional permission, and is used for bookkeeping of permissions[7]. The latter maps object

[7] Technically, one should think of $\mathcal{P}$ as a *ghost variable*, since it does not correspond to real data of the original program.

identifier and field name pairs to values, and is used to model the heap of the real program. These maps can be read from (e.g., $\mathcal{P}[o, f]$) and updated (e.g., $\mathcal{P}[o, f] := 1$) from within the Boogie2 code, which allows Chalice to maintain their state appropriately to reflect the modifications made by the source program. In particular, the `inhale` and `exhale` commands have semantics which include modifications to the $\mathcal{P}$ map, to reflect the addition or removal of permissions by the program.

The critical aspect of Chalice's approach to data races, is to guarantee that assertions about the heap are only allowed when at least some permission is held to each heap location mentioned. This means that assertions cannot be made when it might be possible for other threads to be changing these locations - all logical properties used in the verification are then made robust to possible interference. Syntactically, this is enforced by requiring that assertions used in verification contracts are *self-framing* [6] - which means that the assertion includes enough accessibility predicates to "frame" its heap expressions. For example, the assertion $x.f = 5$ is not self-framing, since it refers to the heap location $x.f$ without permission. On the other hand, $(acc(x.f, 1) * x.f = 5)$ is self-framing.

3 A Total Heaps Semantics for Separation Logic

In this section we present a semantics for separation logic, which is based on states consisting of a *total* heap and a separate permission mask. Intuitively, the idea is that the permission mask specifies which locations in the heap we currently have permission to - this subset of the heap approximately corresponds with the partial heap which would be used in the standard semantics. The advantage of such a semantics is that it is simpler to relate to other logics with similar semantics and to encodings into first-order logic, as we will show later.

To facilitate comparisons, we define our semantics for an *extended* separation logic syntax, including not only the constructs of Definition 1, but also accessibility predicates from Implicit Dynamic Frames, and an enriched expression syntax that depend on the heap.

Definition 5 (Extended Separation Logic). *We define the expressions E and assertions A of* extended separation logic *(SL^+), by the following grammar (in which n stands for any integer constant):*

$$
\begin{aligned}
e &::= x \mid \mathbf{null} \mid n \\
E &::= e \mid E.f \\
A &::= E = E \mid e.f \overset{\pi}{\mapsto} e \mid A * A \mid A \mathrel{-\!\!*} A \mid A \wedge A \mid A \vee A \mid A \rightarrow A \mid acc(E.f, \pi)
\end{aligned}
$$

Note that the syntax of separation logic assertions (ranged over by a; see Definition 1) is a strict subset of the SL^+ assertions A defined above. The syntax of separation logic expressions e is also a strict subset of SL^+ expressions E.

Our aim is to give a total heap semantics for this more-general assertion language, implicitly defining a suitable semantics for both the fragment corresponding to SL assertions, and that corresponding to the assertions of Implicit

Dynamic Frames. This semantics depends on states which are specified by a combination of a total heap and a permission mask which separately tracks permissions to heap locations.

Definition 6 (Total Heaps and Permission Masks). *A* total heap H *is a total map from pairs of object-identifier* o *and field-identifier* f *to* values v*. Heap lookup is written* $H[o, f]$*.*

A permission mask Π *is a total map from pairs of object-identifier and field-identifier to permissions. Permission lookup is written* $\Pi[o, f]$*.*
We write $\Pi_1 \subseteq \Pi_2$ *for* permission extension*, i.e.,* $\forall (o, f).\ \Pi_1[o, f] \leq \Pi_2[o, f]$*.*
We write $\varnothing$ *for the* empty permission mask*; i.e., the mask which assigns* 0 *to all locations.*

Evaluation of extended separation logic expressions depends on a given environment and heap, and is defined by:

$$[\![x]\!]_{\sigma,H} = \sigma(x) \quad [\![n]\!]_{\sigma,H} = n \quad [\![E.f]\!]_{\sigma,H} = H[[\![E]\!]_{\sigma,H}, f] \quad [\![\texttt{null}]\!]_{\sigma,H} = \texttt{null}$$

The meaning of separation logic expressions is preserved (and is independent of the heap), as the following lemma shows:

Lemma 1. $\forall e, \sigma, H.\ [\![e]\!]_{\sigma,H} = [\![e]\!]_{\sigma}$

In order to define our semantics, we need to be able to combine permission masks:

Definition 7 (Combining Permission Masks). *Two permission masks* Π_1 *and* Π_2 *are* compatible*, written* $\Pi_1 \perp \Pi_2$*, if it holds that:*

$$\forall (o, f).\ \Pi_1[o, f] + \Pi_2[o, f] \leq 1$$

The combination *of two permission masks, written* $\Pi_1 * \Pi_2$ *is undefined if* Π_1 *and* Π_2 *are not compatible, and is otherwise defined pointwise to be the following permission mask:*

$$(\Pi_1 * \Pi_2)[o, f] = \Pi_1[o, f] + \Pi_2[o, f]$$

To define and prove results about our semantics, we need operations for replacing and removing portions of a heap, according to a specified permission. To this end, we introduce the following auxiliary definitions.

Definition 8 (Total Heap Operations). *Two heaps* H_1 *and* H_2 agree on permissions Π*, written* $H_1 \overset{\Pi}{\equiv} H_2$*, if the two heaps agree on all locations given non-zero permission by* Π*, i.e.,*

$$H_1 \overset{\Pi}{\equiv} H_2 \iff \forall o, f.\Pi[o, f] > 0 \ \Rightarrow\ H_2[o, f] = H_1[o, f]$$

The restriction of H to Π*, written* $H \restriction \Pi$ *is a* partial fractional heap*, defined by:*

$$\begin{aligned}
&dom(H \restriction \Pi) = \{(o, f) \mid \Pi[o, f] > 0\} \\
&\forall (o, f) \in dom(H \restriction \Pi).\downarrow_1((H \restriction \Pi)[o, f]) = H[o, f] \\
&\forall (o, f) \in dom(H \restriction \Pi).\downarrow_2((H \restriction \Pi)[o, f]) = \Pi[o, f]
\end{aligned}$$

The main difficulty in defining a semantics for *SL* using total heaps is getting the treatment of the magic wand connective correct. Since the standard semantics of this connective involves a quantification over all partial heaps which can be combined with the current one (i.e., all those which are compatible), it is not obvious how a corresponding definition can be made when starting from a total heap. The key idea is to model the "addition" of a new heap fragment which takes place in the standard semantics by acquiring some additional permissions, and then considering all possible heaps which have different values at the locations corresponding to these new permissions. In this way, the "newly acquired" region of our total heap can take on arbitrary new values. We model this by considering all heaps which agree with the current heap over the part to which any permission is held, and also satisfy the requirements of the "additional" heap.

