Logical Methods in Computer Science
Vol. 8 (3:01) 2012, pp. 1–54
www.lmcs-online.org

Submitted Oct. 20, 2011
Published Jul. 31, 2012

THE RELATIONSHIP BETWEEN SEPARATION LOGIC AND IMPLICIT DYNAMIC FRAMES

MATTHEW J. PARKINSON [a] AND ALEXANDER J. SUMMERS [b]

[a] Microsoft Research Cambridge
e-mail address: mattpark@microsoft.com

[b] ETH Zurich
e-mail address: 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 precisely connect the semantics of these two logics. We define a logic whose syntax subsumes both that of a standard separation logic, and that of implicit dynamic frames as sub-syntaxes. We define a total heap semantics for our logic, and, for the separation logic subsyntax, prove it equivalent the standard partial heaps model. In order to define a semantics which works uniformly for both subsyntaxes, we define the novel concept of a *minimal state extension*, which provides a different (but equivalent) definition of the semantics of separation logic implication and magic wand connectives, while also giving a suitable semantics for these connectives in implicit dynamic frames. We show that our resulting semantics agrees with the existing definition of weakest pre-condition semantics for the implicit dynamic frames fragment. Finally, we show that we can encode the separation logic fragment of our logic into the implicit dynamic frames fragment, preserving semantics. For the connectives typically supported by tools, this shows that separation logic can be faithfully encoded in a first-order automatic verification tool (Chalice).

1. Introduction

Separation logic (SL) [6, 12] 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) [18] 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

1998 ACM Subject Classification: F.3.1.

Key words and phrases: Separation Logic, Implicit Dynamic Frames, Verification Logics, Partial Heaps, Permissions, Concurrency.

LOGICAL METHODS IN COMPUTER SCIENCE

DOI:10.2168/LMCS-8 (3:01) 2012

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 investigate the formal relationship between the two approaches. As a medium for this comparison, we define a verification logic (which we name *Total Heaps Permission Logic*) whose syntax includes both that of a typical separation logic, and that of implicit dynamic frames. We define a semantics for Total Heaps Permission Logic based on states which incorporate a *total heap* and a separate *permission mask*, that we show both captures the original semantics of separation logic, and correctly captures the semantics of IDF. Intuitively, 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, and this complicates the definition of the separation logic "magic wand" and implication connectives. In order to define a suitable semantics for these connectives which is compatible with both approaches, we introduce the novel concept of *minimal extensions* of a state, and use this to define a novel semantics for these connectives, which nonetheless agrees with the original semantics for the separation logic fragment of our logic. Correctly reflecting the standard semantics of the separating conjunction and magic wand allows us to use these connectives to define the usual separation logic notion of weakest pre-conditions of commands.

In order to show that our logic correctly captures the semantics of the IDF formulas, we focus on the form of IDF found in the concurrent verification tool Chalice [10]. 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 [9] encoding and the VCs generated from the separation logic proof rules are logically equivalent. This shows that our model directly captures the existing semantics of IDF.

We make use of these strong correspondences to define an encoding of separation logic into implicit dynamic frames that preserves semantics. We then define a subsyntax of separation logic (corresponding to the logical connectives supported by many practical tools), which maps onto the assertion language supported by Chalice, and show that this fragment of separation logic can, via our correspondences, be handled in a purely first-order prover.

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 provide an overview of the challenges in defining our logic and semantics, and present Total Heaps Permission Logic (§3), characterising various properties of our total heap semantics. We prove the correspondence between VCs as calculated in separation logic and in implicit dynamic frames (§4), and then combine our proven results to show how to map a fragment of separation logic into contracts which can be verified by the Chalice tool, preserving their original semantics (§5). Finally, we discuss related work (§6), consider possible extensions and conclude (§7).

The contributions of this paper are as follows:

- We define a total heaps semantics for a logic whose syntax subsumes a separation logic, and prove that, for the separation logic fragment, our total heaps semantics is equivalent with the standard (partial heaps) semantics for the separation 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.
- We present the notion of *minimal extensions* of a state, and show how it can be used to define the semantics of the separation logic implication and magic wand connectives in a new way.

Extensions with regard to the conference version. This paper extends the conference version [15] by providing a different definition of implication that corresponds to that used in Chalice. The conference version provided a definition of implication that was correct with respect to separation logic, but on the formulas used in Chalice it had undesirable behaviour. We have altered the definitions of implication and magic wand to correctly model both Chalice and separation logic.

The paper provides extended discussions of the design of the logic, detailing the requirements which come from each of our target logics. We explicitly define and discuss the various notions of *state extension* which were used implicitly in the formulations of the technical definitions in our precursor paper. The semantics of implication in the logic is discussed in detail, and a new concept of *minimal extensions* is used to obtain a semantics which works well for both target logics. The resulting semantics is formulated differently from the traditional presentation of implication in intuitionistic separation logic; our definition requires checking the subformulas in fewer states.

The syntactic condition on when a Chalice assertion was considered self-framing in the conference paper was overly restrictive, in that it did not reflect that Chalice takes account of the restrictions provided by an assertion: for instance, $acc(x.f, 1) * y = x * y.f = 5$ would not have been considered self-framing in the conference version, but is in this paper, and in Chalice.

We provide a new section (§5) which shows how to combine the previously-proved results to explicitly show that a fragment of separation logic can be equivalently verified using separation logic weakest pre-conditions, or (via an encoding) using implicit dynamic frames specifications and weakest pre-condition calculations.

Finally, we provide full details of all proofs.

2. Background and Motivation

2.1. **Standard Separation Logic.** Separation logic [6, 12] 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♯ [14].

In order to handle concurrency, separation logic has been extended to consider its basic points-to assertions as *permissions* [11], determining which thread is allowed to read and write the corresponding state. To gain flexibility, *fractional permissions* [5, 4] were introduced, allowing the permissions governed by points-to assertions to be split and recombined. 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 2.1 (Separation Logic Assertions (*SL*))**.** We assume a set of *object identifiers*[1], 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) are defined as follows[2]. In this definition, n ranges over integer constants, and $0 < \pi \leq 1$.

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

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 $a_1 * a_2$ expresses that a_1 and a_2 are true and depend on disjoint fragments of the heap. The magic wand $a_1 \mathrel{-\!\!*} a_2$ expresses that if any extra partial heap satisfying a_1 is combined with the current partial heap, then the resulting heap is guaranteed to satisfy a_2.

Fractional permissions[3] [4, 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 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 2.2 (Partial Fractional Heaps [4])**.**

- 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 $(\iota, f) \in dom(h)$.

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

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

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

- Partial heap extension: $h_1 \subseteq h_2$, iff $\forall(\iota, f) \in dom(h_1).\ \downarrow_1(h_1[\iota, f]) = \downarrow_1(h_2[\iota, f])$ and $\downarrow_2(h_1[\iota, f]) \leq \downarrow_2(h_2[\iota, f])$.
- Partial heap compatibility: $h_1 \perp h_2$ 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 nth component of a tuple.

There are two main flavours of separation logic studied in the literature: *classical* separation logic, and *intuitionistic* separation logic [6]. 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 [14].

Definition 2.3 (Standard Semantics for *SL* [4])**.** Environments σ are partial functions[4] from variable names to values. Separation logic expression semantics, $[\![e]\!]_\sigma$ are defined by $[\![x]\!]_\sigma = \sigma(x)$, $[\![n]\!]_\sigma = n$ and $[\![\texttt{null}]\!]_\sigma = \texttt{null}$. The semantics of assertions is then as follows:

$$\begin{array}{lll}
h, \sigma \vDash_{SL} e_1.f \overset{\pi}{\mapsto} e_2 & \iff & \downarrow_2(h[[\![e_1]\!]_\sigma, f]) \geq \pi \ \wedge\ \downarrow_1(h[[\![e_1]\!]_\sigma, f]) = [\![e_2]\!]_\sigma \\
h, \sigma \vDash_{SL} e = e' & \iff & [\![e]\!]_\sigma = [\![e']\!]_\sigma \\
h, \sigma \vDash_{SL} a_1 * a_2 & \iff & \exists h_1, h_2.(\ h = h_1 * h_2 \ \wedge\ h_1, \sigma \vDash_{SL} a_1 \ \wedge\ h_2, \sigma \vDash_{SL} a_2) \\
h, \sigma \vDash_{SL} a_1 \mathrel{-\!\!*} a_2 & \iff & \forall h'.(\ h' \perp h \ \wedge\ h', \sigma \vDash_{SL} a_1 \ \Rightarrow\ h{*}h', \sigma \vDash_{SL} a_2) \\
h, \sigma \vDash_{SL} a_1 \wedge a_2 & \iff & h, \sigma \vDash_{SL} a_1 \ \wedge\ h, \sigma \vDash_{SL} a_2 \\
h, \sigma \vDash_{SL} a_1 \vee a_2 & \iff & h, \sigma \vDash_{SL} a_1 \ \vee\ h, \sigma \vDash_{SL} a_2 \\
h, \sigma \vDash_{SL} a_1 \to a_2 & \iff & \forall h'.(\ h' \perp h \ \wedge\ h{*}h', \sigma \vDash_{SL} a_1 \ \Rightarrow\ h{*}h', \sigma \vDash_{SL} a_2) \\
h, \sigma \vDash_{SL} \exists x.\ a & \iff & \exists v.(\ h, \sigma[x \mapsto v] \vDash_{SL} a)
\end{array}$$

The semantics for the separating conjunction and magic wand express the required splitting and combination of partial heaps. The semantics for logical implication $\to$ considers all possible extensions of the current heap, so that assertion truth is closed under heap extension [6]. In examples, we will sometimes write $e.f \overset{\pi}{\mapsto} \_$ as a shorthand for $\exists x.\ e.f \overset{\pi}{\mapsto} x$.

[4]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.

2.1.1. *Assume/Assert.* Verification in Boogie2 [9] and related technologies uses two commands commonly to encode verification: `assume` A_1 and `assert` A_1. The first allows the verification to work forwards with the additional assumption of A_1, while the second requires A_1 to hold otherwise it will be considered a fault. These can be given weakest precondition semantics of:

$$wp(\texttt{assert}\ A_1, A_2) = A_1 \wedge A_2 \qquad wp(\texttt{assume}\ A_1, A_2) = A_1 \Rightarrow A_2$$

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 can be 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:

$$wp(\texttt{assert}^*\ A_1,\ A_2) = A_1 * A_2 \qquad wp(\texttt{assume}^*\ A_1,\ A_2) = A_1 -\!\!* A_2$$

These commands can be understood as follows: `assert`* A_1 removes a heap fragment satisfying A_1, and `assume`* A_1 adds a heap fragment satisfying A_1. 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 can be 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: `exhale` A_1 and `inhale` A_1, which are also given an intuitive semantics of removing and adding access to state. One outcome of this paper is to make this intuitive connection between `exhale`/`inhale` and `assert`*/`assume`* 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 [8, 7]. 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 [18, 17] 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 [10] 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 deadlocks. 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 intermediate auxiliary Chalice commands `exhale` p and `inhale` p are used. These commands reflect the removal and addition of permissions from the state, as well as expressing assertions and assumptions about heap values. 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 2.4 (Our Chalice Subsyntax)**.** Expressions E, boolean expressions B and assertions p in our fragment of Chalice are given by the following syntax definitions:

$$\begin{array}{rcl} E & ::= & x \mid n \mid \texttt{null} \mid E.f \\ B & ::= & E = E \mid E \neq E \mid B * B \\ p & ::= & B \mid acc(E.f, \pi) \mid p * p \mid B \rightarrow p \end{array}$$

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, $acc(x.f, 1/2) \wedge acc(x.f, 1/2)$ is equivalent to $acc(x.f, 1)$, while $acc(x.f, 1) \wedge acc(x.f, 1)$ is actually equivalent to falsity (it describes a state in which we have more than the full permission to the location $x.f$). 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 or first order logic.

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[5]. The latter maps object-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. This is enforced by requiring that assertions used in verification contracts are *self-framing* [7] - which means that the assertion includes enough accessibility predicates to "frame" its heap expressions. For example, the assertion

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

$x.f = 5$ 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. Total Heaps Permission Logic (*TPL*)

3.1. **Race-free Assertions.** In order for a static verification tool to be able to reason soundly about concurrent programs, a crucial aspect is to be able to give a well-defined semantics to the assertion language employed. Since other executing threads may interfere with the execution of code being verified (for example, by writing to heap locations which the current program also accesses; a data race), assertions which describe properties of the heap do not, in general, even have a well-defined semantics. For example, consider the simple assertion $x.f > 5$. Such an assertion only has a well-defined semantics (at verification time) if, at runtime, the heap location $x.f$ is guaranteed not to be subject to a data race. If another thread writes to this location at the "same time" as the assertion is checked to hold, the truth of the assertion becomes non-deterministic, depending on the interleaving of the memory accesses by the two threads. This makes any reasoning about expressions such as $x.f$ as expressions in a logical sense unreliable: assertions such as $x.f > x.f$ could even be "true" at runtime, due to interference; a behaviour which any useful verifier will struggle to mimic accurately.

For these reasons, verifiers for concurrent programs need to use a verification methodology and assertion language whose semantics avoids data races. Both separation logic and implicit dynamic frames attack this problem by employing notions of (fractional) *permissions*. Permissions permit access to particular heap locations, and can be passed around between threads, with the crucial property that a thread is only allowed to write to a heap location if no other thread holds a permission to the location. By imposing suitable restrictions on the assertion language used for verification, one can then guarantee a data-race-free semantics by passing permissions explicitly along with heap-dependent assertions, and enforcing the policy that an assertion may only mention a heap location if it also carries at least some permission to that location. In implicit dynamic frames, these permissions are represented by "accessibility predicates" $acc(E.f, \pi)$, denoting π permission to location $E.f$. For example, while the assertion $x.f > 5$ does not have a well-defined semantics on its own, the compound assertion $acc(x.f, \pi) * x.f > 5$ does - the presence of the permission to the location $x.f$ guarantees that its value in the heap is robust to interference. More generally, any heap-dependent expression in an assertion can only be given a meaning by its value being fixed with a permission to the appropriate heap locations. A "self-framing" assertion is one which is only satisfied in states which carry enough permissions to fix the values of all heap locations on which it depends; not all assertions are self-framing ($x.f > 5$ is not), but only such assertions can generally be used for verification contracts. The fact that, in implicit dynamic frames, the permission to access a heap location can come from a different part of an assertion (e.g., conjunct) than the constraints on the value at that location, is the main challenge in giving a correct semantics for the logic.

The same challenge does not arise in separation logic, which does not allow heap-dependent expressions, instead providing the special "points-to" predicates as the sole way of handling heap accesses. A "points to" predicate $e_1.f \overset{\pi}{\mapsto} e_2$ plays a dual role in the logic - it provides knowledge of the value e_2 of the heap location $e_1.f$, and it also provides a permission π to this location, making the value robust to interference. Because there

is no other way to refer directly to heap locations, one cannot ever talk about the value of a location without having some permission to that location. The observation of this dual role leads naturally to the idea of encoding separation logic assertions into implicit dynamic frames by replacing every points-to predicate $e_1.f \overset{\pi}{\mapsto} e_2$ by a permission and a heap-dependent expression: $acc(e_1.f, \pi) * e_1.f = e_2$. This observation can be used as the basis of a comparison and translation between the two logics. In fact, our approach is to give a uniform semantics for a logic which subsumes both separation logic and implicit dynamic frames constructs, and then show that the primitive constructs of the former can be represented in the latter.

3.2. **Overview of Our Approach.** In order to formally relate the two paradigms of separation logic and implicit dynamic frames, we define a new logic which subsumes both syntaxes. We call this logic Total Heaps Permission Logic (*TPL*). This logic includes as primitives both the "points-to" predicates of separation logic, and the "accessibility predicates" of implicit dynamic frames, along with an expression syntax which permits heap-dependent expressions. As we will show formally in this paper, this is actually redundant; one can encode the SL-style primitives into implicit dynamic frames. However, our *TPL* serves as a uniform basis for comparing these two logics. We also include all of the common connectives used in separation logic, which subsume those typically implemented in tools (based on either approach).

Our approach is to define a semantics for *TPL*, based on states consisting of a stack (giving meaning to variables), a *total heap*, and a *permissions mask* (defining which locations in the total heap have reliable values for the current thread). Because our semantics is defined compositionally, it actually gives a meaning to assertions which are not (by themselves) well-formed in all states. As discussed above, assertions which mention heap-dependent expressions such as $x.f > 5$ are not necessarily well-defined when considered in isolation. However, because the IDF approach allows for such assertions as *subformulas* of a well-formed assertion, and because we want to define a compositional semantics for our logic, we are obliged to give such assertions a semantics, even though (by themselves) they cannot be used in either approach. In some sense, by encompassing both SL and IDF, we actually make our assertion logic *too* general. The presence of ill-formed assertions in our general logic means that (just as in implicit dynamic frames) we will later have to introduce additional concepts such as *self-framing assertions*, in order to identify the fragments of our logic which are well-behaved.

Definition 3.1 (Total Heaps Permission Logic)**.** We define the expressions E and assertions A of *Total Heaps Permission Logic* (*TPL*), by the following grammar (in which n stands for any integer constant):

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

Note that the syntax of separation logic assertions (ranged over by a; see Definition 2.1) is a strict subset of the *TPL* assertions A defined above. The syntax of separation logic expressions e is also a strict subset of *TPL* expressions E. Similarly, the syntax of Chalice assertions (cf. Definition 2.4) is a subset of our *TPL* syntax.

Our strategy for the rest of the paper is as follows. We will first investigate carefully how to define a suitable total-heaps-based semantics for *TPL*. In particular, we spend considerable attention on the definition of cases which correspond to modelling state extension (the implication and magic wand connectives).

We will then show that, for the subsyntax which corresponds to separation logic assertions, our total heaps semantics coincides with the traditional partial-heaps-based semantics of the logic (cf. Definition 2.3). Thus, we define a total-heaps model for separation logic, which is consistent with the standard model. We will further show that the subsyntax of *TPL* which covers SL assertions can be mapped into the IDF subsyntax, preserving the semantics of the assertions. Thus, we can faithfully map from the SL world to an IDF-based assertion language.

In Section 4, we will show how to connect our *TPL* model to the Chalice verification methodology. In particular, we will show that weakest pre-conditions as calculated by Chalice for a first-order theorem prover, are equivalent to weakest pre-conditions as calculated in separation logic. By combining this result with our ability to faithfully reflect traditional SL semantics in *TPL*, we can show the equivalence of the overall approaches. In particular, for the subsyntax of SL typically supported by automatic tools, we can show that we can encode programs with SL specifications as programs with IDF specifications, and compute equivalent weakest pre-conditions for direct verification by a theorem prover.

3.3. **Total vs Partial Heaps.** One important technical challenge faced in defining a semantics for both logics, is that the semantics of separation logic is defined using *partial heaps* (representing heap fragments, which can be split and recombined), while the implementation of implicit dynamic frames employs a mutable *total heap*, and a separate *permissions mask* to keep track of the permissions held in the current state. In order to make a uniform semantics for the two logics, we needed to bridge this gap between the two paradigms. We achieve this by employing only total heaps and permission masks, and using these to define a semantics that faithfully captures the traditional partial-heaps-based model for the SL subsyntax.

Definition 3.2 (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 $H[o, f]$. We write *field location* to mean a pair of object-identifier and field-identifier.
A *permission mask* P is a total map from pairs of object-identifier and field-identifier to permissions. Permission lookup is written $P[o, f]$.
We write $P_1 \subseteq P_2$ for *permission mask extension*, i.e., $\forall(o, f).\ P_1[o, f] \leq P_2[o, f]$.
We write $\varnothing$ for the *empty permission mask*; i.e., the mask which assigns 0 to all locations.
We write $rds(P)$ for the set of field locations with non-zero permissions in P, that is, $\{(o, f) \mid P[o, f] > 0\}$. We write $\overline{rds(P)}$ for the complement of this set of locations.
A *state* is a triple (H, P, σ) consisting of a heap, a permission mask and an environment σ.
Two permission masks P_1 and P_2 are *compatible*, written $P_1 \perp P_2$, if it holds that:

$$\forall(o, f).\ P_1[o, f] + P_2[o, f] \leq 1$$

The *combination* of two permission masks, written $P_1 * P_2$ is undefined if P_1 and P_2 are not compatible, and is otherwise defined pointwise to be the following permission mask:

$$(P_1 * P_2)[o, f] = P_1[o, f] + P_2[o, f]$$

We define the greatest lower bound of two masks:

$$(P_1 \sqcap P_2)[o, f] = \mathsf{min}(P_1[o, f], P_2[o, f])$$

and the least upper bound of two masks:

$$(P_1 \sqcup P_2)[o, f] = \mathsf{max}(P_1[o, f], P_2[o, f])$$

Finally, we define a partial operation of subtraction on permission masks, $P_1 - P_2$. It is defined if and only if $P_2 \subseteq P_1$, and is defined by:

$$(P_1 - P_2)[o, f] = (P_1[o, f] - P_2[o, f])$$

The crucial observation relating our logic to SL is that, while we will use total heaps in our semantics, we will actually only allow assertions to depend on a subheap; those locations which the current thread can "read"; i.e., that it has at least some permission to. We employ the notation $rds(P)$ (where P is a permission mask), to talk about this set of locations. We will design our semantics such that, for all separation logic assertions, their semantics in a state with heap H and permissions P corresponds to their traditional semantics using the (partial) heap obtained by *restricting* H to just the domain $rds(P)$ (this restriction is formally written as $H{\restriction}P$ later). This idea reflects the intuition that all other locations in the (total) heap H have unreliable values (which may be subject to interference from other threads); only assertions which are appropriately "framed" by sufficient permissions, can be relied upon in a concurrent setting.

When we want to explicitly express that an assertion is robust to interference from other threads, we can do so by considering the evaluation of the assertion in all heaps which agree with the current one on the locations which the current thread can read, according to the permissions mask. Effectively, we introduce a "havoc" of all of the locations to which we hold no permission, and check that the assertion is still guaranteed after all such locations are assigned arbitrary values. In order to define this operation, we introduce the concept of two heaps "agreeing" on the permissions in a mask (as well some other heap constructions) as follows:

Definition 3.3 (Total Heap Operations)**.** Two heaps H_1 and H_2 *agree on a set of object field locations* F, written $H_1 \overset{F}{\equiv} H_2$, if the two heaps contain the same value for each location, i.e.,

$$H_1 \overset{F}{\equiv} H_2 \iff \forall (o, f) \in F.\ H_1[o, f] = H_2[o, f]$$

Two heaps H_1 and H_2 *agree on permissions* P, written $H_1 \overset{P}{\equiv} H_2$, if the two heaps agree on all field locations given non-zero permission by P, i.e.,

$$H_1 \overset{P}{\equiv} H_2 \iff H_1 \overset{rds(P)}{\equiv} H_2$$

The *restriction of* H *to* P, written $H{\restriction}P$ is a *partial fractional heap* (Definition 2.2), defined by:

$$dom(H{\restriction}P) = rds(P)$$
$$\forall (o, f) \in dom(H{\restriction}P).\ (H{\restriction}P)[o, f] = (H[o, f], P[o, f])$$

The *conditional merge of* H_1 *and* H_2 *over a set of locations* F, written $(F\ ?\ H_1 : H_2)$ is a total heap defined by:

$$(F\ ?\ H_1 : H_2)[o, f] = \begin{cases} H_1[o, f] & \text{if } (o, f) \in F \\ H_2[o, f] & \text{otherwise} \end{cases}$$

We write $(P\ ?\ H_1 : H_2)$ as a shorthand for $(\mathit{rds}(P)\ ?\ H_1 : H_2)$.

We can make use of these operations on total heaps to define what it means for an assertion to be *stable* in a certain state (which intuitively means that its truth only depends on heap locations to which it also requires permission to be held).

Definition 3.4 (Interference, Stability, Self-Framing and Pure Assertions)**.** Given a heap H and a permissions mask P, the *interfered heaps from* H, P is a set of heaps defined by:

$$\mathsf{interfere}(H, P) = \{H' \mid H' \overset{P}{\equiv} H\}$$

A set of states S is *stable with extra permissions* P, written as $\mathsf{stable\text{–}with}_P\,(S)$, if the extra permissions are sufficient to make the set closed under interference; i.e.,

$$\mathsf{stable\text{–}with}_P\,(S) \Leftrightarrow (\forall (H, P', \sigma) \in S.\ \forall H' \in \mathsf{interfere}(H, P * P').(H', P', \sigma) \in S)$$

A set of states S is *stable*, written as $\mathsf{stable}(S)$, if the set is closed under interference; i.e.,

$$\mathsf{stable}(S) \Leftrightarrow \mathsf{stable\text{–}with}_{\varnothing}\,(S)$$

We write $\langle\!\langle A \rangle\!\rangle$ to denote the set of states in which the assertion A is true (the actual definition of our semantic judgement $H, P, \sigma \vDash_{\mathit{TPL}} A$ will come later).

$$\langle\!\langle A \rangle\!\rangle = \{(H, P, \sigma) \mid H, P, \sigma \vDash_{\mathit{TPL}} A\}$$

An assertion A is *self-framing* if and only if the set of states satisfying it is stable; i.e., if $\mathsf{stable}(\langle\!\langle A \rangle\!\rangle)$ is true.

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

$$\forall H, P, \sigma.\ ((H, P, \sigma) \in \langle\!\langle A \rangle\!\rangle \Rightarrow (H, \varnothing, \sigma) \in \langle\!\langle A \rangle\!\rangle)$$

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. This is shown as a corollary (Corollary 3.21) of the main theorem in this section.

On the other hand, pure assertions which depend on heap values (such pure assertions are not supported in separation logic, but are employed in implicit dynamic frames) are naturally not self-framing. An assertion such as $x.f = 5$ is considered pure (it does not mention any permissions or points-to predicates); it will be true in a state where we have no permissions, but in which the value of the heap location $x.f$ is 5. Nonetheless, such a state is not stable; when we allow for interference, the value of $x.f$ can be modified, and the truth of the assertion need not be preserved.

3.4. **Pure Assertions and Separating Conjunction.** Our assertion language includes the *separating conjunction* $A * B$ of separation logic (recall Definition 2.3), and permissions can be distributed multiplicatively across this conjunction. In particular, our semantics needs to enforce, for an assertion $A * B$, that the permissions required are the *sum* of the permissions required in each of A and B; we can model this by checking that we can split the permissions mask into two parts, using them to judge the respective conjuncts. A question which then arises is, what should happen to the heap when judging the separating conjunction? In the traditional separation logic semantics, which uses partial heaps, since the heap values and "permissions" are both tracked together in partial heap chunks, it is natural to divide up the partial heap in this case. In the case of partial fractional heaps, the

two resulting heap chunks can still share values, but only to those locations in which both parts hold some permission. With a total-heaps semantics, we have a choice as to how to reflect this "splitting"; we can either only split the permissions mask across the conjuncts, but leave the heap unchanged, or we can also try to simulate the splitting of heap values, say, by throwing away information about certain heap locations when judging the individual conjuncts. By looking only at the separation logic fragment of our logic, we cannot see a clear advantage either way; since all assertions in that logic can only read the same heap values that they provide permission to, the question of what should be done with other heap values is irrelevant there (and a partial heaps model gives no reasonable way to even phrase this decision, since it conflates the notions of permission to read a location, and the action of actually reading it). However, this question is pertinent in the case of implicit dynamic frames, where we have heap-dependent expressions which can occur in pure assertions.

For pure assertions, we want the property that, even when mentioned in a separating conjunction, they do not actually extend the "heap footprint" of what the assertion requires. In particular, we would like to retain the law (which holds in intuitionistic separation logic), that $A_1 * A_2$ is equivalent to $A_1 \wedge A_2$ when either of the two conjuncts are pure. In particular, this motivates that pure assertions should be allowed to depend on the same state as assertions they are conjoined with. Of course, in separation logic, where pure assertions are syntactically restricted to not mention the heap, this "same state" just means the environment σ. But in implicit dynamic frames, we would like heap-dependent pure assertions to be allowed to depend on the same heap values that other conjuncts make readable by providing permissions; when interpreting assertions such as $acc(x.f) * x.f == 5$ this is exactly what we want.

For these reasons, in our total-heaps model, we define the semantics for separating conjunction with a split of the permissions mask, but no change to the heap. This is concretely achieved by checking that we can split P into two pieces, each of which are sufficient to judge the two sub-formulas; the particular definition (which will be provided as part of the definition of our full semantics later) is:

$$H, P, \sigma \vDash_{TPL} A_1 * A_2 \iff$$
$$\exists P_1, P_2.\ (P = P_1 * P_2 \ \wedge\ H, P_1, \sigma \vDash_{TPL} A_1 \ \wedge\ H, P_2, \sigma \vDash_{TPL} A_2)$$

In the case of some implicit dynamic frames assertions, this rule for treating separating conjunction may "separate" a heap-dependent expression from the permission used to fix its values. For example, consider a permissions mask P in which we have full permission to the location $x.f$ (and no other permissions), and a heap H in which $x.f$ has the value 5. The assertion $acc(x.f, 1) * x.f = 5$ is true in such a state. But, in treating the separating conjunction, we are forced to split $P = P_1 * P_2$ and put *all* permission to $x.f$ into P_1, in order to satisfy the left conjunct, leaving P_2 to be the empty permissions mask. The fact that the sub-formula $x.f = 5$ is eventually judged in a state in which we hold no permissions to the relevant location $x.f$ is not a problem - we only need to be sure that permission to this location is held *somewhere* in the whole assertion, and not in this particular sub-formula. That is, the property of an assertion being well-formed (self-framing) is not enforced for its sub-formulas, but only for the assertion as a whole.

3.5. **Modelling Partial Heap Extensions.** One of the most difficult technical challenges in the design of our semantics was correctly handling the *magic wand* ($-\!\!*$) and *implication* ($\rightarrow$) connectives. In the traditional partial heaps semantics of separation logic (Definition

2.3), the semantics of both of these connectives involve considering *extensions* of the current heap. In a semantics based on partial heaps, this is rather straightforward, but with total heaps and permission masks it is not so obvious how to model "heap extension" for these connectives.

One simple option is just to consider permission extension - leave the heap unchanged but consider all larger permissions masks. The problem with this rather-simplistic proposal is that it attaches significance to pre-existing values in our total heap even in the case where we previously had no permission to them. Since such values are generally meaningless, this doesn't give a well-behaved semantics. When one compares with the operation of extending partial heaps, which we are trying to simulate appropriately, it can be seen that the approach doesn't work; when a new heap location is added in a partial heaps model, it can take any value, whereas our total heap only has one value at any one time.

In order to avoid tying ourselves down to the values in our total heaps which are not necessarily currently meaningful, we can instead model heap extension by adding on extra permission and then *havocing* (i.e., assigning arbitrary values to) the heap locations to which we have newly acquired permission. In this way, we make the original heap values stored at these locations irrelevant, and correctly reflect the general operation of adding on a fresh heap location in a partial-heaps-based model. To this end, we define several variants of this idea of how to model state extensions. The differences in the variants come from two decisions. Firstly, when we add on new permission, do we havoc the heap values at all locations to which we previously held no permission (we call this a *global havoc*), or only those which the permissions newly allow us to read (we call this a *local havoc*)? Secondly, when we define the extensions of a state, are we interested in resulting states in which we combine the new permissions with those we held previously, or do we just want to describe the "extra" disjoint part of the state (using the new permissions, but not the old ones)? These questions give rise to the following four concepts:

Definition 3.5 (State Extensions)**.** The set of *locally-havoced extensions* of a state (H, P, σ) is the set of states in which extra permission is added, and possibly-new values are assigned to the newly-readable locations, i.e.,:

$$\mathsf{localExts}(H, P, \sigma) = \{(H', P * P', \sigma) \mid P' \perp P \wedge H' \overset{rds(P) \cup \overline{rds(P')}}{\equiv} H\}$$

The set of *globally-havoced extensions* of a state (H, P, σ) is the set of states in which extra permission is added, and possibly-new values are assigned to the previously-unreadable locations, i.e.,:

$$\mathsf{globalExts}(H, P, \sigma) = \{(H', P * P', \sigma) \mid P' \perp P \wedge H' \in \mathsf{interfere}(H, P)\}$$

The set of *locally-havoced disjoint extensions* of a state (H, P, σ) is the set of states in which extra permission is added, possibly-new values are assigned to the newly-readable locations, and only the extra permissions are kept in the results, i.e.,:

$$\mathsf{localDisjExts}(H, P, \sigma) = \{(H', P', \sigma) \mid P' \perp P \wedge H' \overset{rds(P) \cup \overline{rds(P')}}{\equiv} H\}$$

The set of *globally-havoced disjoint extensions* of a state (H, P, σ) is the set of states in which extra permission is added, possibly-new values are assigned to the previously-unreadable locations, and only the extra permissions are kept in the results,

$$\mathsf{globalDisjExts}(H, P, \sigma) = \{(H', P', \sigma) \mid P' \perp P \wedge H' \in \mathsf{interfere}(H, P)\}$$

As we will see later in this section, we have uses for all of these notions of state extension, and choosing the appropriate one at various points is important for our logic to have the right semantics.

3.6. **Minimal Extensions and Implication.** We now consider how to design an appropriate semantics for implication in our logic, which should manage to work appropriately both for the separation logic and implicit dynamic frames fragments. Recall that, in the traditional semantics of (intuitionistic) separation logic (cf. Definition 2.3), an implicative assertion $a_1 \to a_2$ is true if, in all extensions of the current heap, whenever a_1 is true then a_2 is also true. Therefore, we need to be careful to appropriately model this idea of extending the current state when judging an implication. We use three examples here to guide the discussion of our design:

$$\begin{array}{ll} \textit{Ex. 1:} & x.f \overset{1}{\mapsto} \_ * (x.f \overset{1}{\mapsto} 5 \ \to \ y.g \overset{1}{\mapsto} \_) \\ \textit{Ex. 2:} & acc(x.f, 1) * ((acc(x.f, 1) * x.f = 5) \ \to \ acc(y.g, 1)) \\ \textit{Ex. 3:} & acc(x.f, 1) * (x.f = 5 \to acc(y.g, 1)) \end{array}$$

In (intuitionistic) separation logic, the first formula is actually only true in states which have (full) permission to *both* locations $x.f$ and $y.g$. The reason is that, in judging the implication subformula, we have to consider all extensions of the provided state. Unless the state in which we judge the implication has at least some permission to $x.f$ (and gives a value other than 5 to this location), then when we consider all extensions of the heap we must consider the possibility that the new heap stores a value 5 at this location. Since the left-hand conjunct $x.f \overset{1}{\mapsto} \_$ requires full access to $x.f$, no permission to this location can be left over when judging the implication on the right. The formula can be formally shown to be equivalent to $x.f \overset{1}{\mapsto} \_ * y.g \overset{1}{\mapsto} \_$ according to the standard semantics of Definition 2.3, as follows:

Proof. It suffices to show

$$h, \sigma \vDash_{SL} (x.f \overset{1}{\mapsto} 5 \ \to \ y.g \overset{1}{\mapsto} \_) \wedge ([\![x]\!]_\sigma, f) \notin dom(h) \Rightarrow \downarrow_1(h[[\![y]\!]_\sigma, f]) = 1$$

We can prove this by contradiction. We assume $([\![x]\!]_\sigma, f) \notin dom(h)$ and $\downarrow_1(h[[\![y]\!]_\sigma, f]) \neq 1$, and consider the semantics of the implication:

$$\forall h'.(\ h' \perp h \ \wedge \ h{*}h', \sigma \vDash_{SL} x.f \overset{1}{\mapsto} 5 \ \Rightarrow \ h{*}h', \sigma \vDash_{SL} y.g \overset{1}{\mapsto} \_)$$

By choosing h' to be the heap containing $x.f$ with full permission and value 5, and with no other location in its domain, we deduce a contradiction, since $h{*}h', \sigma \vDash_{SL} x.f \overset{1}{\mapsto} 5$ does hold, while $h{*}h', \sigma \vDash_{SL} y.g \overset{1}{\mapsto} \_$ does not. □

The second example formula listed above is actually a translation of the first into implicit dynamic frames: it is only true in states which have (full) permission to *both* locations $x.f$ and $y.g$ in our semantics, and for the same reasons as the previous example. However, this assertion goes beyond the syntax of implicit dynamic frames typically supported by tools; we will discuss this later. The third formula should mean: we have (full) permission to access $x.f$, and if its value is currently 5 then we also have full access to $y.f$. This kind of assertion is already supported by the Chalice tool, and has exactly that intuitive meaning.

We need to decide which of the notions of state extension, as given in Definition 3.5, we should use to define our semantics for implication. Since, in the traditional separation logic semantics, the two sides of an implication get judged in the whole resulting heap (after the state extension), the latter two "disjoint" variants from the definition are not appropriate. However, we still have the choice between considering locally-havoced or globally-havoced extensions. Let us consider the (slightly-simpler) option of using globally-havoced extensions. This leads us to the following candidate semantics for implication:

$$H, P, \sigma \vDash_{TPL} A_1 \to A_2 \stackrel{?}{\iff}$$
$$\forall P', H'.(\ P' \bot P\ \wedge\ H' \stackrel{P}{\equiv} H\ \wedge\ H', P * P', \sigma \vDash_{TPL} A_1 \Rightarrow\ H', P * P', \sigma \vDash_{TPL} A_2)$$

This definition gives the correct meaning to our first example assertion: since we judge the implication in a state in which we have no permission to $x.f$ (all such permission has to be given to the left-hand side of the $*$), we have to consider heaps H' in which $x.f$ has taken on arbitrary values. In particular, there are some such heaps in which $x.f$ has the value 5, and this forces the requirement that we must also have full permission to $y.g$, just as in the traditional separation logic semantics. In fact, using locally-havoced extensions in our definition (i.e., changing the constraint on H' to be $H' \stackrel{rds(P) \cup \overline{rds(P')}}{\equiv} H$), would also give the right semantics; since we are still required to consider the possibility that we add on permission to $x.f$ in the extra permissions P', and in this case, it is allowed for H' to differ with H on $x.f$'s value. In fact, it is generally the case that the choice of locally-havoced or globally-havoced extensions makes no difference when we consider separation logic assertions. Exactly the same arguments (and resulting semantics) apply to the second of our example assertions.