Definition 9 (Total Heap Semantics for SL^+). *We define validity of SL^+-assertions with respect to a specified total heap H and permission mask Π recursively on the structure of the assertion:*

$$
\begin{aligned}
H, \Pi, \sigma \vDash_{SL^+} e.f \overset{\pi}{\mapsto} e' &\iff \Pi[[\![e]\!]_{\sigma,H}, f] \geq \pi \ \wedge\ H[[\![e]\!]_{\sigma,H}, f] = [\![e']\!]_{\sigma,H} \\
H, \Pi, \sigma \vDash_{SL^+} A * B &\iff \exists \Pi_1, \Pi_2.\ (\Pi = \Pi_1 * \Pi_2 \ \wedge\ H, \Pi_1, \sigma \vDash_{SL^+} A \ \wedge \\
&\qquad\qquad H, \Pi_2, \sigma \vDash_{SL^+} B) \\
H, \Pi, \sigma \vDash_{SL^+} A \mathrel{-\!\!*} B &\iff \forall \Pi', H'.(\ \Pi' \perp \Pi \ \wedge\ H' \overset{\Pi}{\equiv} H \ \wedge\ H', \Pi', \sigma \vDash_{SL^+} A \\
&\qquad\qquad \Rightarrow\ H', \Pi * \Pi', \sigma \vDash_{SL^+} B) \\
H, \Pi, \sigma \vDash_{SL^+} A \wedge B &\iff H, \Pi, \sigma \vDash_{SL^+} A \ \wedge\ H, \Pi, \sigma \vDash_{SL^+} B \\
H, \Pi, \sigma \vDash_{SL^+} A \vee B &\iff H, \Pi, \sigma \vDash_{SL^+} A \ \vee\ H, \Pi, \sigma \vDash_{SL^+} B \\
H, \Pi, \sigma \vDash_{SL^+} A \to B &\iff \forall \Pi', H'.(\ \Pi \subseteq \Pi' \ \wedge\ H' \overset{\Pi}{\equiv} H \ \wedge\ H', \Pi', \sigma \vDash_{SL^+} A \\
&\qquad\qquad \Rightarrow\ H', \Pi', \sigma \vDash_{SL^+} B) \\
H, \Pi, \sigma \vDash_{SL^+} acc(E.f, \pi) &\iff \Pi[[\![E]\!]_{\sigma,H}, f] \geq \pi \\
H, \Pi, \sigma \vDash_{SL^+} E = E' &\iff [\![E]\!]_{\sigma,H} = [\![E']\!]_{\sigma,H}
\end{aligned}
$$

Note the similarity between the definitions for magic wand $\mathrel{-\!\!*}$ and logical implication $\to$.[8] This is because both cases involve heap extension in the partial heap semantics; in our total heap semantics we model heap extension by enabling the assignment of new arbitrary values to the part of the heap we have not yet acquired permission to. We observe that validity of assertions in this semantics is closed under permission extension.

Proposition 1. *If $H, \Pi, \sigma \vDash_{SL^+} A$ and $\Pi \subseteq \Pi'$ then $H, \Pi', \sigma \vDash_{SL^+} A$.*

Proof. By straightforward induction on the structure of the assertion A.

Semantically, an assertion is *framed* by a permission mask if it only depends on the values of heap locations which the permission mask assigns permission to.

[8] Note the intuitionistic implication can be defined in terms of the pointwise classical implication ($\to_c$) in separation logic: $A \to B \iff \text{true} \mathrel{-\!\!*} (A \to_c B)$.

An assertion is semantically *self-framing* if it is only true for permission masks large enough to frame it. These semantic notions of framing and self-framing are formalised as follows:

Definition 10 (Framing, Self-Framing and Pure Assertions). *A permission mask Π'* frames *an assertion A (Π' frames A) if and only if:*

$$\forall \Pi, H, \sigma, H'.\ (H, \Pi, \sigma \vDash_{SL^+} A \wedge H' \overset{\Pi'}{\equiv} H \Rightarrow H', \Pi, \sigma \vDash_{SL^+} A)$$

An assertion A is self-framing *if and only if it is only satisfied under permission masks which frame it, i.e.,*

$$\forall \Pi, H, \sigma, H'.\ (H, \Pi, \sigma \vDash_{SL^+} A \wedge H' \overset{\Pi}{\equiv} H \Rightarrow H', \Pi, \sigma \vDash_{SL^+} A)$$

An assertion A is pure[9] *if and only if it doesn't depend on permissions, i.e.,*

$$\forall \Pi, H, \sigma.\ (H, \Pi, \sigma \vDash_{SL^+} A \Rightarrow H, \varnothing, \sigma \vDash_{SL^+} A)$$

For example, the assertion $x.f = 5$ is only framed by permission masks which give permission to location (x, f). It is not self-framing, since with an environment σ such that $\sigma(x) = \iota$, a heap H such that $H[\iota, f] = 5$ and an empty permission mask Π, we have $H, \Pi, \sigma \vDash_{SL^+} x.f = 5$. However, the heap $H' = H[(\iota, f) \mapsto 4]$ satisfies $H' \overset{\Pi}{\equiv} H$, but we have $H', \Pi, \sigma \nvDash_{SL^+} x.f = 5$. On the other hand, the assertions $acc(x.f, 1) * x.f = 5$ and $x.f \overset{1}{\mapsto} 5$ are both self-framing.

Intuitively, self-framing assertions are robust to arbitrary interference on the rest of the heap. For separation logic assertions, this property holds naturally, since it is impossible for an assertion to talk about the heap without including the appropriate "points-to" predicates, which force the corresponding permissions to be held.

Lemma 2. *All separation logic assertions a (Defn 1) are self-framing.*

Proof. We prove, by straightforward induction on the structure of the assertion a, the equivalent statement:

$$\forall a, H, \Pi.\ (H, \Pi, \sigma \vDash_{SL^+} a \Leftrightarrow \forall H'.\ (H' \overset{\Pi}{\equiv} H \Rightarrow H', \Pi, \sigma \vDash_{SL^+} a))$$

We now turn to relating our total heap semantics for separation logic, with the standard semantics. To do this, we need to relate partial heaps with pairs of total heap and permission mask. Given any total heap H and permission mask Π we can construct a corresponding partial heap $H \upharpoonright \Pi$. Conversely, any partial heap h can be represented as the restriction of a total heap H to the permission mask corresponding to all the permissions in h. This representation however, is not unique - there are many such total heaps H we could choose such that $h = H \upharpoonright \Pi$. However, the different choices of H can only differ over the locations given no permission in Π, and the previous lemma demonstrates that such differences do not affect the semantics of assertions. For our correspondence result, it is therefore without loss of generality to consider partial heaps constructed by $H \upharpoonright \Pi$. We can then show that our total heap semantics for SL is sound and complete with respect to the standard semantics:

[9] We may have considered *pure* assertions to depend on the heap as well. This however, does not have the right logical characterisation: A is pure iff $\forall B, A * B = A \wedge B$.