However, the candidate definition above does not in general have the correct meaning for implicit dynamic frames. In particular, our third example formula does not have the correct semantics. The change of heap forces us to consider extensions in which we alter the value of $x.f$ in the heap, and thus our third example also becomes equivalent to $acc(x.f) * acc(y.f)$, since the value restriction for $x.f$ is made irrelevant by this potential change. To see exactly what we need here, we need to again consider carefully the meaning of $x.f$ in a pure assertion. When such a heap-dependent expression occurs on the left of an implication, its intended meaning depends on where in the assertion we find permission to the heap location. There are two important questions: does a permission to this location *also* occur on the left of the implication (e.g., in our second example formula), and does a permission to this location occur elsewhere in the assertion? If neither occurs, i.e., there is no permission guaranteed to $x.f$ anywhere in the assertion, then the expression is meaningless; we will consider this case ill-formed, and so can give it any semantics. If a permission to $x.f$ occurs on the left of the implication in question (as in our second example formula), then it is possible that the value of the heap location is fixed as part of the heap extension, and therefore we should allow this value to change as part of the extension, as in our last candidate semantics above. In particular, if we judge the truth of the implication in a state in which we *start off* with no permission to the location $x.f$, then it is just by adding the new permission required on the left of the implication that we can read from the location, and so we should consider that the value may be different from that in our original state. Finally, if a permission does *not* occur on the left of the implication, but *does* occur elsewhere in the assertion (as in our third example formula $acc(x.f, 1) * (x.f = 5 \to acc(y.g, 1))$), then we should *not* allow the value of $x.f$ to change when judging the implication; its value must be determined by the

permission outside of the implication, and so should not be allowed to change when judging it.

This analysis leads us to the conclusion that, in our candidate semantics above, we are considering too many extensions. We need to only consider the *minimal* extensions of the state to make the left of the implication true; in particular, we should only add permissions if the left-hand side of the implication explicitly requires them, and only allow values to change at those locations to which we added new permission. To this end, we provide a definition to capture the idea of a *minimal permission extension*, which expresses that the extra permission we add on does not make any more locations readable than is necessary to make the assertion we are concerned with true:

Definition 3.6 (Minimal Permission Extensions)**.** Starting from a state (H, P, σ), we say that P' *is a minimal permission extension of* (H, P, σ) *to satisfy* A, which we write as $(H, P, \sigma) \triangleleft P' \vDash_{TPL} A$, as described by the following formula:

$$(H, P, \sigma) \triangleleft P' \vDash_{TPL} A \iff H, P * P', \sigma \vDash_{TPL} A \wedge \forall P'' \cdot P'' \subseteq P' \wedge rds(P'') \subset rds(P') \Rightarrow H, P * P'', \sigma \nvDash_{TPL} A$$

We abstract over the precise permission values in this minimal extension (by focusing on which locations are readable, using the *rds*() concept), in order to avoid imposing restrictions on the underlying permissions model (in particular, our definition does not depend on there being a greatest lower bound for the acceptable permission values to satisfy an assertion). That is, a minimal permission extension can add on more permission than the assertion really requires, so long as it does not increase the set of locations accessible in the permissions mask by more than necessary.

Using the concept of a minimal permission extension, along with the notion of locally-havoced extensions, we can finally define the semantics for implication which works for our general logic:

$$H, P, \sigma \vDash_{TPL} A_1 \to A_2 \iff \forall (H', P * P', \sigma) \in \mathsf{localExts}(H, P, \sigma) \cdot (H', P, \sigma) \triangleleft P' \vDash_{TPL} A_1 \Rightarrow H', P * P', \sigma \vDash_{TPL} A_2$$

This definition can be informally understood as follows: $A_1 \to A_2$ is true in a state if, for all minimal extensions (and corresponding havocs) of the state such that A_1 holds, A_2 must hold as well. The extension of the state is modelled by adding on the permissions P', and allowing the values of the heap to be modified in exactly the locations which become newly-readable by adding on these permissions. Furthermore, we insist on the permissions added being minimal in the locations which they make readable, while still satisfying A_1.

This definition correctly captures that we sometimes need to consider changing values in the heap when judging an implication, but only when it is permissions in the left-hand side of the implication that allow the reading of the locations. For example, using the definition above, the third example formula $acc(x.f, 1) * (x.f = 5 \to acc(y.f, 1))$ is true exactly in a state where we have (full) permission to $x.f$, and where, if the current value of $x.f$ is 5, we also have (full) permission to $y.f$. On the other hand, the second formula $acc(x.f, 1) * (acc(x.f, 1) * x.f = 5 \to acc(y.f, 1))$ is true exactly when we have full permission to both $x.f$ and $y.f$ - this is because the implication is evaluated in a state with no permission to $x.f$, and when our semantics considers extending this state by just enough permission to make the left-hand side true, we have to allow for the possibility that this makes $x.f$ newly readable, and thus, makes it take on a new value.

Although we did not discuss the semantics of the magic wand connective ($-\!\!*$) in the above, similar considerations lead us to also use the concept of minimal extensions in its definition. The formal definitions of our semantics come in the following subsection.

3.7. **Magic Wand Semantics.** We design our semantics for the "magic wand" connective $-\!\!*$ along similar lines to that for implication. In particular, if one writes a pure assertion A_1 on the left of a wand formula $A_1 -\!\!* A_2$, then we do not want the semantics of the assertion to consider havocing heap locations that A_1 refers to. Put another way, for pure A_1 we would like the property that $A_1 -\!\!* A_2$ is equivalent to $A_1 \to A_2$, which, as we have already decided, should have a semantics which allows A_1 to refer to heap values in our original state, unless extra permission to those locations is added in A_1. This reasoning leads us to again employ our notion of minimal permission extension, but this time we complement it with locally-havoced *disjoint* extensions:

$$H, P, \sigma \vDash_{TPL} A_1 -\!\!* A_2 \iff$$
$$\forall (H', P', \sigma) \in \mathsf{localDisjExts}(H, P, \sigma).(H', \varnothing, \sigma) \lhd P' \vDash_{TPL} A_1 \;\Rightarrow\; H', P * P', \sigma \vDash_{TPL} A_2)$$

3.8. **Total Heaps Semantics for *TPL*.** We can now define our semantics for assertions. We make use of the concept of a minimal permission extension (Definition 3.6) to describe minimal extensions to the whole state when judging implications and magic wand assertions:

Definition 3.7 (Total Heap Semantics for *TPL*)**.** We define validity of *TPL*-assertions with respect to a specified total heap H and permission mask P recursively on the structure of the assertion:

$$H, P, \sigma \vDash_{TPL} E.f \overset{\pi}{\mapsto} E' \iff P[[\![E]\!]_{\sigma,H}, f] \geq \pi \;\wedge\; H[[\![E]\!]_{\sigma,H}, f] = [\![E']\!]_{\sigma,H}$$

$$H, P, \sigma \vDash_{TPL} A_1 * A_2 \iff \exists P_1, P_2.(P = P_1 * P_2 \;\wedge\; H, P_1, \sigma \vDash_{TPL} A_1 \;\wedge\; H, P_2, \sigma \vDash_{TPL} A_2)$$

$$H, P, \sigma \vDash_{TPL} A_1 -\!\!* A_2 \iff \forall (H', P', \sigma) \in \mathsf{localDisjExts}(H, P, \sigma).$$
$$(H', \varnothing, \sigma) \lhd P' \vDash_{TPL} A_1 \;\Rightarrow\; H', P * P', \sigma \vDash_{TPL} A_2$$

$$H, P, \sigma \vDash_{TPL} A_1 \wedge A_2 \iff H, P, \sigma \vDash_{TPL} A_1 \;\wedge\; H, P, \sigma \vDash_{TPL} A_2$$

$$H, P, \sigma \vDash_{TPL} A_1 \vee A_2 \iff H, P, \sigma \vDash_{TPL} A_1 \;\vee\; H, P, \sigma \vDash_{TPL} A_2$$

$$H, P, \sigma \vDash_{TPL} A_1 \to A_2 \iff \forall (H', P * P', \sigma) \in \mathsf{localExts}(H, P, \sigma).$$
$$(H', P, \sigma) \lhd P' \vDash_{TPL} A_1 \;\Rightarrow\; H', P * P', \sigma \vDash_{TPL} A_2$$

$$H, P, \sigma \vDash_{TPL} acc(E.f, \pi) \iff P[[\![E]\!]_{\sigma,H}, f] \geq \pi$$

$$H, P, \sigma \vDash_{TPL} E = E' \iff [\![E]\!]_{\sigma,H} = [\![E']\!]_{\sigma,H}$$

$$H, P, \sigma \vDash_{TPL} \exists x.\, A \iff \exists v.(H, P, \sigma[x \mapsto v] \vDash_{TPL} A)$$

Note the similarity between the definitions for magic wand $-\!\!*$ and logical implication $\to$. 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 added permissions to.

Evaluation of *TPL* expressions depends on a given environment and heap, and is defined by:

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

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

Lemma 3.8. $\forall e, \sigma, H.\ [\![e]\!]_{\sigma,H} = [\![e]\!]_{\sigma}$

The main aims of the rest of this section are to show that our assertion semantics also preserves the original meaning of separation logic assertions.

3.9. **Strengthening and Weakening Results.** In this subsection, we present some of the technical properties which describe how our semantics behaves when we add and remove permissions, and when we extend states.

The following lemma shows the (intuitive) property that we can always discard superfluous permissions to reach a minimal permission extension:

Lemma 3.9 (Minimisation of Permission Masks)**.** If $H, P_1 * P_2, \sigma \vDash_{TPL} A$ then $\exists P_3 \subseteq P_2$ such that $(H, P_1, \sigma) \lhd P_3 \vDash_{TPL} A$. (See page 39 for proof.)

Definition 3.10 (Weakening-closed and Intuitionistic formulas)**.** We define a formula A to be *weakening-closed* if and only if

$$\forall H, P, \sigma, P'.(P \subseteq P' \ \wedge \ H, P, \sigma \vDash_{TPL} A \quad \Rightarrow \quad H, P', \sigma \vDash_{TPL} A)$$

We define a formula A to be *intuitionistic* if and only if

$$\forall H, P, \sigma.(H, P, \sigma \vDash_{TPL} A \quad \Rightarrow \quad \forall (H', P', \sigma) \in \mathsf{globalExts}(H, P, \sigma).\ H', P', \sigma \vDash_{TPL} A)$$

Lemma 3.11. If A is weakening-closed, $(H, P, \sigma) \lhd P' \vDash_{TPL} A$, $P' \subseteq P''$ and $rds(P') = rds(P'')$, then $(H, P, \sigma) \lhd P'' \vDash_{TPL} A$ (See page 39 for proof.)

The following technical lemma shows that any necessary permissions in a state are still necessary in a state with fewer permissions, provided the assertion we are considering is closed under permission extension:

Lemma 3.12 (Minimal Permission Extensions Closed)**.** If $(H, P_1 * P_2, \sigma) \lhd P_3 \vDash_{TPL} A$ and A is weakening-closed, then $\exists P_4 \subseteq P_2$ and $(H, P_1, \sigma) \lhd P_4 * P_3 \vDash_{TPL} A$. (See page 39 for proof.)

The validity of assertions in this semantics is closed under permission extension.

Proposition 3.13. All formulas A are weakening-closed.

Proof sketch: By induction on structure of formula. *(Full proof on page 40)*

In our later results, we sometimes need to be able to define when a minimal permission extension in one state corresponds with a minimal permission extension in another. In particular, we want to be able to show that, under certain conditions, the notion of what is a minimal extension in a current state is robust to interference. An important property which helps us here, is to be able to express that the truth of an assertion is stable in all extensions of a particular state. That is, even if the assertion does not hold in the current state, in all extensions which do satisfy the assertion, its truth will be stable. This can be expressed by the following definitions.

Definition 3.14. An assertion A is *extension framed* in a state (H, P, σ) if and only if, A is stable in all (globally-havoced) extensions, i.e.,

$$\mathsf{ExtFrm}(H, P, \sigma, A) \iff \mathsf{stable}(\mathsf{globalExts}(H, P, \sigma) \cap (\!|A|\!))$$

We also define the set of states in which A is extension framed:

$$\mathsf{ExtFrm}(A) = \{(H, P, \sigma) \mid \mathsf{ExtFrm}(H, P, \sigma, A)\}$$

An assertion A is *disjoint extension framed* in the current memory if and only if, in all (globally-havoced) disjoint extensions, the truth of A is stable with the permissions held originally:

$$\mathsf{DisExtFrm}(H, P, \sigma, A) \iff \mathsf{stable\text{–}with}_P\,(\mathsf{globalDisjExts}(H, P, \sigma) \cap (\!|A|\!)\})$$

We also define the set of states in which A is disjoint-extension framed:

$$\mathsf{DisExtFrm}(A) = \{(H, P, \sigma) \mid \mathsf{DisExtFrm}(H, P, \sigma, A)\}$$

Note that we use the globally-havoced notions of state extension (cf. Definition 3.5), rather than locally-havoced. The reason for this is that, we need this criterion on assertions to be preserved under interference, at various points in our proofs. The following lemma characterises the essential properties that we require of these definitions.

Lemma 3.15.

(1) If $(H, P, \sigma) \in \mathsf{ExtFrm}(A)$, then:
  (a) if $H' \in \mathsf{interfere}(H, P)$, then $(H', P, \sigma) \in \mathsf{ExtFrm}(A)$.
  (b) if $P' \perp P$ and $H' \in \mathsf{interfere}(H, P * P')$ and $H, P * P', \sigma \vDash_{TPL} A$, then $H', P * P', \sigma \vDash_{TPL} A$.

(2) If $(H, P, \sigma) \in \mathsf{DisExtFrm}(A)$, then:
  (a) if $H' \in \mathsf{interfere}(H, P)$, then $(H', P, \sigma) \in \mathsf{DisExtFrm}(A)$.
  (b) if $P' \perp P$ and $H' \in \mathsf{interfere}(H, P * P')$ and $H, P', \sigma \vDash_{TPL} A$, then $H', P', \sigma \vDash_{TPL} A$.

(See page 41 for proof.)

Note that using $\mathsf{localExts}(H, P, \sigma)$, or $\mathsf{localDisjExts}(H, P, \sigma)$, rather than $\mathsf{globalExts}(H, P, \sigma)$, or $\mathsf{globalDisjExts}(H, P, \sigma)$, invalidates part (1.a), or part (2.a), respectively.

The following technical lemma provides sufficient conditions for a minimal permission extension to be robust to interference in the rest of the heap:

Lemma 3.16 (Preservation of Minimal Extensions)**.**

(1) For all $(H_1, P_1, \sigma) \in \mathsf{ExtFrm}(A)$,

$$\forall P_2 \perp P_1, \forall H_2 \overset{P_1 * P_2}{\equiv} H_1.((H_1, P_1, \sigma) \lhd P_2 \vDash_{TPL} A \Rightarrow (H_2, P_1, \sigma) \lhd P_2 \vDash_{TPL} A)$$

(2) For all $(H_1, P_1, \sigma) \in \mathsf{DisExtFrm}(A)$,

$$\forall P_2 \perp P_1, \forall H_2 \overset{P_1 * P_2}{\equiv} H_1.((H_1, \varnothing, \sigma) \lhd P_2 \vDash_{TPL} A \Rightarrow (H_2, \varnothing, \sigma) \lhd P_2 \vDash_{TPL} A)$$

(3) If A is self-framing and $P_2 \perp P_1$ and $(H_1, P_1, \sigma) \lhd P_2 \vDash_{TPL} A$ and $H_2 \overset{P_1 * P_2}{\equiv} H_1$, then $(H_2, P_1, \sigma) \lhd P_2 \vDash_{TPL} A$

(See page 42 for proof.)

We will make use of these technical lemmas in the next subsections, in order to characterise properties of our general semantics.

3.10. **Correspondence with Separation Logic Semantics.** In this subsection, we examine the correspondence between the semantics which our definition implies for the separation logic fragment of our logic, and the traditional semantics of separation logic. In order to precisely characterise the laws which hold of the logic, we require a notion of semantic entailment.

Definition 3.17 (Semantic Entailment, Validity and Equivalence)**.** A *TPL* assertion A is *semantically valid* (written $\vDash_{TPL} A$) if it holds in all situations; i.e.,

$$\vDash_{TPL} A \Leftrightarrow \ \forall H, P, \sigma.\ H, P, \sigma \vDash_{TPL} A$$

Given *TPL* assertions A_1 and A_2, we say that A_1 *semantically entails* A_2 (and write $A_1 \vDash_{TPL} A_2$) if and only if A_2 holds whenever A_1 does; i.e.,

$$A_1 \vDash_{TPL} A_2 \ \Leftrightarrow\ \langle\!\langle A_1 \rangle\!\rangle \subseteq \langle\!\langle A_2 \rangle\!\rangle$$

Given *TPL* assertions A_1 and A_2, we say that A_1 *is equivalent to* A_2 (and write $A_1 \equiv_{TPL} A_2$) if and only if $A_1 \vDash_{TPL} A_2$ and $A_2 \vDash_{TPL} A_1$.

For pure assertions, our (rather complex) definition of implication can be simplified to a simple boolean evaluation of the conditional:

Lemma 3.18 (Pure Assertions are Boolean Conditionals)**.** *If A_1 is pure, then:*

$$H, P, \sigma \vDash_{TPL} A_1 \to A_2 \quad \Longleftrightarrow \quad (H, P, \sigma \vDash_{TPL} A_1 \ \ \Rightarrow \ \ H, P, \sigma \vDash_{TPL} A_2)$$

Proof. We first observe that if A_1 is pure and $(H, P, \sigma) \triangleleft P' \vDash_{TPL} A_1$, then $P' = \varnothing$. Simplifying the semantic definition of $\to$ using the $\varnothing$ gives the required semantics. □

Note that this property was not true in the semantics of the precursor paper [15], and prevents the former work from correctly modelling Chalice's implication.