Theorem 1 (Correctness of Total Heap Semantics). *For all SL-assertions a, environments σ, total heaps H, and permission mask Π:*

$$H, \Pi, \sigma \vDash_{SL^+} a \iff (H \upharpoonright \Pi), \sigma \vDash_{SL} a$$

Proof. By induction on the structure of the assertion a, using Lemma 1.

This result demonstrates that our total heap semantics correctly models the standard semantics of separation logic assertions. However, because our assertion language is more general than that of separation logic, not all properties of the separation logic connectives transfer across to the full generality of SL^+. For example, in separation logic, the assertions $a \mathrel{-\!\!*} (b \mathrel{-\!\!*} c)$ and $(a * b) \mathrel{-\!\!*} c$ are (always) equivalent. This is not quite the case in SL^+. In order to precisely characterise the laws which hold, we require a notion of semantic entailment.

Definition 11 (Semantic Entailment, Validity and Equivalence). *A SL^+ assertion A is* semantically valid *(written $\vDash_{SL^+} A$) if it holds in all situations; i.e.,*

$$\vDash_{SL^+} A \Leftrightarrow \forall H, \Pi, \sigma.\ H, \Pi, \sigma \vDash_{SL^+} A$$

Given SL^+ assertions A and B, we say that A semantically entails *B (and write $A \vDash_{SL^+} B$) if and only if B holds whenever A does; i.e.,*

$$A \vDash_{SL^+} B \ \Leftrightarrow\ \forall H, \Pi, \sigma.\ (H, \Pi, \sigma \vDash_{SL^+} A \ \Rightarrow\ H, \Pi, \sigma \vDash_{SL^+} B)$$

Given SL^+ assertions A and B, we say that A is equivalent to *B (and write $A \equiv_{SL^+} B$) if and only if $A \vDash_{SL^+} B$ and $B \vDash_{SL^+} A$.*

We can now show how various laws which hold for separation logic transfer (in some cases partially) to our more general setting of SL^+.

Proposition 2. *For all SL^+ assertions A_1, A_2, A_3:*

1. $\vDash_{SL^+} A_1 * (A_1 \mathrel{-\!\!*} A_2) \to A_2$ *and* $\vDash_{SL^+} A_1 \wedge (A_1 \to A_2) \to A_2$
2. $A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3) \vDash_{SL^+} (A_1 * A_2) \mathrel{-\!\!*} A_3$ *and* $A_1 \to (A_2 \to A_3) \vDash_{SL^+} (A_1 \wedge A_2) \to A_3$
3. $\forall H, \Pi, \sigma$, *if* $\forall \Pi'.\ (\Pi' \perp \Pi) \wedge H, \Pi', \sigma \vDash_{SL^+} A_1 \ \Rightarrow\ \Pi * \Pi'$ *frames* A_1, *then:* $H, \Pi, \sigma \vDash_{SL^+} (A_1 * A_2) \mathrel{-\!\!*} A_3 \ \Rightarrow\ H, \Pi, \sigma \vDash_{SL^+} A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3)$
4. $\forall H, \Pi, \sigma$, *if* $\forall \Pi'.\ (\Pi' \perp \Pi) \wedge H, \Pi * \Pi', \sigma \vDash_{SL^+} A_1 \ \Rightarrow\ \Pi * \Pi'$ *frames* A_1, *then:* $H, \Pi, \sigma \vDash_{SL^+} (A_1 \wedge A_2) \to A_3 \ \Rightarrow\ H, \Pi, \sigma \vDash_{SL^+} A_1 \to (A_2 \to A_3)$
5. *If* $A_1 \vDash_{SL^+} (A_2 \mathrel{-\!\!*} A_3)$ *then* $(A_1 * A_2) \vDash_{SL^+} A_3$
6. *If A_1 is self-framing and* $(A_1 * A_2) \vDash_{SL^+} A_3$ *then* $A_1 \vDash_{SL^+} (A_2 \mathrel{-\!\!*} A_3)$

To see that the usual separation logic laws do not all hold in general, consider for example the two assertions $P_1 \stackrel{def}{=} (x.f = 1 \mathrel{-\!\!*} (x.f = 2 \mathrel{-\!\!*} 1 = 2))$ and $P_2 \stackrel{def}{=} (x.f = 1 * x.f = 2) \mathrel{-\!\!*} 1 = 2$. The assertion P_2 is always true, essentially because no heap exists which satisfies $(x.f = 1 * x.f = 2)$, and so the implication in the semantics of the wand holds vacuously. However, the assertion P_1 is not always true - if we consider the case where we do not have permission to $x.f$ when checking the wand, we can pick two heaps which agree on our existing permissions and which

assign $x.f$ the values 1 and 2 respectively. However, $1 = 2$ will of course be false in any configuration.

The usual separation logic laws do however hold for self-framing assertions (which by Lemma 2) includes all separation logic assertions).

Corollary 1. *For all self-framing SL^+ assertions A_1, A_2, A_3:*

1. $\vDash_{SL^+} A_1 * (A_1 \mathrel{-\!\!*} A_2) \to A_2$
2. $\vDash_{SL^+} A_1 \wedge (A_1 \to A_2) \to A_2$
3. $A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3) \equiv_{SL^+} (A_1 * A_2) \mathrel{-\!\!*} A_3$
4. $A_1 \to (A_2 \to A_3) \equiv_{SL^+} (A_1 \wedge A_2) \to A_3$
5. $A_1 \vDash_{SL^+} (A_2 \mathrel{-\!\!*} A_3)$ *if and only if* $(A_1 * A_2) \vDash_{SL^+} A_3$

To complete this section, we observe that we are able to eliminate the "points-to" assertions from our syntax without loss of expressiveness. This is because of the following proposition:

Proposition 3. *For all e,f,e',π we have $e.f \overset{\pi}{\mapsto} e' \equiv_{SL^+} acc(e.f, \pi) * e.f = e'$.*

Proof. Directly from the semantics.

This result along with Theorem 1 shows that we can faithfully represent *SL* assertions in an implicit dynamic frames logic, in which permissions are tracked by accessibility predicates, and assertions about the heap are managed independently. Because our proofs are inductive on the structure of assertions, this representation result can also be applied to any fragments of *SL*. In particular, if we take the core fragment of *SL* typically supported by tools (in which assertions are built from separating conjunction and a restricted form of implication in which $A \to B$ is only allowed if A is permission-free), then we can faithfully encode this fragment into the logic of Chalice (Definition 4).

However, Chalice has its own semantics for this logic, which is implicitly defined via the weakest-precondition semantics for the language. Therefore, in order to provide a strong connection between standard separation logic and the Chalice methodology we must also show that our total heap semantics can be used to accurately reflect the semantics of Chalice. This is the focus of the next section.

4 Verification Conditions

In this section, we precisely connect the semantics of our assertion language with Chalice. Chalice does not provide a direct model for its assertion language. It instead defines the semantics of assertions using the weakest pre-condition semantics of the commands `inhale` and `exhale`. We show that this semantics precisely corresponds with the semantics in SL^+.

4.1 Chalice

Chalice is defined by a translation into Boogie2 [8], which generates verification conditions on a many-sorted classical logic with first-order quantification. It has sorts for mathematical maps, which are used by Chalice to encode both the heap and the permission mask. We use ϕ to range over formulas in this logic, and $\sigma \vDash_{FO} \phi$ to mean ϕ holds in the standard semantics of first-order logic given the interpretation of free variables σ, and $\vDash_{FO} \phi$ means holds in all interpretations.

The definitions throughout this section generate expressions that have these two specific free variables: $\mathcal{H}$ for the current heap, and $\mathcal{P}$ for the current permission masks. Thus, $\mathcal{H}[x, f] = 5$ means in the current heap the variable x's field named f contains value 5. In the assertion logic, this corresponds to $x.f = 5$ where the heap access is implicit.

To enable us to relate the verification conditions in separation logic with those in Chalice, we need to be able to relate formulas in one approach with the other. We can provide a syntactic translation from the Chalice assertion logic into the first-order logic.

Definition 12. *We translate expressions that implicitly access the heap into expressions that explicitly access the heap as follows:*

$$[\![x]\!] = x \qquad [\![\mathtt{null}]\!] = \mathtt{null} \qquad [\![E.f]\!] = \mathcal{H}[[\![E]\!], f]$$

and we translate formulas as follows:

$$\begin{aligned} [\![acc(E.f, \pi)]\!] &= \mathcal{P}[[\![E]\!], f] \geq \pi \\ [\![p * q]\!] &= \exists \mathcal{P}_1, \mathcal{P}_2.\ [\![p]\!][\mathcal{P}_1/\mathcal{P}] \wedge [\![q]\!][\mathcal{P}_2/\mathcal{P}] \wedge (\mathcal{P}_1 * \mathcal{P}_2 = \mathcal{P}) \\ [\![E = E']\!] &= [\![E]\!] = [\![E']\!] \end{aligned}$$

where $[\mathcal{P}_1/\mathcal{P}]$ means the replacement of $\mathcal{P}$ with $\mathcal{P}_1$, and $\mathcal{P}_1 * \mathcal{P}_2 = \mathcal{P}$ is a ternary predicate, true if and only if $\forall i. \mathcal{P}_1(i) + \mathcal{P}_2(i) \leq 1 \wedge \mathcal{P}(i) = \mathcal{P}_1(i) + \mathcal{P}_2(i)$.

For example, the formula $acc(x.f, \pi) * x.f = 5$ will be translated to

$$\exists \mathcal{P}_1, \mathcal{P}_2.\ (\mathcal{P}_1[x.f] \geq \pi) \wedge (\mathcal{H}[x, f] = 5) \wedge \mathcal{P} = \mathcal{P}_1 * \mathcal{P}_2$$

which we can simplify to $(\mathcal{P}[x.f] \geq \pi) \wedge (\mathcal{H}[x, f] = 5)$, that is the current heap contains 5 at x, f and the current permission mask has at least π permission on that location.

By interpreting the heap variable with concrete heap, and the permission mask variable with a concrete permission mask, we can show that the translated formula is true iff the original SL^+ formula was true.

Lemma 3. $\sigma, \mathcal{H} \mapsto H, \mathcal{P} \mapsto \Pi \vDash_{FO} [\![p]\!] \iff H, \Pi, \sigma \vDash_{SL^+} p$
where $\sigma, \mathcal{H} \mapsto H, \mathcal{P} \mapsto \Pi$ is an interpretation that has all the mappings of σ and additionally maps the current heap, $\mathcal{H}$, and permission mask, $\mathcal{P}$, to the heap, H, and the permission mask, Π.

Chalice does not allow arbitrary formulas to be used as argument to `inhale` and `exhale`: it restricts the formulas to be self-framing. Chalice does not use the semantic check from earlier, but instead uses a syntactic formulation that checks self-framing from left-to-right. Note that this means that syntactic self-framing is not symmetric with respect to $*$. For instance, $acc(x.f, \pi) * x.f = 5$ is syntactically self-framing, but $x.f = 5 * acc(x.f, \pi)$ is not. Somewhat surprisingly this is required by the way the verification conditions are generated.

Definition 13. *We define the footprint*[10] *of a formula; an expression with the $\mathcal{H}$ variable free in it, that has the type of a set of locations and field name pairs.*

$$\mathit{foot}(E = E') = \{\} \qquad \mathit{foot}(acc(E.f, \pi)) = \{(\llbracket E \rrbracket, f)\}$$
$$\mathit{foot}(A * B) = \mathit{foot}(A) \cup \mathit{foot}(B)$$

We define a boolean expression $\mathit{syn\_framed}_\psi(E)$ to mean that all the fields mentioned in E are in the set ψ.

$$\mathit{syn\_framed}_\psi(E.f) = \mathit{syn\_framed}_\psi(E) \wedge \llbracket E \rrbracket.f \in \psi$$
$$\mathit{syn\_framed}_\psi(x) = \mathit{syn\_framed}_\psi(\mathtt{null}) = \mathsf{True}$$

We lift this to formulas as

$$\mathit{syn\_framed}_\psi(E = E') \iff \mathit{syn\_framed}_\psi(E) \wedge \mathit{syn\_framed}_\psi(E')$$
$$\mathit{syn\_framed}_\psi(acc(E.f, \pi)) \iff \mathit{syn\_framed}_\psi(E.f)$$
$$\mathit{syn\_framed}_\psi(A * B) \iff \mathit{syn\_framed}_\psi(A) \wedge \mathit{syn\_framed}_{\psi \cup \mathit{foot}(A)}(B)$$

Note that when we check that B is framed in $A * B$, we can use the footprint of A; these syntactic checks do not treat $*$ as associative and commutative.

A formula, A, is syntactically self-framing, if it is framed by the empty set, $\mathit{syn\_framed}_\emptyset(A)$.

We can now provide the definitions of the weakest pre-conditions of the commands for `inhale` and `exhale`. In Figure 1, we present the weakest pre-conditions of commands in Chalice from [9]. We write $wp_{\mathrm{ch}}(C, \phi)$ for the weakest pre-condition of the command C given the post-condition ϕ. Chalice models the inhaling and exhaling of permission by mutating the permission mask variable. To exhale an equality (or any formula not mentioning the permission mask) we simply assert it must be true. This does not need to modify the permission mask. To exhale $p * q$, first we exhale p and then q. When an access predicate is exhaled, first we check that the permission mask contains sufficient permission, and then we remove the permission from the mask.