The following lemma also shows how our definition of semantics for implication and the magic wand can be simplified if we know that the immediate subformulas are self-framing assertions (in this case, we do not encounter the technical difficulties which led us to employ *minimal* extensions; cf. Section 3.6):

Lemma 3.19 (Simplified Semantics for Self-Framing Conditionals)**.**

(1) If A_1 and A_2 are both self-framing, then:

(a) $H, P, \sigma \vDash_{TPL} A_1 \to A_2$ if and only if:

$$\forall (H', P', \sigma) \in \mathsf{localExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1 \ \ \Rightarrow \ \ H', P', \sigma \vDash_{TPL} A_2)$$

(b) $H, P, \sigma \vDash_{TPL} A_1 \to A_2$ if and only if:

$$\forall (H', P', \sigma) \in \mathsf{globalExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1 \ \ \Rightarrow \ \ H', P', \sigma \vDash_{TPL} A_2)$$

(2) If A_1 and A_2 are both self-framing, then:

(a) $H, P, \sigma \vDash_{TPL} A_1 \mathrel{-\!\!*} A_2$ if and only if:

$$\forall (H', P', \sigma) \in \mathsf{localDisjExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1 \ \ \Rightarrow \ \ H', P * P', \sigma \vDash_{TPL} A_2)$$

(b) $H, P, \sigma \vDash_{TPL} A_1 \mathrel{-\!\!*} A_2$ if and only if:

$$\forall (H', P', \sigma) \in \mathsf{globalDisjExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1 \ \ \Rightarrow \ \ H', P * P', \sigma \vDash_{TPL} A_2)$$

(See page 42 for proof.)

This lemma provides two alternative semantics for the implication and wand connectives, which are both equivalent to our actual semantics of Definition 3.7 if we restrict the logic to self-framing subformulas. In particular, to model the fragment of our logic which corresponds to separation logic, these alternative semantics are sufficient. The latter alternative for each connective (defined in terms of globally-havoced extensions) is the semantics used in our precursor paper [15], while the former (using locally-havoced extensions) is convenient to simplify several of our proofs.

Note that the concepts of minimal permission extensions, and locally-havoced extensions (neither of which were used in our precursor paper) are *not* motivated by our desire to correctly model separation logic semantics in our total heaps model; as the lemma above makes explicit, a simpler semantics could have been defined if this was our only goal. However, that semantics does not extend to correctly handle the implication in implicit dynamic frames, for which we needed the concept of minimal extensions to get the general case correct, as is motivated in Subsection 3.6.

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 P we can construct a corresponding partial heap $H{\restriction}P$. 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 $h = H{\restriction}P$. However, the different choices of H can only differ over the locations given no permission in P, and Corollary 3.21 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 $H{\restriction}P$. We can then show that our total heap semantics for *SL* is sound and complete with respect to the standard semantics:

Theorem 3.20 (Correctness of Total Heap Semantics)**.** For all *SL*-assertions a, environments σ, total heaps H, and permission masks P:

$$H, P, \sigma \vDash_{TPL} a \iff (H{\restriction}P), \sigma \vDash_{SL} a$$

Proof sketch: By induction on the structure of a. (Full proof on page 44)

This result demonstrates that our total heap semantics correctly models the standard semantics of separation logic assertions.

Corollary 3.21. *All separation logic assertions a (Defn 2.1) are self-framing.*

Corollary 3.22. *All separation logic assertions a are intuitionistic.*

3.11. **Separation Logic Laws.** 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 *TPL*. 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 *TPL*. We can show how various laws which hold for separation logic transfer (in some cases partially) to our more general setting of *TPL*. Firstly, we need a technical lemma which shows how to break down minimal permission extensions over (separating and logical) conjunctions:

Lemma 3.23 (Decomposing Minimal Permission Extensions over Conjunctions)**.**
(1) If $(H, \varnothing, \sigma) \lhd P' \vDash_{TPL} A_1 * A_2$ then $\exists P_1, P_2$ such that $P' = P_1 * P_2$ and $(H, \varnothing, \sigma) \lhd P_1 \vDash_{TPL} A_1$ and $(H, \varnothing, \sigma) \lhd P_2 \vDash_{TPL} A_2$.
(2) If $(H, P, \sigma) \lhd P' \vDash_{TPL} A_1 \wedge A_2$ then $\exists P_1, P_2$ such that $P' = P_1 * P_2$ and $(H, P, \sigma) \lhd P_1 \vDash_{TPL} A_1$ and $(H, P * P_1, \sigma) \lhd P_2 \vDash_{TPL} A_2$.

(See page 45 for proof.)

Certain technical properties which follow do not hold for general formulas, but only for those that do not behave disjunctively. Following, O'Hearn *et al.* [13], we call these formulas *supported*. A formula such as $acc(x.f, 1) \vee acc(y.f, 1)$ is not supported, while $(b = 1 \to acc(x.f, 1)) * (b \neq 1 \to acc(y.f, 1)$ is supported.

Definition 3.24 (Supported Formulas)**.** A formula A is *supported* iff for all H, σ, P_1 and P_2, if $H, P_1, \sigma \vDash_{TPL} A$ and $H, P_2, \sigma \vDash_{TPL} A$, then $H, P_1 \sqcap P_2, \sigma \vDash_{TPL} A$.

Supported assertions allow minimal permission extensions to be combined for both $*$ and $\wedge$.

Lemma 3.25 (Composing Minimal Permission Extensions over Supported Conjunctions)**.**
(1) If $(H, \varnothing, \sigma) \lhd P_1 \vDash_{TPL} A_1$ and $(H, \varnothing, \sigma) \lhd P_2 \vDash_{TPL} A_2$ and A_1 and A_2 are supported, then $(H, \varnothing, \sigma) \lhd P_1 * P_2 \vDash_{TPL} A_1 * A_2$.
(2) If $(H, P, \sigma) \lhd P_1 \vDash_{TPL} A_1$ and $(H, P * P_1, \sigma) \lhd P_2 \vDash_{TPL} A_2$ and A_1 and A_2 are supported, then $(H, P, \sigma) \lhd P_1 * P_2 \vDash_{TPL} A_1 \wedge A_2$.

(See page 46 for proof.)

We can now show which of the usual separation logic laws carry over to our more general logic, and under which conditions:

Proposition 3.26. For all *TPL* assertions A_1, A_2, A_3:
(1) $A_1 * (A_1 \mathrel{-\!\!*} A_2) \vDash_{TPL} A_2$
(2) $A_1 \wedge (A_1 \to A_2) \vDash_{TPL} A_2$
(3) (a) $\mathsf{DisExtFrm}(A_1) \cap \langle\!\langle A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3) \rangle\!\rangle \subseteq \langle\!\langle (A_1 * A_2) \mathrel{-\!\!*} A_3 \rangle\!\rangle$
(b) if A_1 and A_2 are supported, then:
$\mathsf{DisExtFrm}(A_1) \cap \langle\!\langle (A_1 * A_2) \mathrel{-\!\!*} A_3 \rangle\!\rangle \subseteq \langle\!\langle (A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3) \rangle\!\rangle$
(c) if both $A_1 * A_2$ and A_3 are self-framing, then:
$\mathsf{DisExtFrm}(A_1) \cap \langle\!\langle (A_1 * A_2) \mathrel{-\!\!*} A_3 \rangle\!\rangle \subseteq \langle\!\langle (A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3) \rangle\!\rangle$
(4) (a) $\mathsf{ExtFrm}(A_1) \cap \langle\!\langle A_1 \to (A_2 \to A_3) \rangle\!\rangle \subseteq \langle\!\langle (A_1 \wedge A_2) \to A_3 \rangle\!\rangle$
(b) if A_1 and A_2 are supported, then:
$\mathsf{ExtFrm}(A_1) \cap \langle\!\langle (A_1 \wedge A_2) \to A_3 \rangle\!\rangle \subseteq \langle\!\langle A_1 \to (A_2 \to A_3) \rangle\!\rangle$
(c) if both $A_1 \wedge A_2$ and A_3 are self-framing, then:
$\mathsf{ExtFrm}(A_1) \cap \langle\!\langle (A_1 \wedge A_2) \to A_3 \rangle\!\rangle \subseteq \langle\!\langle A_1 \to (A_2 \to A_3) \rangle\!\rangle$
(5) If $A_1 \vDash_{TPL} (A_2 \mathrel{-\!\!*} A_3)$ then $(A_1 * A_2) \vDash_{TPL} A_3$
(6) If A_1 is self-framing and $(A_1 * A_2) \vDash_{TPL} A_3$ then $A_1 \vDash_{TPL} (A_2 \mathrel{-\!\!*} A_3)$

(See page 47 for proof.)

To see that the usual separation logic laws do not all hold in general, consider for example the two assertions $A_1 \stackrel{def}{=} (x.f = 1 \mathrel{-\!\!*} (acc(x.f, 1) \mathrel{-\!\!*} \mathsf{false}))$ and $A_2 \stackrel{def}{=} (x.f = 1 * acc(x.f, 1)) \mathrel{-\!\!*} \mathsf{false}$. The assertion A_2 is equivalent to $acc(x.f, \_)$, that is a permissions mask, which cannot be extended with disjoint full access to $x.f$. However, the assertion A_1 is also true in models where the heap maps $x.f$ to a value other than 1, as the outer wand does not get to change the current heap.

The usual separation logic laws do however hold for self-framing assertions which (by Lemma 3.21) includes all separation logic assertions.

Corollary 3.27. *For all self-framaing TPL assertions A_1, A_2, A_3:*
(1) $A_1 * (A_1 \mathrel{-\!\!*} A_2) \vDash_{TPL} A_2$
(2) $A_1 \wedge (A_1 \to A_2) \vDash_{TPL} A_2$
(3) $A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3) \equiv_{TPL} (A_1 * A_2) \mathrel{-\!\!*} A_3$
(4) $A_1 \to (A_2 \to A_3) \equiv_{TPL} (A_1 \wedge A_2) \to A_3$
(5) $A_1 \vDash_{TPL} (A_2 \mathrel{-\!\!*} A_3)$ *if and only if* $(A_1 * A_2) \vDash_{TPL} A_3$

3.12. **Existentials and Substitution.** Next we consider when it is valid to replace a variable with an expression it is equal to; that is, under what condition is $\exists x.x = E * A$ equivalent to $A[E/x]$. If the expression does not depend on the heap, then this equivalence holds.

Lemma 3.28. *For any* separation logic expression e*:*

$$(\exists x.x = e * A) \equiv_{TPL} A[e/x]$$

Proof. We prove

$$H, P, \sigma \vDash_{TPL} A[e/x] \Leftrightarrow H, P, \sigma[x \mapsto [\![e]\!]_{H,\sigma}] \vDash_{TPL} A$$

and

$$[\![E[E'/x]]\!]_{H,\sigma} \Leftrightarrow [\![E]\!]_{H,\sigma[x \mapsto [\![E']\!]_{H,\sigma}]}$$

by straightforward inductions on structures of A and E, respectively. □

However, if the expression depends on the heap, then the problem is more challenging. Consider the example formula

$$\exists v.\ v = x.f * acc(x.f, \pi) * (acc(x.f, \pi) \mathrel{-\!\!*} v = 5)$$

This formula is semantically equivalent (noting that changes to the heap do not affect the interpretation of v) to

$$acc(x.f, \pi) * x.f = 5$$

However, if we apply the standard substitution on the formula, replacing v with the expression $x.f$, then we get

$$acc(x.f, \pi) * (acc(x.f, \pi) \mathrel{-\!\!*} x.f = 5)$$

which is equivalent to *false* (recall that the semantics for the $\mathrel{-\!\!*}$ connective considers "adding on" new permission for $x.f$ in this case, which includes considering changing its value arbitrarily). More abstractly, the difficulty here is that the semantics of $\mathrel{-\!\!*}$, and $\to$ consider *changes* to the current heap; in general this is incompatible with treating heap-dependent expressions as purely syntactic entities which can be moved around amongst subformulas freely, as we would if we wanted a substitution property for such expressions. In particular, the meaning of a heap-dependent expression can differ in different positions in a formula, depending on its nesting under $\mathrel{-\!\!*}$ and $\to$ connectives[6]. For this reason, if we wanted such a property, we would need to restrict the uses of $\mathrel{-\!\!*}$ and $\to$ to enable the substitution of

[6]One can compare with the analogous situation in standard separation logic: an SL formula such as $x.f \overset{\pi}{\mapsto} u * (x.f \overset{\pi}{\mapsto} v \to u = v)$ is not actually valid in traditional intuitionistic separation logic semantics for the same reasons; the semantics of the implication connective includes the concept of "adding on" new access to $x.f$ when evaluating the implication.

expressions with heap dependencies. To illustrate this, we define a class of formulas, that are *substitutable*. These are the formulas that only contain pure formulas on the left of $-\!\!*$ and $\rightarrow$.

Definition 3.29 (Substitutable formulas)**.** We define a formula as *substitutable*, $\mathsf{subst}(A)$, by

$$\begin{array}{lll}
\mathsf{subst}(A_1 \multimap A_2) & \iff \mathsf{subst}(A_1) \wedge \mathsf{subst}(A_2) \text{ and } A_1 \text{ is pure} & \\
 & & (\text{where } \multimap \in \{-\!\!*, \rightarrow\}) \\
\mathsf{subst}(A_1 \circ A_2) & \iff \mathsf{subst}(A_1) \wedge \mathsf{subst}(A_2) & (\text{where } \circ \in \{\vee, \wedge, *\}) \\
\mathsf{subst}(\exists x.\ A) & \iff \mathsf{subst}(A) & \\
\mathsf{subst}(acc(E.f, \pi)) & \iff \mathsf{subst}(E = E') \iff \mathsf{subst}(E.f \overset{\pi}{\mapsto} E') \iff \text{always} &
\end{array}$$

As substitutable formulas only have pure formulas on the left of $-\!\!*$ and $\rightarrow$, there is a single heap that is used to evaluate the entire formula.

Lemma 3.30. *If* $\mathsf{subst}(A)$*, then*

$$(\exists x.x = E * A) \equiv_{TPL} A[E/x]$$

Proof. We prove

$$H, P, \sigma \vDash_{TPL} A[E/x] \Leftrightarrow H, P, \sigma[x \mapsto [\![E]\!]_{H,\sigma}] \vDash_{TPL} A$$

by straightforward induction on the $\mathsf{subst}(A)$ predicate. The $-\!\!*$ and $\rightarrow$ cases use that for pure formulas they behave like boolean conditionals (Lemma 3.18). We reuse the expression substitutability proof from previous lemma. □

In Section 5, we will present an encoding from the SL fragment to the IDF fragment of our logic, which preserves semantics. A natural question to ask is, can we encode back from IDF to SL, at least for those IDF assertions which are self-framing? In general, it is surprisingly difficult to define a suitable syntactic translation. A tempting approach is to convert all $acc(x.f, \pi)$ assertions into $x.f \overset{\pi}{\mapsto} v$ for some fresh logical variable v, and then to replace any heap-dependent expressions $x.f$ with v elsewhere in the assertion. But this approach fails in two ways: firstly, it does not deal correctly with aliasing. The criteria for an IDF to be self-framing take account of constraints imposed by the assertion itself; for example, $acc(x.f) * x = y * y.f = 4$ is self-framing. This makes a syntactic replacement of heap-dependent expressions challenging. Furthermore, the correctness of the replacement of all heap-dependent expressions with logical variables, depends on a substitution property holding for such expressions. As discussed above, this does not hold for the general logic; the meaning of a heap-dependent expression is actually fixed by the "closest scoped" occurrence of a permission to that location, with respect to implications and wands; in the presence of aliasing this is hard to determine.

For the subsyntaxes of these logics typically supported by tools, which generally only allow for pure assertions on the left of $\rightarrow$ formulas (and do not support $-\!\!*$ in general), we do get a substitution property, as shown above. However, the problem of correctly handling aliasing between heap locations when translating heap-dependent expressions, still seems to make defining a correct syntactic translation challenging.

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 *TPL*.

4.1. **Chalice.** Chalice is defined by a translation into Boogie2 [9], 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. Similarly, $\vDash_{FO} \phi$ means that ϕ holds in all such 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 mask. Thus, $\mathcal{H}[x, f] = 5$ means that in the current heap the variable x's field named f contains the value 5. In the assertion logic, this corresponds to $x.f = 5$, in which the heap access is implicit.

To define the verification conditions for Chalice, we need to be able to translate expressions into the underlying logic using access to the map $\mathcal{H}$. We can provide a syntactic translation from the Chalice assertion logic into the first-order logic.

Definition 4.1. We translate expressions that implicitly access the heap into expressions that explicitly access the heap as follows:

$$\llbracket x \rrbracket = x \qquad \llbracket \texttt{null} \rrbracket = \texttt{null} \qquad \llbracket E.f \rrbracket = \mathcal{H}[\llbracket E \rrbracket, f]$$

we translate boolean expressions as:

$$\llbracket B_1 * B_2 \rrbracket = \llbracket B_1 \rrbracket \wedge \llbracket B_2 \rrbracket \quad \llbracket E = E' \rrbracket = \llbracket E \rrbracket = \llbracket E' \rrbracket \quad \llbracket E \neq E' \rrbracket = \llbracket E \rrbracket \neq \llbracket E' \rrbracket$$

First, we must show some basic facts about the properties of Chalice assertions (cf. Definition 2.4): every Chalice boolean expression is pure, and every Chalice assertion is supported.

Lemma 4.2. *Every Chalice boolean expression B is pure.*

Proof. By trivial induction on B. □

Lemma 4.3. *Every Chalice formula p is supported.*

Proof. We must show

$$H, P, \sigma \vDash_{TPL} p \ \wedge \ H, P', \sigma \vDash_{TPL} p \ \Rightarrow \ H, P \sqcap P', \sigma \vDash_{TPL} p$$

We proceed by induction on p

$p \equiv B$: As B is pure, we know that if it is satisfied in a state, it will also be satisfied with any alternative permission mask.

$p \equiv acc(E.f, \pi)$: P and P' must map the field location to π or greater, therefore $P \sqcap P'$ will also map the field location to π or greater.

$p \equiv p_1 * p_2$: Assume $H, P, \sigma \vDash_{TPL} p_1 * p_2$, and $H, P', \sigma \vDash_{TPL} p_1 * p_2$. Therefore there exist P_1, P_2, P_1' and P_2' such that $P_1 * P_2 = P$ and $P_1' * P_2' = P'$ and $H, P_1, \sigma \vDash_{TPL} p_1$ and $H, P_2, \sigma \vDash_{TPL} p_2$ and $H, P_1', \sigma \vDash_{TPL} p_1$ and $H, P_2', \sigma \vDash_{TPL} p_2$. By induction, we know $H, P_1 \sqcap P_1', \sigma \vDash_{TPL} p_1$ and $H, P_2 \sqcap P_2', \sigma \vDash_{TPL} p_2$, and thus we can deduce that $H, (P_1 \sqcap P_1') * (P_2 \sqcap P_2'), \sigma \vDash_{TPL} p_1 * p_2$. We can show $(P_1 \sqcap P_1') * (P_2 \sqcap P_2') \subseteq (P_1 * P_2) \sqcap (P_1' * P_2')$, which by Proposition 3.13 proves the obligation.