To inhale an equality, it is simply the same as assuming it. To inhale a $p * q$, we first inhale p and then q. There are two cases for inhaling a permission: (1) we don't currently have any permission to that location; and (2) we do currently have permission to that location. The first case proceeds by adding the permission, and then havocing the contents of that location, that is, making sure any previous value of the variable has been forgotten. The second case simply adds the permission to the permission mask.

[10] $\mathsf{foot}(\_)$ corresponds to the *required access set* in [15].

$$wp_{\text{ch}}(\texttt{exhale}(E = E'), \phi) = wp_{\text{ch}}(\texttt{assert } \lfloor\!\lfloor E = E' \rfloor\!\rfloor, \phi)$$

$$wp_{\text{ch}}(\texttt{exhale}(p_1 * p_2), \phi) = wp_{\text{ch}}(\texttt{exhale}(p_1); \texttt{exhale}(p_2), \phi)$$

$$\begin{aligned} &wp_{\text{ch}}(\texttt{exhale}(acc(E.f, \pi), \phi) \\ &\quad = wp_{\text{ch}}(\texttt{assert}(\mathcal{P}[\lfloor\!\lfloor E \rfloor\!\rfloor, f]) \geq \pi; \mathcal{P}[\lfloor\!\lfloor E \rfloor\!\rfloor, f] := \mathcal{P}[\lfloor\!\lfloor E \rfloor\!\rfloor, f] - \pi, \phi) \end{aligned}$$

$$wp_{\text{ch}}(\texttt{inhale}(E = E'), \phi) = wp_{\text{ch}}(\texttt{assume } \lfloor\!\lfloor E = E' \rfloor\!\rfloor, \phi)$$

$$wp_{\text{ch}}(\texttt{inhale}(p_1 * p_2), \phi) = wp_{\text{ch}}(\texttt{inhale}(p_1); \texttt{inhale}(p_2), \phi)$$

$$\begin{aligned} &wp_{\text{ch}}(\texttt{inhale}(acc(E.f, \pi), \phi) \\ &\quad = wp_{\text{ch}}(\texttt{assume}(\mathcal{P}[\lfloor\!\lfloor E \rfloor\!\rfloor, f] = 0); \mathcal{P}[\lfloor\!\lfloor E \rfloor\!\rfloor, f] := \pi; \texttt{havoc}(\mathcal{H}[\lfloor\!\lfloor E \rfloor\!\rfloor, f]), \phi) \\ &\qquad \wedge wp_{\text{ch}}(\texttt{assume}(0 < \mathcal{P}[\lfloor\!\lfloor E \rfloor\!\rfloor, f] \leq 1 - \pi); \mathcal{P}[\lfloor\!\lfloor E \rfloor\!\rfloor, f] \texttt{+=} \pi; , \phi) \end{aligned}$$

where

$$\begin{aligned} &wp_{\text{ch}}(\mathcal{P}[o, f] := x, \phi) = \phi[upd(\mathcal{P}, (o, f), x)/\mathcal{P}] \\ &wp_{\text{ch}}(\texttt{havoc}(\mathcal{H}[x, f]), \phi) = \phi[upd(\mathcal{H}, (x, f), z)/\mathcal{H}] \qquad \text{fresh z} \\ &wp_{\text{ch}}(\texttt{assume } E, \phi) = E \to \phi \\ &wp_{\text{ch}}(\texttt{assert } E, \phi) = E \wedge \phi \\ &wp_{\text{ch}}(C_1; C_2, \phi) = wp_{\text{ch}}(C_1, wp_{\text{ch}}(C_2, \phi)) \end{aligned}$$

where $upd(a, b, c)[b] = c$ and $upd(a, b, c)[d] = a[d]$ provided $d \neq b$.

Fig. 1. Abridged weakest pre-condition semantics for Chalice [9]

4.2 Relationship

In the rest of this section, we show that the verification conditions (VCs) generated by Chalice are equivalent to those generated by SL^+. We focus on the `inhale` and `exhale` commands as these represent the semantics of the Chalice assertion language. By showing the equivalence, we show that our model of SL^+ is also a model for Chalice.

We write $wp_{\text{sl}}(C, A)$, to be the weakest pre-condition in SL^+ of the formula A with respect to the command C. We treat `inhale` and `exhale` as the multiplicative versions of `assume` and `assert` (see §2.1), and thus have the following weakest pre-conditions:

$$wp_{\text{sl}}(\texttt{exhale}(A), B) = A * B \qquad wp_{\text{sl}}(\texttt{inhale}(A), B) = A \mathrel{-\!\!*} B$$

Our core result is to show that both `inhale` and `exhale` have equivalent VCs in the two approaches.

Definition 14 (*equiv*(C)). *We define the VCs of a command as equivalent in both systems, equiv(C), iff for every self-framing SL^+ assertions, A, we have*

$$\vDash_{FO} \lfloor\!\lfloor wp_{\text{sl}}(C, A) \rfloor\!\rfloor \iff wp_{\text{ch}}(C, \lfloor\!\lfloor A \rfloor\!\rfloor)$$

Our notion of equivalence of VCs only requires commands to have equivalent weakest pre-conditions for self-framing post-conditions, thus we need to show that each command preserves self-framing. If commands did not preserve self-framing, then sequencing could not be proved by induction on the sub-commands.

Lemma 4. *Each command preserves semantic self-framing: If A and p are self-framing, then so are $wp_{\mathrm{sl}}(\texttt{inhale}\ p, A)$ and $wp_{\mathrm{sl}}(\texttt{exhale}\ p, A)$.*

The key to showing our semantics for SL^+ correctly embodies Chalice is to show that the VCs generated for the `inhale` and `exhale` commands are equivalent. The `exhale` is straightforward.

Lemma 5. $\forall p.\ equiv(\texttt{exhale}\ p)$

Proof. By induction on p.

The proof of `inhale` is more involved. This depends on the inhaled formula being syntactically self-framing. We must first prove a collection of lemmas to enable the proof to proceed by induction. (1) We need to connect the weakest pre-condition of an inhale with the footprint of a formula: that is, know that inhaling a formula adds its footprint to the permission mask. (2) The Chalice's VCs break `inhale` $p*q$ into two `inhale`s. This logically corresponds to currying/uncurrying a wand formula in SL^+, but this is only true if the first formula is framed in the current world, so we need an analog of this for the VC world. Finally, (3) we need to know that introducing additional havocs to locations outside the current permission mask does not affect the weakest pre-condition assuming that the post-condition is framed by the current permission mask.

Lemma 6.

1. $wp_{\mathrm{ch}}(\texttt{assume}(\psi < \mathcal{P}); \texttt{inhale}\ p, A)$ *is equivalent to* $wp_{\mathrm{ch}}(\texttt{assume}(\psi < \mathcal{P}); \texttt{inhale}\ p\ ; \texttt{assume}(\psi \cup \mathsf{foot}(p) < \mathcal{P}), A)$*;*
2. *If* $\mathsf{syn\_framed}_{\psi}(p_1)$*, then* $wp_{\mathrm{ch}}(\texttt{assume}(\psi < \mathcal{P}), [\![p_1 \mathrel{-\!\!*} (p_2 \mathrel{-\!\!*} A)]\!])$ *is equivalent to* $wp_{\mathrm{ch}}(\texttt{assume}(\psi < \mathcal{P}), [\![(p_1 * p_2) \mathrel{-\!\!*} A]\!])$*; and*
3. *If* $\mathsf{syn\_framed}_{\psi}(A)$*, then* $wp_{\mathrm{ch}}(\texttt{assume}(\psi < \mathcal{P}); \texttt{havoc}(\mathcal{H}[\overline{\mathcal{P}}]), [\![A]\!])$ *is equivalent to* $wp_{\mathrm{ch}}(\texttt{assume}(\psi < \mathcal{P}), [\![A]\!])$.

where $\psi < \mathcal{P}$ means $\forall i \in \psi.\mathcal{P}[i] \neq 0$, and $\texttt{havoc}(\mathcal{H}[\overline{\mathcal{P}}])$ means `havoc` *every field in the heap with no permission in $\mathcal{P}$.*

Proof. 1. By induction on p.
2. Direct consequence of Proposition 2.3 and 2.2.
3. From definition of framing.

We want to show that if p is syntactically self-framing, then `inhale` p is equivalent in both approaches. However, we need to prove a stronger fact that accounts for the permissions we may have inhaled so far. In particular, as `inhale` $p_1 * p_2$ is implemented by first inhaling p_1 and then p_2, when we consider inhaling p_2 it may not be self-framing. However, the context will have inhaled sufficient permissions that it is framed in that context. We prove that the VCs are equivalent in a context in which the inhale is framed.

Lemma 7. *If* $\mathsf{syn\_framed}_{\psi}(p)$ *and* $\mathsf{syn\_framed}_{\psi \cup \mathsf{foot}(p)}(A)$*, then*

$$wp_{\mathrm{ch}}(\texttt{assume}(\psi < \mathcal{P});\ \texttt{inhale}\ p, [\![A]\!]) \iff wp_{\mathrm{ch}}(\texttt{assume}(\psi < \mathcal{P}), [\![wp_{\mathrm{sl}}(\texttt{inhale}\ p, A)]\!])$$

Proof. By induction on p. The $*$ case uses Lemma 6.1 to rearrange the program so that the inductive hypothesis can be used on q_2, and Lemma 6.2. is used to rearrange separation logic assertion:

$$
\begin{aligned}
& wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P});\ \texttt{inhale}\ q_1 * q_2, [\![Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}); \texttt{inhale}\ q_1; \texttt{assume}(\psi \cup \mathsf{foot}(q_1) < \mathcal{P}); \texttt{inhale}\ q_2, [\![Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}); \texttt{inhale}\ q_1; \texttt{assume}(\psi \cup \mathsf{foot}(q_1) < \mathcal{P}); [\![q_2 \mathrel{-\!\!*} Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}); \texttt{inhale}\ q_1, [\![q_2 \mathrel{-\!\!*} Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}), [\![q_1 \mathrel{-\!\!*} (q_2 \mathrel{-\!\!*} Q)]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}), [\![(q_1 * q_2) \mathrel{-\!\!*} Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}), [\![wp_{\text{sl}}(\texttt{inhale}\ q_1 * q_2, Q)]\!])
\end{aligned}
$$

The *acc* case uses Lemma 6.3 to show that their is no difference between havocing just the inhaled location, and havoc all locations that you do not have permissions to. Assume that $\mathcal{P}[[\![E]\!].f] = 0$,

$$
\begin{aligned}
& wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}); \texttt{inhale}\ acc(E.f, \pi), [\![Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}); \texttt{inhale}\ acc(E.f, \pi); \texttt{havoc}(\mathcal{H}[\overline{\mathcal{P}}]), [\![Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}); \mathcal{P}[[\![E]\!].f] := \pi; \texttt{havoc}(\mathcal{H}[[\![E]\!].f]); \texttt{havoc}(\mathcal{H}[\overline{\mathcal{P}}]), [\![Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}); \texttt{havoc}(\mathcal{H}[\overline{\mathcal{P}}]); \mathcal{P}[[\![E]\!].f] := \pi, [\![Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}), [\![acc(E.f, \pi) \mathrel{-\!\!*} Q]\!]) \\
\Leftrightarrow\ & wp_{\text{ch}}(\texttt{assume}(\psi < \mathcal{P}), [\![wp_{\text{sl}}(\texttt{inhale}\ acc(E.f, \pi), Q)]\!])
\end{aligned}
$$

Case where $\mathcal{P}[[\![E]\!].f] \neq 0$ follows similiarly.

Corollary 2. *If p is syntactically self-framing, then equiv($\texttt{inhale}\, p$).*

Proof. As we only consider self-framing post-conditions follows trivially from previous lemma.

Remark 1. Without the syntactic self-framing requirement on inhales, it would be unsound to break $\texttt{inhale}\ A * B$ into $\texttt{inhale}\ A; \texttt{inhale}\ B$. In particular, in the Chalice semantics, the behaviour of $\texttt{inhale}(A * B)$ and $\texttt{inhale}(B * A)$ are different. For instance, consider $\texttt{inhale}(x.f = 3 * acc(x.f))$ and $\texttt{inhale}(acc(x.f) * x.f = 3)$.

$$
\begin{aligned}
wp_{\text{ch}}(\texttt{inhale}(x.f = 3 * acc(x.f)), [\![x.f = 3]\!]) &\iff x.f \neq 3 \\
wp_{\text{ch}}(\texttt{inhale}(acc(x.f) * x.f = 3), [\![x.f = 3]\!]) &\iff \mathsf{true}
\end{aligned}
$$

The translation given by Smans *et al.* [15] does not suffer this problem as it does the analogue of `inhale` in a single step. However, it checks self-framing in a similar way, and thus would also rule out the first `inhale`.

In this section, we have shown that the encoding of `inhale` and `exhale` into Boogie2 is equivalent to the separation logic weakest pre-condition semantics. As a consequence, we have shown two things: (1) our model accurately reflects the semantics of Chalice's assertion language, and (2) a fragment of separation logic can be directly encoded into Chalice precisely preserving its semantics.