$p \equiv B \to p'$: Case split on $H, \varnothing, \sigma \vDash_{TPL} B$. If B is true, then using Lemma 3.18 the result follows directly by induction. If B is false, then using Lemma 3.18 we have $H, \varnothing, \sigma \vDash_{TPL} B \to p$ as required. □

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 more-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. In Chalice the check is actually implemented by a Boogie program. Here, we use the logic to define an equivalent condition[7].

Definition 4.4 (Syntactic Self-Framing)**.** We define a condition $\mathsf{sframed}(E)$ to express that all the fields mentioned in E are accessible.

$$\begin{aligned} \mathsf{sframed}(E.f) &\iff \mathsf{sframed}(E) \wedge acc(E.f, \_) \\ \mathsf{sframed}(x) &\iff \mathsf{True} \\ \mathsf{sframed}(\texttt{null}) &\iff \mathsf{True} \end{aligned}$$

We lift this to boolean expressions as

$$\begin{aligned} \mathsf{sframed}(E = E') &\iff \mathsf{sframed}(E) \wedge \mathsf{sframed}(E') \\ \mathsf{sframed}(E \neq E') &\iff \mathsf{sframed}(E) \wedge \mathsf{sframed}(E') \\ \mathsf{sframed}(B_1 * B_2) &\iff \mathsf{sframed}(B_1) \wedge (B_1 \to \mathsf{sframed}(B_2)) \end{aligned}$$

We lift this to formulas as

$$\begin{aligned} \mathsf{sframed}(B \to p) &\iff \mathsf{sframed}(B) \wedge (B \to \mathsf{sframed}(p)) \\ \mathsf{sframed}(acc(E.f, \pi)) &\iff \mathsf{sframed}(E) \\ \mathsf{sframed}(p_1 * p_2) &\iff \mathsf{sframed}(p_1) \wedge (p_1 \mathrel{-\!\!*} \mathsf{sframed}(p_2)) \end{aligned}$$

Note that when we check that p_2 is framed in $p_1 * p_2$, we can use the assertion p_1; these checks do not treat $*$ as commutative.

A formula, p, is syntactically self-framing, if and only if $\vDash_{TPL} \mathsf{sframed}(p)$.

We prove some basic facts about $\mathsf{sframed}(E)$ and $\mathsf{sframed}(B)$: (1) in any state in which $\mathsf{sframed}(E)$ holds, changing the value at any locations without permissions does not affect E's evaluation; (2) $\mathsf{sframed}(E)$ is (semantically) self-framing; (3) in any state in which $\mathsf{sframed}(B)$ holds, changing the value at any locations without permissions does not affect B's evaluation.

[7]The end result of this section can be used to prove it is equivalent to verifying the Boogie program that Chalice would generate.

$$wp_{\text{ch}}(\texttt{exhale}(B), \phi) = wp_{\text{ch}}(\texttt{assert } [\![B]\!], \phi)$$

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

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

$$wp_{\text{ch}}(\texttt{exhale}(B \rightarrow p), \phi) = wp_{\text{ch}}((\texttt{assume}([\![B]\!]); \texttt{exhale}(p)) + \texttt{assume}(\neg[\![B]\!]), \phi)$$

$$wp_{\text{ch}}(\texttt{inhale}(B), \phi) = wp_{\text{ch}}(\texttt{assume } [\![B]\!], \phi)$$

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

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

$$wp_{\text{ch}}(\texttt{inhale}(B \rightarrow p), \phi) = wp_{\text{ch}}((\texttt{assume}([\![B]\!]); \texttt{inhale}(p)) + \texttt{assume}(\neg[\![B]\!]), \phi)$$

where

$$wp_{\text{ch}}(\mathcal{P}[o, f] := x, \phi) = \phi[upd(\mathcal{P}, (o, f), x)/\mathcal{P}]$$
$$wp_{\text{ch}}(\texttt{havoc}(\mathcal{H}[x, f]), \phi) = \phi[upd(\mathcal{H}, (x, f), z)/\mathcal{H}] \qquad \text{fresh z}$$
$$wp_{\text{ch}}(\texttt{assume } \phi', \phi) = \phi' \rightarrow \phi$$
$$wp_{\text{ch}}(\texttt{assert } \phi', \phi) = \phi' \wedge \phi$$
$$wp_{\text{ch}}(C_1; C_2, \phi) = wp_{\text{ch}}(C_1, wp_{\text{ch}}(C_2, \phi))$$
$$wp_{\text{ch}}(C_1 + C_2, \phi) = wp_{\text{ch}}(C_1, \phi) \wedge wp_{\text{ch}}(C_2, \phi)$$

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

Figure 1: Abridged weakest pre-condition semantics for Chalice [10]

Lemma 4.5.

(1) If $H, P, \sigma \vDash_{TPL} \mathsf{sframed}(E)$, and $H' \overset{P}{\equiv} H$ then $[\![E]\!]_{H,\sigma} = [\![E]\!]_{H',\sigma}$.
(2) $\mathsf{sframed}(E)$ is self-framing
(3) If $H, P, \sigma \vDash_{TPL} \mathsf{sframed}(B)$, and $H' \overset{P}{\equiv} H$ then $H, P, \sigma \vDash_{TPL} B$ if and only if $H', P, \sigma \vDash_{TPL} B$.
(4) $\mathsf{sframed}(B)$ is self-framing.

(See page 51 for proof.)

The key property we require of the $\mathsf{sframed}(p)$ definition is that it allows a wand ($-\!\!*$) of a separating conjunction, to be considered as a sequence of wands.

Lemma 4.6.

(1) $\mathsf{sframed}(p_1) \wedge ((p_1 * p_2) -\!\!* p) \vDash_{TPL} p_1 -\!\!* (p_2 -\!\!* p)$
(2) $\mathsf{sframed}(p_1) \wedge (p_1 -\!\!* (p_2 -\!\!* p)) \vDash_{TPL} (p_1 * p_2) -\!\!* p$

Proof sketch: This proof follows from Proposition 3.26.3.a and 3.26.3.b, Lemma 4.3, and showing

$$\forall p.\ \langle\!\langle \mathsf{sframed}(p) \rangle\!\rangle \subseteq \mathsf{DisExtFrm}(p)$$

This is proved by induction on p. (Full proof on page 52)

We can now provide the definitions of the weakest pre-conditions of the commands `inhale` and `exhale`. In Figure 1, we present the weakest pre-conditions of commands in Chalice from [10]. We write $wp_{\text{ch}}(C, \phi)$ for the weakest pre-condition of the command

C given the post-condition ϕ. Chalice models the inhaling and exhaling of permissions by mutating the permission mask variable. To exhale an equality (or any formula not mentioning the permission mask) we simply assert that 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 accessibility 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 is simply the same as assuming it. To inhale $p * q$, 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.

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 *TPL*. 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 *TPL* is also a model for Chalice.

We write $wp_{\text{sl}}(C, A)$, to be the weakest pre-condition in *TPL* of the formula A with respect to the command C. We treat `inhale` and `exhale` as the multiplicative versions of `assume` and `assert` (see §2.1.1), and thus have the following weakest pre-conditions:

$$wp_{\text{sl}}(\texttt{exhale}(A_1), A_2) = A_1 * A_2 \qquad wp_{\text{sl}}(\texttt{inhale}(A_1), A_2) = A_1 \mathrel{-\!\!*} A_2$$

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

First we must extend the first order logic we are considering to additionally contain a proposition to represent separation logic assertions.

Definition 4.7 ($\mathsf{interp}(A)$)**.** We extend the many sorted first order logic with an additional atomic proposition $\mathsf{interp}(A)$, which represents the interpretation of an arbitrary TPL formula in first order logic.

$$\sigma, \mathcal{H} \mapsto H, \mathcal{P} \mapsto P \vDash_{FO} \mathsf{interp}(A) \quad \iff \quad H, P, \sigma \vDash_{TPL} A$$

Note, this definition is not required by Chalice, but it allows us to express our proof by induction on the structure of the formula, by providing a single logic in which we can describe both the Chalice VCs and the TPL judgements.

Definition 4.8 ($equiv(C)$)**.** We define the VCs of a command as equivalent in both systems, $equiv(C)$, iff for every *TPL* assertion A, we have

$$\vDash_{FO} \mathsf{interp}(wp_{\text{sl}}(C, A)) \iff wp_{\text{ch}}(C, \mathsf{interp}(A))$$

The key to showing that our semantics for *TPL* correctly reflects that of Chalice is to show that the VCs generated for the `inhale` and `exhale` commands are equivalent. The `exhale` is straightforward.

Lemma 4.9. $\forall p.\ equiv(\texttt{exhale}\ p)$

Proof. By induction on p. □

The proof for `inhale` is more involved. This depends on the inhaled formula being syntactically self-framing. We define two Boogie commands, $\texttt{assertTPL}(A)$ to assert that a TPL assertion must hold at this point in the execution, and $\texttt{assertFrm}(p)$ to assert that a Chalice formula will be framed if it is inhaled in the current state.

Definition 4.10.

- $\texttt{assertTPL}(A) = \texttt{assert}(\mathsf{interp}(A))$
- $\texttt{assertFrm}(p) = \texttt{assertTPL}(\mathsf{sframed}(p))$

We lift the ability to curry and uncurry $\mathrel{-\!\!*}$ into the VC world. This is required to allow us to prove that an inhale of a $*$ can be replaced by two inhales, as Chalice does.

Lemma 4.11.

$$wp_{\text{ch}}(\mathit{assertFrm}(p_1), \mathsf{interp}(p_1 \mathrel{-\!\!*} (p_2 \mathrel{-\!\!*} A))) \iff wp_{\text{ch}}(\mathit{assertFrm}(p_1), \mathsf{interp}((p_1 * p_2) \mathrel{-\!\!*} A))$$

Proof. The left to right direction follows using Lemma 4.6:

$$\begin{aligned}
&wp_{\text{ch}}(\texttt{assertFrm}(p_1), \mathsf{interp}(p_1 \mathrel{-\!\!*} (p_2 \mathrel{-\!\!*} A))) \\
&\Rightarrow wp_{\text{ch}}(\texttt{assertFrm}(p_1), \mathsf{interp}(\mathsf{sframed}(p_1)) \wedge \mathsf{interp}(p_1 \mathrel{-\!\!*} (p_2 \mathrel{-\!\!*} A))) \\
&\Rightarrow wp_{\text{ch}}(\texttt{assertFrm}(p_1), \mathsf{interp}(\mathsf{sframed}(p_1) \wedge (p_1 \mathrel{-\!\!*} (p_2 \mathrel{-\!\!*} A)))) \\
&\Rightarrow wp_{\text{ch}}(\texttt{assertFrm}(p_1), \mathsf{interp}(\mathsf{sframed}(p_1) \wedge ((p_1 * p_2) \mathrel{-\!\!*} A)))
\end{aligned}$$

The reverse direction follows similarly. □

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 need 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.

We consider states in which, if we extend the environment to satisfy p, then p will be framed. In these states, asserting the formula $p \mathrel{-\!\!*} A$ and then inhaling p, is equivalent to inhaling p, and then asserting that A must hold.

Lemma 4.12.

$$\begin{aligned}
&wp_{\text{ch}}(\mathit{assertFrm}(p); \mathit{assertTPL}(p \mathrel{-\!\!*} A); \mathit{inhale}\ p, \phi) \\
&\qquad \iff wp_{\text{ch}}(\mathit{assertFrm}(p); \mathit{inhale}\ p; \mathit{assertTPL}(A), \phi)
\end{aligned}$$

Proof. We abbreviate $\texttt{assertFrm}(p)$ to $\texttt{assFrm}(p)$, and $\texttt{assertTPL}(A)$ to $\texttt{assTPL}(A)$.

By induction on p. We first consider the $*$ case:

$$\begin{aligned}
&wp_{\text{ch}}(\texttt{assFrm}(p_1 * p_2); \texttt{assTPL}((p_1 * p_2) \mathrel{-\!\!*} A); \texttt{inhale}\ p_1 * p_2, \phi) \\
&\iff wp_{\text{ch}}(\texttt{assFrm}(p_1 * p_2); \texttt{assFrm}(p_1); \texttt{assTPL}((p_1 * p_2) \mathrel{-\!\!*} A); \texttt{inhale}\ p_1 * p_2, \phi) \\
&\iff wp_{\text{ch}}(\texttt{assFrm}(p_1 * p_2); \texttt{assFrm}(p_1); \texttt{assTPL}(p_1 \mathrel{-\!\!*} p_2 \mathrel{-\!\!*} A); \texttt{inhale}\ p_1; \texttt{inhale}\ p_2, \phi) \\
&\iff wp_{\text{ch}}(\texttt{assFrm}(p_1); \texttt{assTPL}(p_1 \mathrel{-\!\!*} \mathsf{sframed}(p_2)); \texttt{inhale}\ p_1; \texttt{assTPL}(p_2 \mathrel{-\!\!*} A); \texttt{inhale}\ p_2, \phi) \\
&\iff wp_{\text{ch}}(\texttt{assFrm}(p_1); \texttt{inhale}\ p_1; \texttt{assFrm}(p_2); \texttt{assTPL}(p_2 \mathrel{-\!\!*} A); \texttt{inhale}\ p_2, \phi) \\
&\iff wp_{\text{ch}}(\texttt{assFrm}(p_1); \texttt{inhale}\ p_1; \texttt{assFrm}(p_2); \texttt{inhale}\ p_2; \texttt{assTPL}(A), \phi) \\
&\iff wp_{\text{ch}}(\texttt{assFrm}(p_1); \texttt{assTPL}(p_1 \mathrel{-\!\!*} \mathsf{sframed}(p_2)); \texttt{inhale}\ p_1; \texttt{inhale}\ p_2; \texttt{assTPL}(A), \phi) \\
&\iff wp_{\text{ch}}(\texttt{assFrm}(p_1 * p_2); \texttt{inhale}\ p_1 * p_2; \texttt{assTPL}(A), \phi)
\end{aligned}$$

For the access permission case, we can subdivide this into three further cases, (1) where the model contains no permission for $E.f$; (2) where the model contains more than 0, and less than or equal to $1 - \pi$ permission $E.f$; and (3) where the model contains more that

$1-\pi$ permission for $E.f$. The third case is trivial, so we just present the first two. First we consider the case $\mathcal{P}[\![E]\!].f] = 0$:

$$\begin{aligned}
&wp_{\text{ch}}(\texttt{assTPL}(acc(E.f,\pi) \mathrel{-\!\!*} A); \texttt{inhale}\ acc(E.f,\pi), \phi)\\
&\iff wp_{\text{ch}}(\texttt{assTPL}(acc(E.f,\pi) \mathrel{-\!\!*} A); \mathcal{P}[\![E]\!], f] = \pi; \texttt{havoc}\ \mathcal{H}[\![E]\!], f], \phi)\\
&\iff wp_{\text{ch}}(\texttt{inhale}\ acc(E.f,\pi); \texttt{assTPL}(A), \phi)
\end{aligned}$$

Second, we consider the case: $0 < \mathcal{P}[\![E]\!].f] \leq 1-\pi$,

$$\begin{aligned}
&wp_{\text{ch}}(\texttt{assTPL}(acc(E.f,\pi) \mathrel{-\!\!*} A); \texttt{inhale}\ acc(E.f,\pi), \phi)\\
&\iff wp_{\text{ch}}(\texttt{assTPL}(acc(E.f,\pi) \mathrel{-\!\!*} A); \mathcal{P}[\![E]\!], f] \mathrel{+}= \pi, \phi)\\
&\iff wp_{\text{ch}}(\texttt{inhale}\ acc(E.f,\pi); \texttt{assTPL}(A), \phi)
\end{aligned}$$

Finally, we present the implication case. We split this into two cases depending on whether the left of the implication holds. First we assume $[\![B]\!]$ holds in the current model:

$$\begin{aligned}
&wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{assTPL}((B \to p) \mathrel{-\!\!*} A); \texttt{inhale}(B \to p), \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{assTPL}(p \mathrel{-\!\!*} A); \texttt{inhale}\ p, \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{assTPL}(B \to \mathsf{sframed}(p)); \texttt{assTPL}(p \mathrel{-\!\!*} A); \texttt{inhale}\ p, \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{assFrm}(p); \texttt{assTPL}(p \mathrel{-\!\!*} A); \texttt{inhale}(p), \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{assFrm}(p); \texttt{inhale}(p); \texttt{assTPL}(A), \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{inhale}(p); \texttt{assTPL}(A), \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{inhale}(B \to p); \texttt{assTPL}(A), \phi)
\end{aligned}$$

and secondly, we assume $[\![B]\!]$ does not hold in the model:

$$\begin{aligned}
&wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{assTPL}((B \to p) \mathrel{-\!\!*} A); \texttt{inhale}(B \to p), \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{assTPL}(\mathsf{true} \mathrel{-\!\!*} A); \texttt{inhale}(\mathsf{true}), \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{assTPL}(\mathsf{true} \mathrel{-\!\!*} A), \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{assTPL}(A), \phi)\\
&\Leftrightarrow wp_{\text{ch}}(\texttt{assFrm}(B \to p); \texttt{inhale}(B \to p); \texttt{assTPL}(A), \phi)
\end{aligned}$$

□

Corollary 4.13. *If p is syntactically self-framing, then equiv($\texttt{inhale}\, p$).*

Proof. By the previous lemma, we know:

$$\begin{aligned}
&wp_{\text{ch}}(\texttt{assFrm}(p); \texttt{assTPL}(p \mathrel{-\!\!*} A); \texttt{inhale}\ p, \mathsf{true})\\
&\quad\iff wp_{\text{ch}}(\texttt{assFrm}(p); \texttt{inhale}\ p; \texttt{assTPL}(A), \mathsf{true})
\end{aligned}$$

As $wp_{\text{ch}}(\texttt{inhale}\ p, \mathsf{true}) \equiv \mathsf{true}$ and $wp_{\text{ch}}(\texttt{assTPL}(A), \mathsf{true}) \equiv \mathsf{interp}(A)$, we know

$$wp_{\text{ch}}(\texttt{assFrm}(p), \mathsf{interp}(p \mathrel{-\!\!*} A)) \iff wp_{\text{ch}}(\texttt{assFrm}(p); \texttt{inhale}\ p, \mathsf{interp}(A))$$

As p is syntactically self-framing we have

$$\mathsf{interp}(p \mathrel{-\!\!*} A) \iff wp_{\text{ch}}(\texttt{inhale}\ p, \mathsf{interp}(A))$$

By the definition of separation logic weakest pre-conditions, we have

$$\mathsf{interp}(wp_{\text{sl}}(\texttt{inhale}\ p, A)) \iff wp_{\text{ch}}(\texttt{inhale}\ p, \mathsf{interp}(A))$$

as required. □

Remark 4.14. Without the syntactic self-framing requirement on inhales, it would be unsound to break $\texttt{inhale}\ A_1 * A_2$ into $\texttt{inhale}\ A_1; \texttt{inhale}\ A_2$. In particular, in the Chalice semantics, the behaviour of $\texttt{inhale}(A_1 * A_2)$ and $\texttt{inhale}(A_2 * A_1)$ are different. For instance, consider $\texttt{inhale}(x.f = 3 * acc(x.f))$ and $\texttt{inhale}(acc(x.f) * x.f = 3)$.