5 Related Work

In this paper, we have provided a version of separation logic [5, 11], which allows arbitrary expressions over the heap. We have modified the standard presentation of an object-oriented heap for separation logic [12] to separate the notion of access from value. Most previous separation logics have combined these two concepts. One notable exception is the separation logic for reasoning about Cminor [1]. This logic also separates the ability to access memory, the mask, from the actual contents of the heap. The choice in this work was to enable a reuse of a existing operational semantics for Cminor, rather than producing a new operational semantics involving partial states. In the Cminor separation logic, they do not consider the definition of magic wand, or weakest pre-condition semantics, which is crucial for the connection with Chalice [9].

Smans' original presentation of IDF was implemented in a tool, VeriCool [15, 14]. The results in this paper, should also apply to the verification conditions generated by VeriCool. In recent work, Smans *et al.* [16] describe an IDF approach as a separation logic. However, they do not present a model of the assertions, just the VCs of their analog of inhale and exhale. Hence, it does not provide the strong connection between the VCs and the model of separation logic that we have provided.

There have been many other approaches based on dynamic frames [6, 7] to enable automated verification with standard verification tool chains, for instance, Dafny [13] and Region Logic [2]. Like Chalice, both also encoded into Boogie2. The connection between these logics and separation logic is less clear. They explicitly talk about the footprint of an assertion, rather than implicitly. However, our new separation logic might facilitate greater comparison.

6 Extensions and Applications

In this section we highlight the potential impact of our connection between separation logic and the implicit dynamic frames of Chalice, by explaining several ways in which ideas from one world can be transferred to the other.

Supporting Extra Connectives Our extended separation logic supports many more connectives than have previously existed in implicit dynamic frames logics. For example, the support for a "magic wand" in the logic (or indeed an unrestricted logical implication) is a novel contribution, which paves the way for investigating how to extend Chalice to support this much-richer assertion language. While a formal semantics for the magic wand does not immediately tell us how to implement inhaling and exhaling such assertions correctly, it provides us with a means of formally evaluating such a proposal. Furthermore, our direct semantics for the assertion logic of Chalice provides a means of judging whether a particular implementation is faithful to the intended logical semantics.

Evaluating the Chalice Implementation Various design decisions in the Chalice methodology can be evaluated using our formal semantics. For example, Chalice deals with potential interference from other threads by "havocing" heap locations whenever permission to the location is newly granted. An alternative design would be to "havoc" such locations whenever all permission to them was *given up* in an exhale, instead. This would provide different weakest pre-conditions for Chalice commands, and it would be interesting to investigate what differences this design decision makes from a theoretical perspective. Our results provide the necessary basis for such investigations.

Separation logics typically feature recursive (abstract) predicates in their assertion language. The Chalice tool also includes an experimental implementation of recursive predicates (without arguments), along with the use of "functions" in specifications to describe properties of the state in a way which could support information hiding. In the course of investigating how to extend our results to handle predicates in the assertion logics, we discovered that the current approach to handling predicates/functions in Chalice is actually unsound in the presence of functions and the decision to havoc on inhales rather than exhales. We, and the Chalice authors, are now working on a redesign of Chalice predicates based on our findings. As above, the formal semantics and connections we have provided give us excellent tools for evaluating such a redesign.

Implementing Separation Logic One exciting outcome of the results we have presented is that a certain fragment of separation logic specifications can be directly represented in implicit dynamic frames and automatically verified using the Chalice tool. This is a consequence of three results:

1. We have shown that our total heap semantics for separation logic coincides with its prior partial heaps semantics.
2. We have shown that we can replace all "points-to" predicates with logical primitives from implicit dynamic frames, preserving semantics.
3. We have shown that the Chalice weakest-pre-condition calculation agrees with the weakest pre-conditions used in separation logic verification.

The critical aspect which is missing is the treatment of predicates - once we can extend our correspondence results to handle recursively-defined predicates in the logics (which are used in virtually all separation logic verification examples), then it will be possible to exploit our work to use Chalice to implement separation logic verification. This will open up many interesting practical areas of work, in comparing the performance and encodings of verification problems between Chalice and separation logic based tools.

Old Expressions We have also observed that the use of a total heap semantics seems to make it easy to support certain extra specification features in a separation logic assertion language. In particular, the use of "old" expressions in method contracts (allowing post-conditions to explicitly mention values of variables and heap locations in the pre-state of the method call) is awkward to

support in a partial heaps semantics, since it expresses relationships between partial heap fragments which may not have obviously-related domains. As a consequence, separation logic based tools typically do not support this feature. However, with our total heap semantics it seems rather easy to evaluate old expressions by simply replacing our total heap with a copy of the pre-heap. While the details remain to be worked out, this seems to suggest that both separation logic and implicit dynamic frames can be made more expressive using the connections proved in our work.

Acknowledgements

We thank Mike Dodds, David Naumann, Ioannis Kassios, Peter Müller and Sophia Drossopoulou for feedback on drafts of this paper.

References

1. A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In *TPHOLs*, pages 5–21, 2007.
2. A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, pages 387–411, 2008.
3. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
4. J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.
5. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM Press, 2001.
6. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.
7. I. T. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, 2006.
8. K. R. M. Leino. This is Boogie 2. Available from http://research.microsoft.com/en-us/um/people/leino/papers.html.
9. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.
10. P. W. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.
11. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, London, UK, 2001. Springer-Verlag.
12. M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, November 2005.
13. K. Rustan and M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.
14. J. Smans. *Specification and Automatic Verification of Frame Properties for Java-like Programs (Specificatie en automatische verificatie van frame eigenschappen voor Java-achtige programma's)*. PhD thesis, FWO-Vlaanderen, May 2009.
15. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653, pages 148–172, July 2009.
16. J. Smans, B. Jacobs, and F. Piessens. Heap-dependent expressions in separation logic. In *FMOODS/FORTE*, pages 170–185, 2010.

A Some proofs (not in published version)

A.1 Theorem 1: Correctness of Total-Heaps Semantics

For all *SL*-assertions a, environments σ, total heaps H, and permissions maps Π:

$$H, \Pi, \sigma \vDash_{SL^+} a \iff (H \upharpoonright \Pi), \sigma \vDash_{SL} a$$

Proof. By induction on the structure of a.