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

The translation given by Smans *et al.* [18] 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`.

5. Mapping Separation Logic into Implicit Dynamic Frames

We are now in a position to draw together our various results, and show that SL-based verification can be simulated using IDF and Chalice. The overall approach is to show that, if one calculates weakest pre-conditions for a program using SL specifications, then there is a corresponding translated program in which one uses IDF specifications, and can calculate Chalice weakest pre-conditions which turn out to be equivalent to the SL ones. Just as in Section 3, we can use the projection of a total heap down to a permissions mask, to relate the evaluation of the two resulting assertions.

Just as in the preceding section, we focus our attention on inhale and exhale statements, since all commands which deal with changes to the footprint/permissions held in the state (e.g., method calls, fork/join of threads, acquire/release of locks) can be de-sugared down to these (other commands such as variable assignment can be treated uniformly in both worlds). Therefore, we aim to prove that the two different ways of calculating weakest pre-conditions produce equivalent results for both inhale and exhale statements.

Firstly, we state a few simple results which distribute equivalent assertions over various constructions.

Lemma 5.1 (Distributing Equivalences)**.** *For all TPL assertions A_1 and A_2 such that $A_1 \equiv_{TPL} A_2$, we have:*

(1) *For all TPL assertions A_3: $wp_{\text{sl}}(\texttt{inhale}(A_1), A_3) \equiv_{TPL} wp_{\text{sl}}(\texttt{inhale}(A_2), A_3)$ and similarly $wp_{\text{sl}}(\texttt{exhale}(A_1), A_3) \equiv_{TPL} wp_{\text{sl}}(\texttt{exhale}(A_2), A_3)$.*
(2) *The first-order assertions $\mathsf{interp}(A_1)$ and $\mathsf{interp}(A_2)$ are equivalent.*
(3) *For all TPL assertions A_3, the first-order assertions $wp_{\text{ch}}(\texttt{inhale}(A_3), \mathsf{interp}(A_1))$ and $wp_{\text{ch}}(\texttt{inhale}(A_3), \mathsf{interp}(A_2))$ are equivalent (and analogously for $\texttt{exhale}(A_3)$).*
(4) *For all TPL assertions A_3, the first-order assertion $\mathsf{interp}(wp_{\text{sl}}(\texttt{inhale}(A_1), A_3))$ is equivalent to $\mathsf{interp}(wp_{\text{sl}}(\texttt{inhale}(A_2), A_3))$. Similarly $\mathsf{interp}(wp_{\text{sl}}(\texttt{exhale}(A_1), A_3))$ is equivalent to $\mathsf{interp}(wp_{\text{sl}}(\texttt{exhale}(A_2), A_3))$.*

Proof. The first three parts follow straightforwardly from the corresponding definitions (and the fact that the definitions for weakest pre-conditions never inspect the post-condition). The fourth part is simply a combination of the first two. □

We now need to identify the fragment of separation logic which can be encoded into the syntax of Chalice (cf. Definition 2.4). This syntax roughly corresponds with the syntaxes supported by most separation-logic-based tools (in particular, no points-to predicates are permitted on the left of implications; a very common restriction which avoids needing to implement the full technical complexity of the connective; cf. Lemma 3.18). In order to avoid introducing further meta-variables to our notation, we will reuse the notation for full separation logic (a for assertions, e for expressions), but will clarify explicitly when we mean the restricted assertion syntax defined here.

Definition 5.2 (Restricted Separation Logic)**.** Expressions e, boolean expressions b and *restricted separation logic assertions* a are given by the following syntax definitions:

$$\begin{array}{lcl} e & ::= & x \mid \texttt{null} \mid n \\ b & ::= & e = e \mid e \neq e \mid b * b \\ a & ::= & b \mid e.f \overset{\pi}{\mapsto} e \mid a * a \mid b \rightarrow a \mid \exists v.\ e.f \overset{\pi}{\mapsto} v * a \end{array}$$

We allow a restricted form of existential in the syntax. It requires that the existential is witnessed by a particular field in the heap. This restriction is often implicit in tools for separation logic that support existentials. Without this restriction tools are typically incomplete.

We can represent the separation logic points-to predicate in terms of the Chalice accessibility predicate and a (heap-dependent) equality.

Proposition 5.3. *For all e,f,e',π we have $e.f \overset{\pi}{\mapsto} e' \equiv_{TPL} acc(e.f,\pi) * e.f = e'$.*

Proof. Directly from the semantics. □

Thus, we define the obvious translation from restricted separation logic assertions to those of Chalice:

Definition 5.4 (Mapping Restricted Separation Logic to Chalice)**.** We define a mapping $[a]$ from restricted separation logic assertions to Chalice assertions (cf. Definition 2.4), recursively as follows:

$$\begin{array}{rcl} [b] & = & b \\ [e_1.f \overset{\pi}{\mapsto} e_2] & = & (acc(e_1.f,\pi) * e_1.f = e_2) \\ [a_1 * a_2] & = & [a_1] * [a_2] \\ [b \rightarrow a] & = & b \rightarrow [a] \\ [\exists v.\ e.f \overset{\pi}{\mapsto} v * a] & = & acc(e.f,\pi) * [a][e.f/v] \end{array}$$

As the existential is witnessed by a particular heap location, in the translation to Chalice the existential can be eliminated by substituting the heap dependent expression. The translation preserves the semantics of the original assertion, which is a simple generalisation of Proposition 5.3:

Lemma 5.5 (Mapping to Chalice Preserves Semantics)**.** *For all restricted separation logic assertions a, we have $a \equiv_{TPL} [a]$.*

Proof. By straightforward induction on the structure of a, using Definition 3.7 and Proposition 5.3. The existential case uses Lemma 3.30. □

We can now combine the results of this section to relate the two different notions of weakest pre-conditions in combination with the appropriate translation from one assertion syntax to the other.

Theorem 5.6 (Weakest Pre-condition Calculations Equivalent)**.** *For any restricted SL assertion a, and any TPL assertion A, we have:*

$$\begin{array}{rcl} \vDash_{FO} \mathsf{interp}(wp_{\mathrm{sl}}(\mathit{inhale}(a), A)) & \iff & wp_{\mathrm{ch}}(\mathit{inhale}([a]), \mathsf{interp}(A)) \\ & \textit{and} & \\ \vDash_{FO} \mathsf{interp}(wp_{\mathrm{sl}}(\mathit{exhale}(a), A)) & \iff & wp_{\mathrm{ch}}(\mathit{exhale}([a]), \mathsf{interp}(A)) \end{array}$$

Proof. Consider the first case of the result (for inhale statements). By Lemma 5.1(4) and Lemma 5.5, we have that:

$$\vDash_{FO} \mathsf{interp}(wp_{\mathrm{sl}}(\mathtt{inhale}(a), A)) \iff \mathsf{interp}(wp_{\mathrm{sl}}(\mathtt{inhale}([a]), A))$$

By Corollary 4.13 (noting that $[a]$ is a syntactically self-framing Chalice assertion), we also know that:

$$\vDash_{FO} \mathsf{interp}(wp_{\mathrm{sl}}(\mathtt{inhale}([a]), A)) \iff wp_{\mathrm{ch}}(\mathtt{inhale}([a]), \mathsf{interp}(A))$$

Combining these two lines, we have the claimed result.

The case for exhale statements is analogous, using Lemma 4.9 instead of Corollary 4.13. □

Finally, we can draw together these results with the main result of Section 3, to show the equivalence of the two overall approaches.

Corollary 5.7 (Verifying Restricted Separation Logic in Chalice)**.** *For any restricted SL assertion a, and any (unrestricted) SL assertion a', and any environment σ, total heap H, and permission mask P, we have both:*

$$\begin{array}{rcl} (H \restriction P), \sigma & \vDash_{SL} & wp_{\mathrm{sl}}(\mathit{exhale}(a), a') \\ & \Leftrightarrow & \\ \sigma, \mathcal{H} \mapsto H, \mathcal{P} \mapsto P & \vDash_{FO} & wp_{\mathrm{ch}}(\mathit{exhale}([a]), \mathsf{interp}([a'])) \end{array}$$

and:

$$\begin{array}{rcl} (H \restriction P), \sigma & \vDash_{SL} & wp_{\mathrm{sl}}(\mathit{inhale}(a), a') \\ & \Leftrightarrow & \\ \sigma, \mathcal{H} \mapsto H, \mathcal{P} \mapsto P & \vDash_{FO} & wp_{\mathrm{ch}}(\mathit{inhale}([a]), \mathsf{interp}([a'])) \end{array}$$

Proof. By Theorem 5.6 and Definition 4.7, we have:

$$\begin{array}{rcl} H, P, \sigma \vDash_{TPL} wp_{\mathrm{sl}}(\mathtt{exhale}(a), a') & \Leftrightarrow & \sigma, \mathcal{H} \mapsto H, \mathcal{P} \mapsto P \vDash_{FO} wp_{\mathrm{ch}}(\mathtt{exhale}([a]), \mathsf{interp}(a')) \\ & & \textit{and} \\ H, P, \sigma \vDash_{TPL} wp_{\mathrm{sl}}(\mathtt{inhale}(a), a') & \Leftrightarrow & \sigma, \mathcal{H} \mapsto H, \mathcal{P} \mapsto P \vDash_{FO} wp_{\mathrm{ch}}(\mathtt{inhale}([a]), \mathsf{interp}(a')) \end{array}$$

By Lemmas 5.5 and 5.1(2), we have that the two assertions $\mathsf{interp}(a')$ and $\mathsf{interp}([a'])$ are equivalent. Therefore, by Lemma 5.1(3) and Theorem 3.20, we obtain the two desired equivalences. □

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.

6. Related Work

In this paper, we have provided a logic related to separation logic [6, 12], which allows arbitrary expressions over the heap. We have modified the standard presentation of an object-oriented heap for separation logic [14] to separate the notion of access from value (and thereby also relate to implicit dynamic frames [17]). 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 [10]. Benton and Leperchey [3] also provide a logic for sequential program reasoning that uses total heaps and maps defining which locations can be accessed.

Smans' original presentation of IDF was implemented in a tool, VeriCool [18, 17]. The results in this paper should also apply to the verification conditions generated by VeriCool. In recent work, Smans *et al.* [19] describe an IDF approach as a separation logic. However, they do not present a model of the assertions, just the VCs of their analogues to inhale and exhale. Hence, the work does not provide the strong connection between the VCs and the model of separation logic that we have provided. Vericool does however have a sound implementation of abstract predicates and pure functions (in fact, two different approaches; one for verification condition generation, as in [18], and one for symbolic execution, as in [19]). However, the approach for verification condition generation requires the formalisation of weakest pre-conditions in the presence of background axioms (used to define the meanings of predicates and pure methods). The use of these axioms cannot be summarised simply as part of a weakest pre-condition calculation, since the approach taken allows the prover to instantiate these axioms in an unbounded (and potentially non-terminating) way. Similarly, a comparison with a symbolic-execution-based approach would require more technical machinery than our current arguments based on first-order verification conditions. A comparison based on a runtime semantics for a language (as used to formalise soundness in [18]) might work better than one dealing with a verification semantics, but this is beyond the scope of our work.

There have been many other approaches based on dynamic frames [7, 8] to enable automated verification with standard verification tool chains; for instance, Dafny [16] and Region Logic [2]. Like Chalice, both also encode 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 future comparisons.

7. 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 *TPL* 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.

In addition, while the notions of minimal permission extensions and locally-havoced heap extensions are technically complex, it seems that the resulting semantics for the magic wand may actually simplify the problem of defining a suitable weakest-precondition semantics. This is because, whereas the classical semantics of the separation logic magic wand involves a quantification over states (which is problematic for encoding to a first-order prover), the semantics we present in this paper can, in the (common) case of supported assertions, eliminate the need for the quantifier altogether; we need only check the unique, minimal extension of the initial state to make the left-hand side true, if such an extension exists. Exploring the practical consequences of these observations will be interesting future work.

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, along with other Chalice contributors, 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 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, and typically use logical variables to connect old and new values of heap locations. 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. Consider the following two specifications, where the left one uses old expressions and the right one a logical variable v:

requires $acc(x.f)$	requires $acc(x.f) * x.f = v$
ensures $acc(x.f) * x.f = old(x.f) + 1$	ensures $acc(x.f) * x.f = v + 1$

To use the logical variable specification, we must find a witness for the logical variable v, while with old expressions this witness is not required as it simply places a constraint on the possible old and new heaps. This is because the assertions describing the value relationship in the old expression specification only appears in the post-condition, which, from the caller's perspective, ends up as an assume. On the other hand, in the logical variable alternative, the variable appears both in the pre- and post-conditions, hence it also ends up used in an assert (when the caller exhales the pre-condition). Moving to old expressions may have benefits for building tool support for separation logic.

Acknowledgements. We thank Mike Dodds, David Naumann, Ioannis Kassios, Peter Müller, Sophia Drossopoulou and the anonymous ESOP and LMCS reviewers for feedback on this work.

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] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, 2005.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.

[5] J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.

[6] 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.

[7] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.

[8] I. T. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, 2006.

[9] K. R. M. Leino. This is Boogie 2. Available from `http://research.microsoft.com/en-us/um/people/leino/papers.html`.

[10] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.

[11] P. W. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.

[12] 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.

[13] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.

[14] M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, November 2005.

[15] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. In G. Barthe, editor, *European Symposium on Programming (ESOP)*, volume 6602 of *Lecture Notes in Computer Science*, pages 439–458. Springer-Verlag, 2011.

[16] K. Rustan and M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

[17] 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.

[18] 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.

[19] J. Smans, B. Jacobs, and F. Piessens. Heap-dependent expressions in separation logic. In *FMOODS/FORTE*, pages 170–185, 2010.

Appendix A. Proofs

Lemma 3.9 (Minimisation of Permission Masks)**.** *If* $H, P_1 * P_2, \sigma \vDash_{TPL} A$ *then* $\exists P_3 \subseteq P_2$ *such that* $(H, P_1, \sigma) \lhd P_3 \vDash_{TPL} A$.

Proof. By complete (strong) induction on $rds(P_2)$ using subset ordering.

Case split on whether the following formula holds:

$$\exists P_4 \subseteq P_2.\ rds(P_4) \subset rds(P_2) \wedge H, P_1 * P_4, \sigma \vDash_{TPL} A$$

Assume first that it does. By induction, we know $\exists P_5 \subseteq P_4.\ (H, P_1, \sigma) \lhd P_5 \vDash_{TPL} A$, and by transitivity of $\subseteq$, we have $P_5 \subseteq P_2$, as required.

Secondly, assume that the formula does not hold. Therefore,

$$\forall P_4 \subseteq P_2.\ rds(P_4) \subset rds(P_2) \Rightarrow H, P_1 * P_4, \sigma \nvDash_{TPL} A$$

Hence, $(H, P_1, \sigma) \lhd P_2 \vDash_{TPL} A$. We can prove the obligation by picking $P_3 = P_2$. □

Lemma 3.11. *If* A *is weakening-closed,* $(H, P, \sigma) \lhd P' \vDash_{TPL} A$, $P' \subseteq P''$ *and* $rds(P') = rds(P'')$*, then* $(H, P, \sigma) \lhd P'' \vDash_{TPL} A$

Proof. As A is weakening-closed, we know:

$$H, P * P'', \sigma \vDash_{TPL} A$$

If P'' is not minimal, then neither was P', which is a contradiction. So P'' must also be a minimal extension. □

Lemma 3.12 (Minimal Permission Extensions Closed)**.** *If* $(H, P_1 * P_2, \sigma) \lhd P_3 \vDash_{TPL} A$ *and* A *is weakening-closed, then* $\exists P_4 \subseteq P_2$ *and* $(H, P_1, \sigma) \lhd P_4 * P_3 \vDash_{TPL} A$.

Proof. First, we prove a more general result, and apply Lemma 3.11 to get the required result. We prove, that:

> If we know that A is weakening-closed, then, if we also have $(H, P_1 * P_2, \sigma) \lhd P_3 \vDash_{TPL} A$ then $\exists P_4 \subseteq P_2$ and $P_5 \subseteq P_3$ such that $rds(P_5) = rds(P_3)$ and $(H, P_1, \sigma) \lhd P_4 * P_5 \vDash_{TPL} A$.

We know $H, P_1 * P_2 * P_3, \sigma \vDash_{TPL} A$, and by Lemma 3.9, we have that there exists P_6 such that $(H, P_1, \sigma) \lhd P_6 \vDash_{TPL} A$ and $P_6 \subseteq P_2 * P_3$. Choose P_4 and P_5 such that $P_5 = P_3 \sqcap P_6$, and $P_4 = P_6 - (P_3 \sqcap P_6)$.

We need to show $P_4 \subseteq P_2$ and $rds(P_5) = rds(P_3)$. From our assumptions, we know that $P_4 * (P_3 \sqcap P_6) = P_6$, and thus $P_4 \subseteq P_6$, and $P_6 \subseteq P_2 * P_3$. We split into two cases:

($P_4 \nsubseteq P_2$): Therefore, there exists (ι, f) such that $P_4[\iota, f] > P_2[\iota, f]$. Therefore, $P_6[\iota, f] > P_2[\iota, f]$, but by assumption, we know $P_6[\iota, f] \leq P_2[\iota, f] + P_3[\iota, f]$. Consider two sub-cases:

- ($P_6[\iota, f] < P_3[\iota, f]$): Therefore, as $P_4 = P_6 - (P_3 \sqcap P_6)$ we know $P_4[\iota, f] = 0$ contradicting $P_4[\iota, f] > P_2[\iota, f]$.
- ($P_3[\iota, f] \leq P_6[\iota, f]$): Therefore, as $P_4 * (P_3 \sqcap P_6) = P_6$ we know $P_4[\iota, f] + P_3[\iota, f] = P_6[\iota, f]$. As $P_6 \subseteq P_2 * P_3$, we know $P_4[\iota, f] + P_3[\iota, f] \leq P_2[\iota, f] + P_3[\iota, f]$, hence $P_4[\iota, f] \leq P_2[\iota, f]$, contradicting assumption.