$a \equiv x.f \overset{\pi}{\mapsto} v$:

$$\begin{aligned} & H, \Pi, \sigma \vDash_{SL^+} a \\ \Leftrightarrow\ & \Pi[\sigma(x), f] \geq \pi \ \textit{and}\ H[\sigma(x), f] = v \\ \Leftrightarrow\ & (x, f) \in \textit{dom}(H \upharpoonright \Pi) \ \textit{and}\ (H \upharpoonright \Pi)[\sigma(x), f] = v \\ \Leftrightarrow\ & (H \upharpoonright \Pi), \sigma \vDash_{SL} a \end{aligned}$$

$a \equiv e_1 = e_2$:

$$\begin{aligned} & H, \Pi, \sigma \vDash_{SL^+} a \\ \Leftrightarrow\ & \sigma(e_1) = \sigma(e_2) \\ \Leftrightarrow\ & (H \upharpoonright \Pi), \sigma \vDash_{SL} a \end{aligned}$$

$a \equiv a_1 * a_2$:

$$\begin{aligned} & H, \Pi, \sigma \vDash_{SL^+} a \\ \Leftrightarrow\ & \exists \Pi_1, \Pi_2.\ \Pi = \Pi_1 * \Pi_2 \ \textit{and} \\ & H, \Pi_1, \sigma \vDash_{SL^+} a_1 \ \textit{and}\ H, \Pi_2, \sigma \vDash_{SL^+} a_2 \\ \Leftrightarrow\ & \exists \Pi_1, \Pi_2.\ \Pi = \Pi_1 * \Pi_2 \ \textit{and} \\ & (H \upharpoonright \Pi_1), \sigma \vDash_{SL} a_1 \ \textit{and}\ (H \upharpoonright \Pi_2), \sigma \vDash_{SL} a_2 \\ \Leftrightarrow\ & (H \upharpoonright \Pi), \sigma \vDash_{SL} a \end{aligned}$$

$a \equiv a_1 \mathrel{-\!\!*} a_2$:

$$\begin{aligned} & H, \Pi, \sigma \vDash_{SL^+} a \\ \Leftrightarrow\ & \forall \Pi_1 \perp \Pi, \forall H_1 \overset{\Pi}{\equiv} H.\ H_1, \Pi_1, \sigma \vDash_{SL^+} a_1 \ \Rightarrow H_1, \Pi * \Pi_1, \sigma \vDash_{SL^+} a_2 \\ \Leftrightarrow\ & \forall \Pi_1 \perp \Pi, \forall H_1 \overset{\Pi}{\equiv} H.\ (H_1 \upharpoonright \Pi_1), \sigma \vDash_{SL} a_1 \ \Rightarrow (H_1 \upharpoonright (\Pi * \Pi_1)), \sigma \vDash_{SL} a_2 \\ \Leftrightarrow\ & \forall \Pi_1 \perp \Pi, \forall H_1 \overset{\Pi}{\equiv} H.\ (H_1 \upharpoonright \Pi_1), \sigma \vDash_{SL} a_1 \ \Rightarrow (H \upharpoonright \Pi * H_1 \upharpoonright \Pi_1), \sigma \vDash_{SL} a_2 \\ \Leftrightarrow\ & \forall H_3 \perp (H \upharpoonright \Pi).\ H_3, \sigma \vDash_{SL} a_1 \ \Rightarrow (H \upharpoonright \Pi) * H_3, \sigma \vDash_{SL} a_2 \\ \Leftrightarrow\ & (H \upharpoonright \Pi), \sigma \vDash_{SL} a \end{aligned}$$

A.2 Lemma 5

Proof. $p \equiv E$:

$$\begin{aligned} & \llbracket wp_{\text{sl}}(\texttt{exhale}\ E, Q) \rrbracket \\ \Longleftrightarrow\ & \llbracket E * Q \rrbracket \\ \Longleftrightarrow\ & \llbracket E \wedge Q \rrbracket \\ \Longleftrightarrow\ & \llbracket E \rrbracket \mathcal{H} \wedge \llbracket Q \rrbracket \\ \Longleftrightarrow\ & wp_{\text{ch}}(\texttt{assert}\ E, \llbracket Q \rrbracket) \\ \Longleftrightarrow\ & wp_{\text{ch}}(\texttt{exhale}\ E, \llbracket Q \rrbracket) \end{aligned}$$

$p \equiv acc(x.f, \pi)$:

$$
\begin{aligned}
&\llbracket wp_{\text{sl}}(\texttt{exhale}\ acc(x.f,\pi), Q)\rrbracket \\
&\iff \llbracket acc(x.f,\pi) * Q\rrbracket \\
&\iff \exists \Pi_1, \Pi_2.\, \Pi_1 = [x, f \mapsto \pi] \wedge \llbracket Q\rrbracket[\Pi_2/\Pi] \wedge \Pi_1 * \Pi_2 = \Pi \\
&\iff \exists \Pi_1, \Pi_2.\, \Pi_1 = [x, f \mapsto \pi] \wedge \llbracket Q\rrbracket H, \Pi_2 \wedge \Pi_2 = \Pi - \Pi_1 \wedge \Pi_1 \subseteq \Pi \\
&\iff \exists \Pi_2.\, \llbracket Q\rrbracket H, \Pi_2 \wedge \Pi_2 = \Pi - [x, f \mapsto \pi] \wedge [x, f \mapsto \pi] \subseteq \Pi \\
&\iff \llbracket Q\rrbracket H, \Pi - [x, f \mapsto \pi] \wedge [x, f \mapsto \pi] \subseteq \Pi \\
&\iff wp_{\text{ch}}(\texttt{assert}\ \Pi[x,f] \geq \pi, \llbracket Q\rrbracket H, \Pi - [x, f \mapsto \pi]) \\
&\iff wp_{\text{ch}}(\texttt{assert}\ \Pi[x,f] \geq \pi; \Pi[x,f] := \Pi[x,f] - \pi, \llbracket Q\rrbracket) \\
&\iff wp_{\text{sl}}(\texttt{exhale}\ acc(x.f,\pi), \llbracket Q\rrbracket)
\end{aligned}
$$

$p \equiv p_1 * p_2$:

$$
\begin{aligned}
&\llbracket wp_{\text{sl}}(\texttt{exhale}\ p_1 * p_2, Q)\rrbracket \\
&\iff \llbracket p_1 * p_2 * Q\rrbracket && \text{(by definition of wp)} \\
&\iff \llbracket wp_{\text{sl}}(\texttt{exhale}\ p_1, p_2 * Q)\rrbracket && \text{(by definition of wp)} \\
&\iff \llbracket wp_{\text{sl}}(\texttt{exhale}\ p_1, wp_{\text{sl}}(\texttt{exhale}\ p_2, Q))\rrbracket && \text{(by definition of wp)} \\
&\iff wp_{\text{ch}}(\texttt{exhale}\ p_1, \llbracket wp_{\text{sl}}(\texttt{exhale}\ p_2, Q)\rrbracket) && \text{(by IH)} \\
&\iff wp_{\text{ch}}(\texttt{exhale}\ p_1; \texttt{exhale}\ p_2, \llbracket Q\rrbracket) && \text{(by IH)} \\
&\iff wp_{\text{ch}}(\texttt{exhale}\ p_1 * p_2, \llbracket Q\rrbracket) && \text{(by definition of wp)}
\end{aligned}
$$