($P_4 \subseteq P_2$): We split into two sub-cases:

- ($rds(P_5) = rds(P_3)$): The result follows directly.
- ($rds(P_5) \subset rds(P_3)$): For this case, using the assumption that A is weakening-closed, we get $H, P_1 * P_2 * P_5, \sigma \vDash_{TPL} A$, which is in contradiction with the assumption that $(H, P_1 * P_2, \sigma) \lhd P_3 \vDash_{TPL} A$. □

Proposition 3.13. *All formulas A are weakening-closed.*

Proof. By induction on A. By $P \subseteq P'$ we can assume there exists P'' such that $P * P'' = P'$.
($A \equiv acc(E.f, \pi)$):

$$\begin{array}{ll} & H, P, \sigma \vDash_{TPL} A \\ \Leftrightarrow & P[\![E]\!]_{H,\sigma}, f] \geq \pi \qquad \textit{(by defn)} \\ \Rightarrow & P'[\![E]\!]_{H,\sigma}, f] \geq \pi \quad \textit{(as } P \subseteq P'\textit{)} \\ \Leftrightarrow & H, P', \sigma \vDash_{TPL} A \end{array}$$

($A \equiv E = E'$):

$$\begin{array}{ll} & H, P, \sigma \vDash_{TPL} A \\ \Leftrightarrow & [\![E]\!]_{H,\sigma} = [\![E']\!]_{H,\sigma} \quad \textit{(by defn)} \\ \Leftrightarrow & H, P', \sigma \vDash_{TPL} A \end{array}$$

($A \equiv E.f \mapsto E'$): Follows using a combination of arguments from previous two cases.
($A \equiv A_1 * A_2$):

$$\begin{array}{ll} & H, P, \sigma \vDash_{TPL} A \\ \Rightarrow & \exists P_1, P_2.\ P_1 * P_2 = P \wedge H, P_1, \sigma \vDash_{TPL} A_1 \wedge H, P_2, \sigma \vDash_{TPL} A_2 \end{array}$$

We introduce P_1 and P_2, and define $P_3 = P_2 * P''$. By induction, we know

$$\begin{array}{ll} \Rightarrow & P_1 * P_2 = P \wedge H, P_1, \sigma \vDash_{TPL} A_1 \wedge H, P_3, \sigma \vDash_{TPL} A_2 \\ \Rightarrow & P_1 * P_3 = P' \wedge H, P_1, \sigma \vDash_{TPL} A_1 \wedge H, P_3, \sigma \vDash_{TPL} A_2 \\ \Rightarrow & H, P', \sigma \vDash_{TPL} A \qquad \textit{(by defn)} \end{array}$$

($A \equiv A_1 \wedge A_2$, $A \equiv A_1 \vee A_2$): Trivially by induction.
($A \equiv A_1 \mathrel{-\!\!*} A_2$): By unfolding the obligation, we can assume

$$\begin{array}{l} H, P, \sigma \vDash_{TPL} A_1 \mathrel{-\!\!*} A_2 \\ P_1 \perp P' \\ H_1 \overset{rds(P') \cup \overline{rds(P_1)}}{\equiv} H \\ (H_1, \varnothing, \sigma) \lhd P_1 \vDash_{TPL} A_1 \end{array}$$

and must prove

$$H_1, P' * P_1, \sigma \vDash_{TPL} A_2$$

By assumptions, we know $P_1 \perp P$ and $H_1 \overset{rds(P) \cup \overline{rds(P_1)}}{\equiv} H$ as P is smaller than P'. Therefore, using $\mathrel{-\!\!*}$ assumption, we have

$$H_1, P * P_1, \sigma \vDash_{TPL} A_2$$

By inductive hypothesis, we know

$$H_1, P' * P_1, \sigma \vDash_{TPL} A_2$$

as required.
($A \equiv A_1 \to A_2$): By unfolding the obligation, we can assume

$$\begin{array}{l} H, P, \sigma \vDash_{TPL} A_1 \to A_2 \\ P_2 \perp P' \\ H_2 \overset{rds(P') \cup \overline{rds(P_2)}}{\equiv} H \\ (H_2, P', \sigma) \lhd P_2 \vDash_{TPL} A_1 \end{array}$$

and must prove

$$H_2, P' * P_2, \sigma \vDash_{TPL} A_2$$

By Lemma 3.12, and as A_1 is closed under permission extension by inductive hypothesis, we have that there exists a P_1 such that

$$P_1 \subseteq P''$$
$$(H_2, P, \sigma) \lhd P_1 * P_2 \vDash_{TPL} A_1$$

As $P \subseteq P'$, we can show

$$\begin{array}{lll} & H_2 \overset{rds(P') \cup \overline{rds(P_2)}}{\equiv} H & \\ \Rightarrow & H_2 \overset{rds(P) \cup \overline{rds(P_2)}}{\equiv} H & \textit{(as } P \subseteq P'\textit{)} \\ \Rightarrow & H_2 \overset{rds(P) \cup \overline{rds(P_2 * P_1)}}{\equiv} H & \textit{(as } P_2 \subseteq P_1 * P_2\textit{)} \end{array}$$

Therefore by $\rightarrow$ assumption, we have

$$H_2, P * P_1 * P_2, \sigma \vDash_{TPL} A_2$$

As $P_1 \subseteq P''$, by inductive hypothesis we have

$$H_2, P * P'' * P_2, \sigma \vDash_{TPL} A_2$$

as required.

($A \equiv \exists x.\ A'$):

$$\begin{array}{l} H, P, \sigma \vDash_{TPL} \exists x.\ A' \\ \Rightarrow \exists v.\ H, P, \sigma[x \mapsto v] \vDash_{TPL} A' \\ \Rightarrow \exists v.\ H, P', \sigma[x \mapsto v] \vDash_{TPL} A' \ \textit{(by inductive hypothesis)} \\ \Rightarrow H, P', \sigma \vDash_{TPL} \exists x.\ A' \end{array}$$

Lemma 3.15.

(1) *If* $(H, P, \sigma) \in \mathsf{ExtFrm}(A)$*, then:*
   (a) *if* $H' \in \mathsf{interfere}(H, P)$*, then* $(H', P, \sigma) \in \mathsf{ExtFrm}(A)$.
   (b) *if* $P' \perp P$ *and* $H' \in \mathsf{interfere}(H, P * P')$ *and* $H, P * P', \sigma \vDash_{TPL} A$*, then* $H', P * P', \sigma \vDash_{TPL} A$.

(2) *If* $(H, P, \sigma) \in \mathsf{DisExtFrm}(A)$*, then:*
   (a) *if* $H' \in \mathsf{interfere}(H, P)$*, then* $(H', P, \sigma) \in \mathsf{DisExtFrm}(A)$.
   (b) *if* $P' \perp P$ *and* $H' \in \mathsf{interfere}(H, P * P')$ *and* $H, P', \sigma \vDash_{TPL} A$*, then* $H', P', \sigma \vDash_{TPL} A$.

Proof.

(1) (a) This follows directly, since if we have $H' \in \mathsf{interfere}(H, P)$, then it follows that $\mathsf{globalExts}(H, P, \sigma) = \mathsf{globalExts}(H', P, \sigma)$.
   (b) By assumptions, we know that $(H, P * P', \sigma) \in \mathsf{globalExts}(H, P, \sigma)$, and also that $H' \in \mathsf{interfere}(H, P * P')$. As $\mathsf{globalExts}(H, P, \sigma) \cap \langle\!\langle A \rangle\!\rangle$ is stable, we know $H', P * P', \sigma \vDash_{TPL} A$ as required.

(2) (a) This follows directly, since if $H' \in \mathsf{interfere}(H, P)$, then $\mathsf{globalDisjExts}(H, P, \sigma) = \mathsf{globalDisjExts}(H', P, \sigma)$.
   (b) By assumptions, we know that $(H, P', \sigma) \in \mathsf{globalDisjExts}(H, P, \sigma)$, and also that $H' \in \mathsf{interfere}(H, P * P')$. As $\mathsf{globalDisjExts}(H, P, \sigma) \cap \langle\!\langle A \rangle\!\rangle$ is stable with P, we know $H', P', \sigma \vDash_{TPL} A$ as required.

Lemma 3.16 (Preservation of Minimal Extensions)**.**

(1) *For all* $(H_1, P_1, \sigma) \in \mathsf{ExtFrm}(A)$,

$$\forall P_2 \perp P_1, \forall H_2 \overset{P_1 * P_2}{\equiv} H_1.((H_1, P_1, \sigma) \triangleleft P_2 \vDash_{TPL} A \Rightarrow (H_2, P_1, \sigma) \triangleleft P_2 \vDash_{TPL} A)$$

(2) *For all* $(H_1, P_1, \sigma) \in \mathsf{DisExtFrm}(A)$,

$$\forall P_2 \perp P_1, \forall H_2 \overset{P_1 * P_2}{\equiv} H_1.((H_1, \varnothing, \sigma) \triangleleft P_2 \vDash_{TPL} A \Rightarrow (H_2, \varnothing, \sigma) \triangleleft P_2 \vDash_{TPL} A)$$

(3) *If* A *is self-framing and* $P_2 \perp P_1$ *and* $(H_1, P_1, \sigma) \triangleleft P_2 \vDash_{TPL} A$ *and* $H_2 \overset{P_1 * P_2}{\equiv} H_1$*, then* $(H_2, P_1, \sigma) \triangleleft P_2 \vDash_{TPL} A$

Proof.

(1) Assume $P_2 \perp P_1$, $H_2 \overset{P_1 * P_2}{\equiv} H_1$ and $(H_1, P_1, \sigma) \triangleleft P_2 \vDash_{TPL} A$. By $(H_1, P_1, \sigma) \in \mathsf{ExtFrm}(A)$ we have $H_2, P_1 * P_2, \sigma \vDash_{TPL} A$, and thus we are left to prove:

$$\forall P_3 \subseteq P_2.\ rds(P_3) \subset rds(P_2) \Rightarrow H_2, P_1 * P_3, \sigma \nvDash_{TPL} A$$

We assume $P_3 \subseteq P_2$, $rds(P_3) \subset rds(P_2)$ and $H_2, P_1 * P_3, \sigma \vDash_{TPL} A$ and seek a contradiction. By Lemma 3.15(1)(b) we can prove $H_1, P_1 * P_3, \sigma \vDash_{TPL} A$, and thus using $(H_1, P_1, \sigma) \triangleleft P_2 \vDash_{TPL} A$ we deduce a contradiction.

(2) Assume $P_2 \perp P_1$, $H_2 \overset{P_1 * P_2}{\equiv} H_1$ and $(H_1, \varnothing, \sigma) \triangleleft P_2 \vDash_{TPL} A$. By $(H_1, P_1, \sigma) \in \mathsf{DisExtFrm}(A)$ we have $H_2, P_2, \sigma \vDash_{TPL} A$, and thus we are left to prove:

$$\forall P_3 \subseteq P_2.\ rds(P_3) \subset rds(P_2) \Rightarrow H_2, P_3, \sigma \nvDash_{TPL} A$$

We assume $P_3 \subseteq P_2$, $rds(P_3) \subset rds(P_2)$ and $H_2, P_3, \sigma \vDash_{TPL} A$ and seek a contradiction. By Lemma 3.15(2)(b) we can prove $H_1, P_3, \sigma \vDash_{TPL} A$, and thus using $(H_1, \varnothing, \sigma) \triangleleft P_2 \vDash_{TPL} A$ we deduce a contradiction.

(3) Since A is self-framing, we know $H_2, P_1 * P_2, \sigma \vDash_{TPL} A$, and thus we are left to prove:

$$\forall P_3 \subseteq P_2.\ rds(P_3) \subset rds(P_2) \Rightarrow H_2, P_1 * P_3, \sigma \nvDash_{TPL} A$$

We assume $P_3 \subseteq P_2$, $rds(P_3) \subset rds(P_2)$ and $H_2, P_1 * P_3, \sigma \vDash_{TPL} A$ and seek a contradiction. By self-framing assumption, we know $H_1, P_1 * P_3, \sigma \vDash_{TPL} A$, but this contradicts initial minimality assumption.

Lemma 3.19 (Simplified Semantics for Self-Framing Conditionals)**.**

(1) *If* A_1 *and* A_2 *are both self-framing, then:*

(a) $H, P, \sigma \vDash_{TPL} A_1 \to A_2$ *if and only if:*

$$\forall (H', P', \sigma) \in \mathsf{localExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1 \ \Rightarrow\ H', P', \sigma \vDash_{TPL} A_2)$$

(b) $H, P, \sigma \vDash_{TPL} A_1 \to A_2$ *if and only if:*

$$\forall (H', P', \sigma) \in \mathsf{globalExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1 \ \Rightarrow\ H', P', \sigma \vDash_{TPL} A_2)$$

(2) *If* A_1 *and* A_2 *are both self-framing, then:*

(a) $H, P, \sigma \vDash_{TPL} A_1 \mathrel{-\!\!*} A_2$ *if and only if:*

$$\forall (H', P', \sigma) \in \mathsf{localDisjExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1 \ \Rightarrow\ H', P * P', \sigma \vDash_{TPL} A_2)$$

(b) $H, P, \sigma \vDash_{TPL} A_1 \mathrel{-\!\!*} A_2$ *if and only if:*

$$\forall (H', P', \sigma) \in \mathsf{globalDisjExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1 \ \Rightarrow\ H', P * P', \sigma \vDash_{TPL} A_2)$$

Proof.

(1) (a) We need to show that:

$$\left(\begin{array}{l}\forall P_1, H_1.(\ (H_1, P * P_1, \sigma) \in \mathsf{localExts}(H, P, \sigma) \wedge\ (H_1, P, \sigma) \lhd P_1 \vDash_{TPL} A_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow H_1, P * P_1, \sigma \vDash_{TPL} A_2)\end{array}\right)$$
$$\Leftrightarrow$$
$$\left(\begin{array}{l}\forall P_2, H_2.(\ (H_2, P * P_2, \sigma) \in \mathsf{localExts}(H, P, \sigma) \wedge\ H_2, P * P_2, \sigma \vDash_{TPL} A_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow H_2, P * P_2, \sigma \vDash_{TPL} A_2)\end{array}\right)$$

The right-to-left direction is easy, since the left-hand formula requires that we check the implication in strictly fewer states (only those which are obtained via minimal extensions). For the left-to-right direction, assume that for some arbitrary P_2, H_2 we have $P_2 \perp P$ and $H_2 \overset{rds(P) \cup \overline{rds(P_2)}}{\equiv} H$ and $H_2, P * P_2, \sigma \vDash_{TPL} A_1$. Then we need to show that: $H_2, P * P_2, \sigma \vDash_{TPL} A_2$. By Lemma 3.9, there exists $P_3 \subseteq P_2$ such that $(H_2, P, \sigma) \lhd P_3 \vDash_{TPL} A_1$. Define $H_3 = (P * P_3\ ?\ H_2 : H)$. Then, by construction, $H_3 \overset{P * P_3}{\equiv} H_2$ and $H_3 \overset{rds(P) \cup \overline{rds(P_3)}}{\equiv} H$.
By Lemma 3.16 (3), since A_1 is self-framing, we have $(H_3, P, \sigma) \lhd P_3 \vDash_{TPL} A_1$. Now, using the assumption from the left-hand-side of our overall goal, choose $P_1 = P_3$ and $H_1 = H_3$, and we obtain $H_3, P * P_3, \sigma \vDash_{TPL} A_2$. Since A_2 is self-framing, we have $H_2, P * P_3, \sigma \vDash_{TPL} A_2$. Then, by Proposition 3.13, we obtain $H_2, P * P_2, \sigma \vDash_{TPL} A_2$ as required.

(b) By the previous part, it suffices to show that :

$$\forall (H', P', \sigma) \in \mathsf{localExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1\ \Rightarrow\ H', P', \sigma \vDash_{TPL} A_2)$$
$$\Leftrightarrow$$
$$\forall (H', P', \sigma) \in \mathsf{globalExts}(H, P, \sigma).\ (H', P', \sigma \vDash_{TPL} A_1\ \Rightarrow\ H', P', \sigma \vDash_{TPL} A_2)$$

The $(\Leftarrow)$ direction is immediate, since $\mathsf{localExts}(H, P, \sigma) \subseteq \mathsf{globalExts}(H, P, \sigma)$. To show the $(\Rightarrow)$ direction, we assume the former formula, and suppose that we have some $(H', P', \sigma) \in \mathsf{globalExts}(H, P, \sigma)$ such that $H', P', \sigma \vDash_{TPL} A_1$ holds. Define $H'' = (P'\ ?\ H' : H)$. By construction, $(H'', P', \sigma) \in \mathsf{localExts}(H, P, \sigma)$ and $H'' \overset{P'}{\equiv} H'$. Since A_1 is self-framing, we conclude that $H', P', \sigma \vDash_{TPL} A_1$ holds. Therefore, by the assumed formula, we can conclude that $H'', P', \sigma \vDash_{TPL} A_2$ is true. Since A_2 is self-framing, we conclude $H', P', \sigma \vDash_{TPL} A_2$ as required.

(2) (a) We need to show that:

$$\left(\begin{array}{l}\forall P_1, H_1.(\ P_1 \perp P\ \wedge H_1 \overset{rds(P) \cup \overline{rds(P_1)}}{\equiv} H\ \wedge\ (H_1, \varnothing, \sigma) \lhd P_1 \vDash_{TPL} A_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow H_1, P * P_1, \sigma \vDash_{TPL} A_2)\end{array}\right)$$
$$\Leftrightarrow$$
$$\left(\begin{array}{l}\forall P_2, H_2.(\ P_2 \perp P\ \wedge H_2 \overset{rds(P) \cup \overline{rds(P_2)}}{\equiv} H\ \wedge\ H_2, P_2, \sigma \vDash_{TPL} A_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow H_2, P * P_2, \sigma \vDash_{TPL} A_2)\end{array}\right)$$

The right-to-left direction is easy, since the left-hand formula requires that we check the implication in strictly fewer states (only those which are obtained via minimal extensions). For the left-to-right direction, assume that for some arbitrary P_2, H_2 we have $P_2 \perp P$ and $H_2 \overset{rds(P) \cup \overline{rds(P_2)}}{\equiv} H$ and $H_2, P_2, \sigma \vDash_{TPL} A_1$. Then we need to show that: $H_2, P * P_2, \sigma \vDash_{TPL} A_2$. By Lemma 3.9, there exists $P_3 \subseteq P_2$ such that

$(H_2, \varnothing, \sigma) \lhd P_3 \vDash_{TPL} A_1$. Define $H_3 = (P * P_3 \ ? \ H_2 : H)$. Then, by construction, $H_3 \overset{P*P_3}{\equiv} H_2$ and $H_3 \overset{rds(P) \cup \overline{rds(P_3)}}{\equiv} H$.

By Lemma 3.16 (3), since A_1 is self-framing, we have $(H_3, \varnothing, \sigma) \lhd P_3 \vDash_{TPL} A_1$. Now, using the assumption from the left-hand-side of our overall goal, choose $P_1 = P_3$ and $H_1 = H_3$, and we obtain $H_3, P * P_3, \sigma \vDash_{TPL} A_2$. Since A_2 is self-framing, we have $H_2, P * P_3, \sigma \vDash_{TPL} A_2$. Then, by Proposition 3.13, we obtain $H_2, P * P_2, \sigma \vDash_{TPL} A_2$ as required.

(b) By similar argument to part (1)(b).

Theorem 3.20 (Correctness of Total Heap Semantics)**.** *For all SL-assertions a, environments σ, total heaps H, and permission masks P:*

$$H, P, \sigma \vDash_{TPL} a \iff (H \restriction P), \sigma \vDash_{SL} a$$

Proof. By induction on a. First note, if the property holds of an *SL*-assertion, then the assertion is self-framing. Thus, inductively we can assume all sub-assertions are self-framing.

$(a \equiv e.f \overset{\pi}{\mapsto} e')$:

$$\begin{array}{lll} & H, P, \sigma \vDash_{TPL} a & \\ \Leftrightarrow & P[[\![e]\!]_{\sigma,H}, f] \geq \pi \ \wedge \ H[[\![e]\!]_{\sigma,H}, f] = [\![e']\!]_{\sigma,H} & \textit{(by defn.)} \\ \Leftrightarrow & ([\![e]\!]_{\sigma,H}, f) \in dom(H \restriction P) \ \wedge \ \downarrow_1((H \restriction P)[[\![e]\!]_{\sigma,H}, f]) = [\![e']\!]_{\sigma,H} \ \wedge & \\ & \downarrow_2((H \restriction P)[[\![e]\!]_{\sigma,H}, f]) \geq \pi & \textit{(by defn. of } (H \restriction P)\textit{)} \\ \Leftrightarrow & ([\![e]\!]_{\sigma}, f) \in dom(H \restriction P) \ \wedge \ \downarrow_1((H \restriction P)[[\![e]\!]_{\sigma}, f]) = [\![e']\!]_{\sigma} \ \wedge & \\ & \downarrow_2((H \restriction P)[[\![e]\!]_{\sigma}, f]) \geq \pi & \textit{(by Lemma 3.8)} \\ \Leftrightarrow & (H \restriction P), \sigma \vDash_{SL} a & \textit{(by defn.)} \end{array}$$

$(a \equiv e{=}e')$:

$$\begin{array}{lll} & H, P, \sigma \vDash_{TPL} a & \\ \Leftrightarrow & [\![e]\!]_{\sigma,H} = [\![e']\!]_{\sigma,H} & \textit{(by defn.)} \\ \Leftrightarrow & [\![e]\!]_{\sigma} = [\![e']\!]_{\sigma} & \textit{(by Lemma 3.8)} \\ \Leftrightarrow & (H \restriction P), \sigma \vDash_{SL} a & \textit{(by defn.)} \end{array}$$

$(a \equiv a_1 * a_2)$:

$$\begin{array}{lll} & H, P, \sigma \vDash_{TPL} a & \\ \Leftrightarrow & \exists P_1, P_2.(P = P_1 * P_2 \ \wedge & \\ & H, P_1, \sigma \vDash_{TPL} a_1 \ \wedge \ H, P_2, \sigma \vDash_{TPL} a_2) & \textit{(by defn.)} \\ \Leftrightarrow & \exists P_1, P_2.(P = P_1 * P_2 \ \wedge & \\ & (H \restriction P_1), \sigma \vDash_{SL} a_1 \ \wedge \ (H \restriction P_2), \sigma \vDash_{SL} a_2) & \textit{(by induction, twice)} \\ \Leftrightarrow & (H \restriction P), \sigma \vDash_{SL} a & \textit{(by defn.)} \end{array}$$

$(a \equiv a_1 \wedge a_2)$,$(a \equiv a_1 \vee a_2)$: Straightforwardly, by induction.

$(a \equiv a_1 \rightarrow a_2)$:

$$
\begin{array}{ll}
& H, P, \sigma \vDash_{TPL} a \\
\Leftrightarrow & \forall P_1, H_1.(\ P_1 \perp P\ \wedge\ H_1 \overset{rds(P)\cup\overline{rds(P_1)}}{\equiv} H\ \wedge \\
& H_1, P * P_1, \sigma \vDash_{TPL} a_1\ \Rightarrow\ H_1, P * P_1, \sigma \vDash_{TPL} a_2) \\
& \qquad\qquad \textit{(by Lemma 3.19 (1))} \\
\Leftrightarrow & \forall P_1, H_1.(\ P_1 \perp P\ \wedge\ H_1 \overset{rds(P)\cup\overline{rds(P_1)}}{\equiv} H\ \wedge \\
& (H_1 \restriction (P * P_1)), \sigma \vDash_{SL} a_1\ \Rightarrow\ (H_1 \restriction (P * P_1)), \sigma \vDash_{SL} a_2) \\
& \qquad\qquad \textit{(by induction, twice)} \\
\Leftrightarrow & \forall P_1, H_1.(\ P_1 \perp P\ \wedge\ H_1 \overset{rds(P)\cup\overline{rds(P_1)}}{\equiv} H\ \wedge \\
& ((H_1 \restriction P) * (H_1 \restriction P_1)), \sigma \vDash_{SL} a_1\ \Rightarrow\ ((H_1 \restriction P) * (H_1 \restriction P_1)), \sigma \vDash_{SL} a_2) \\
& \qquad\qquad \textit{(by defn.)} \\
\Leftrightarrow & \forall P_1, H_1.(\ P_1 \perp P\ \wedge\ H_1 \overset{rds(P)\cup\overline{rds(P_1)}}{\equiv} H\ \wedge \\
& ((H \restriction P) * (H_1 \restriction P_1)), \sigma \vDash_{SL} a_1\ \Rightarrow\ ((H \restriction P) * (H_1 \restriction P_1)), \sigma \vDash_{SL} a_2) \\
& \qquad\qquad \textit{(since } H_1 \overset{P}{\equiv} H\textit{)} \\
\Leftrightarrow & \forall P_1, h_1.(\ P_1 \perp dom(H \restriction P)\ \wedge\ dom(h_1) = rds(P_1)\ \wedge \\
& (\forall (\iota, f) \in (dom(h_1) \cap dom(H \restriction P)).h_1[\iota, f] = (H \restriction P)[\iota, f])\ \wedge \\
& ((H \restriction P) * h_1), \sigma \vDash_{SL} a_1\ \Rightarrow\ ((H \restriction P) * h_1), \sigma \vDash_{SL} a_2)) \\
& \qquad\qquad \textit{(by defn. of } (H_1 \restriction P_1)\textit{)} \\
\Leftrightarrow & \forall h_1.(\ h_1 \perp (H \restriction P)\ \wedge \\
& ((H \restriction P) * h_1), \sigma \vDash_{SL} a_1\ \Rightarrow\ ((H \restriction P) * h_1), \sigma \vDash_{SL} a_2) \\
& \qquad\qquad \textit{(by defn. of } h_1 \perp (H \restriction P)\textit{)} \\
\Leftrightarrow & (H \restriction P), \sigma \vDash_{SL} a_1 \rightarrow a_2
\end{array}
$$

$(a \equiv a_1 \mathrel{-\!\!*} a_2)$: Analogous to previous case, using Lemma 3.19 (2) instead of Lemma 3.19 (1).
$(a \equiv \exists x.\ a')$: We have:

$$
\begin{array}{ll}
& H, P, \sigma \vDash_{TPL} \exists x.\ a' \\
\Leftrightarrow & \exists v.\ H, P, \sigma[x \mapsto v] \vDash_{TPL} a' \\
\Leftrightarrow & \exists v.\ H \restriction P, \sigma[x \mapsto v] \vDash_{SL} a'\ \textit{by inductive hypothesis} \\
\Leftrightarrow & H \restriction P, \sigma \vDash_{SL} \exists x.\ a'
\end{array}
$$

Lemma 3.23 (Decomposing Minimal Permission Extensions over Conjunctions)**.**

(1) *If* $(H, \varnothing, \sigma) \triangleleft P' \vDash_{TPL} A_1 * A_2$ *then* $\exists P_1, P_2$ *such that* $P' = P_1 * P_2$ *and* $(H, \varnothing, \sigma) \triangleleft P_1 \vDash_{TPL} A_1$ *and* $(H, \varnothing, \sigma) \triangleleft P_2 \vDash_{TPL} A_2$.

(2) *If* $(H, P, \sigma) \triangleleft P' \vDash_{TPL} A_1 \wedge A_2$ *then* $\exists P_1, P_2$ *such that* $P' = P_1 * P_2$ *and* $(H, P, \sigma) \triangleleft P_1 \vDash_{TPL} A_1$ *and* $(H, P * P_1, \sigma) \triangleleft P_2 \vDash_{TPL} A_2$.

Proof.

(1) We prove an equivalent statement:

$$
\begin{array}{l}
(H, \varnothing, \sigma) \triangleleft P \vDash_{TPL} A_1 * A_2\ \wedge\ P_3 * P_4 = P\ \wedge\ H, P_3, \sigma \vDash_{TPL} A_1\ \wedge\ H, P_4, \sigma \vDash_{TPL} A_2 \\
\quad \Rightarrow \exists P_1, P_2.\ P_1 * P_2 = P\ \wedge\ (H, \varnothing, \sigma) \triangleleft P_1 \vDash_{TPL} A_1\ \wedge\ (H, \varnothing, \sigma) \triangleleft P_2 \vDash_{TPL} A_2
\end{array}
$$

by complete (strong) induction on $|rds(P_3) \cap rds(P_4)|$. In the proof, we use the shorthand $P[(\iota, f) \mapsto \pi]$ to denote the permission mask that returns π for (ι, f) and behaves like P for all other entries, and also the shorthand $P \setminus (\iota, f)$ for $P[(\iota, f) \mapsto 0]]$.

We now consider two cases:

$(\exists(\iota,f) \in rds(P_3).\ H,(P_3 \setminus (\iota,f)),\sigma \vDash_{TPL} A_1)$
By Proposition 3.13, we know $H,(P_4[(\iota,f) \mapsto P[\iota,f]]),\sigma \vDash_{TPL} A_2$. We know $P_4[\iota,f] > 0$, otherwise $P_3 * P_4$ was not minimal to start with. Therefore $|rds(P_3 \setminus (\iota,f)) \cap rds(P_4[(\iota,f) \mapsto P[\iota,f]])| < |rds(P_3) \cap rds(P_4)|$. By construction, we know $(P_3 \setminus (\iota,f)) * (P_4[(\iota,f) \mapsto P[\iota,f]]) = P_3 * P_4$, so this case holds by induction choosing $(P_3 \setminus (\iota,f))$ for P_3 and $P_4[(\iota,f) \mapsto P[\iota,f]]$ for P_4 in inductive hypothesis.

$(\forall(\iota,f) \in rds(P_3).\ H,(P_3 \setminus (\iota,f)),\sigma \nvDash_{TPL} A_1)$
Therefore, $(H,\varnothing,\sigma) \lhd P_3 \vDash_{TPL} A_1$. Consider two sub-cases:

$(\exists(\iota,f) \in rds(P_4).\ H,(P_4 \setminus (\iota,f)),\sigma \vDash_{TPL} A_2)$
By Proposition 3.13, we know $H,(P_3[(\iota,f) \mapsto P[\iota,f]]),\sigma \vDash_{TPL} A_1$. By construction, we know $P_3[\iota,f] > 0$, otherwise $P_3 * P_4$ was not minimal to start with. Therefore $|rds(P_4 \setminus (\iota,f)) \cap rds(P_3[(\iota,f) \mapsto P[\iota,f]])| < |rds(P_3) \cap rds(P_4)|$. We know $(P_4 \setminus (\iota,f)) * (P_3[(\iota,f) \mapsto P[\iota,f]]) = P_3 * P_4$, so this case holds by induction choosing $(P_4 \setminus (\iota,f))$ for P_4 and $P_3[(\iota,f) \mapsto P[\iota,f]]$ for P_3 in inductive hypothesis.

$(\forall(\iota,f) \in rds(P_4).\ H,(P_4 \setminus (\iota,f)),\sigma \nvDash_{TPL} A_2)$
Then, $(H,\varnothing,\sigma) \lhd P_4 \vDash_{TPL} A_2$. Hence, we have solution choosing $P_3 = P_1$ and $P_4 = P_2$.

(2) We can assume $(H,P,\sigma) \lhd P' \vDash_{TPL} A_1 \wedge A_2$ and hence $H,P * P',\sigma \vDash A_1$ and $H,P * P',\sigma \vDash A_2$. By Lemma 3.9, we know there exists P'_1 such that $P'_1 \subseteq P'$ and $(H,P,\sigma) \lhd P'_1 \vDash_{TPL} A_1$. Define

$$P_1 = \lambda(\iota,f).\ \text{if } P'_1[\iota,f] = 0 \text{ then } 0 \text{ else } P'[\iota,f].$$

Note that $P_1 \subseteq P'$. By Lemma 3.11, we know $(H,P,\sigma) \lhd P_1 \vDash_{TPL} A_1$.

We know $H,(P * P_1) * (P' - P_1),\sigma \vDash_{TPL} A_2$, and by Lemma 3.9 we know there exists $P'_2 \subseteq (P' - P_1)$ such that $(H,P * P_1,\sigma) \lhd P'_2 \vDash_{TPL} A_2$. Define

$$P_2 = \lambda(\iota,f).\ \text{if } P'_2[\iota,f] = 0 \text{ then } 0 \text{ else } P'[\iota,f].$$

Note that $P_2 \subseteq P' - P_1$, and thus $P_1 * P_2 \subseteq P'$. By Lemma 3.11, we deduce that $(H,P,\sigma) \lhd P_2 \vDash_{TPL} A_2$.

By construction of P_1 and P_2, either $P_1 * P_2 = P'$ or $rds(P_1 * P_2) \subset rds(P')$. In the first case we are done. In the second case, we seek a contradiction. We know $H,P * P_1 * P_2,\sigma \vDash_{TPL} A_2$, and by Prop 3.13, we know $H,P * P_1 * P_2,\sigma \vDash_{TPL} A_1$, hence $H,P * P_1 * P_2,\sigma \vDash_{TPL} A_1 \wedge A_2$. As we know $P_1 * P_2 \subseteq P'$, but that contradicts the initial assumption of P' being minimal.

Lemma 3.25 (Composing Minimal Permission Extensions over Supported Conjunctions)**.**

(1) *If* $(H,\varnothing,\sigma) \lhd P_1 \vDash_{TPL} A_1$ *and* $(H,\varnothing,\sigma) \lhd P_2 \vDash_{TPL} A_2$ *and* A_1 *and* A_2 *are supported, then* $(H,\varnothing,\sigma) \lhd P_1 * P_2 \vDash_{TPL} A_1 * A_2$.

(2) *If* $(H,P,\sigma) \lhd P_1 \vDash_{TPL} A_1$ *and* $(H,P * P_1,\sigma) \lhd P_2 \vDash_{TPL} A_2$ *and* A_1 *and* A_2 *are supported, then* $(H,P,\sigma) \lhd P_1 * P_2 \vDash_{TPL} A_1 \wedge A_2$.

Proof.

(1) Proof by contradiction. We know $H,P_1 * P_2,\sigma \vDash_{TPL} A_1 * A_2$ holds, therefore we assume that there exists $P' \subseteq P_1 * P_2$ such that $rds(P') \subset rds(P_1 * P_2)$ and $H,P',\sigma \vDash_{TPL} A_1 * A_2$. Therefore, there exist P_3 and P_4 such that $P_3 * P_4 = P'$ and $H,P_3,\sigma \vDash_{TPL} A_1$ and $H,P_4,\sigma \vDash_{TPL} A_2$. From $rds(P') \subset rds(P_1 * P_2)$, we know $(\iota,f) \in rds(P_1 * P_2)$ and $(\iota,f) \notin rds(P')$. W.l.o.g assume $(\iota,f) \in rds(P_1)$. As A_1 is supported, we know $H,P_1 \sqcap P_3,\sigma \vDash_{TPL} A_1$. Since $(P_1 \sqcap P_3) \subseteq P_3 \subseteq P'$ we have $(\iota,f) \notin rds(P_1 \sqcap P_3)$. But this contradicts $(H,P,\sigma) \lhd P_1 \vDash_{TPL} A_1$, since $(\iota,f) \in rds(P_1)$.

(2) By assumptions and Prop 3.13, we have $H, P * P_1 * P_2, \sigma \vDash_{TPL} A_1 \wedge A_2$. We show $(H, P, \sigma) \lhd P_1 * P_2 \vDash_{TPL} A_1 \wedge A_2$ by contradiction. Assume there exists $P_3 \subseteq (P_1 * P_2)$ such that $rds(P_3) \subset rds(P_1 * P_2)$ and $H, P * P_3, \sigma \vDash_{TPL} A_1 \wedge A_2$. Then there exists $(\iota, f) \in rds(P_1 * P_2)$ such that $(\iota, f) \notin rds(P_3)$. Case split:
   ($(\iota, f) \in rds(P_1)$) Note that $(P * P_3) \sqcap (P * P_1) = P * (P_1 \sqcap P_3)$, thus as A_1 is supported we have $H, P * (P_1 \sqcap P_3), \sigma \vDash_{TPL} A_1$. but this contradicts $(H, P, \sigma) \lhd P_1 \vDash_{TPL} A_1$
   ($(\iota, f) \in rds(P_2)$) Note that $(P * P_3) \sqcap (P * P_1 * P_2) = P * (P_3 \sqcap (P_1 * P_2))$. As A_2 is supported, we know $H, P * (P_3 \sqcap (P_1 * P_2)), \sigma \vDash_{TPL} A_2$, but this contradicts $(H, P * P_1, \sigma) \lhd P_2 \vDash_{TPL} A_2$.

Proposition 3.26. *For all TPL assertions A_1, A_2, A_3:*

(1) $A_1 * (A_1 \mathrel{-\!\!*} A_2) \vDash_{TPL} A_2$
(2) $A_1 \wedge (A_1 \to A_2) \vDash_{TPL} A_2$
(3) (a) $\mathsf{DisExtFrm}(A_1) \cap \langle\!\langle A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3) \rangle\!\rangle \subseteq \langle\!\langle (A_1 * A_2) \mathrel{-\!\!*} A_3 \rangle\!\rangle$
   (b) *if A_1 and A_2 are supported, then:*
   $\mathsf{DisExtFrm}(A_1) \cap \langle\!\langle (A_1 * A_2) \mathrel{-\!\!*} A_3 \rangle\!\rangle \subseteq \langle\!\langle (A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3) \rangle\!\rangle$
   (c) *if both $A_1 * A_2$ and A_3 are self-framing, then:*
   $\mathsf{DisExtFrm}(A_1) \cap \langle\!\langle (A_1 * A_2) \mathrel{-\!\!*} A_3 \rangle\!\rangle \subseteq \langle\!\langle (A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3) \rangle\!\rangle$
(4) (a) $\mathsf{ExtFrm}(A_1) \cap \langle\!\langle A_1 \to (A_2 \to A_3) \rangle\!\rangle \subseteq \langle\!\langle (A_1 \wedge A_2) \to A_3 \rangle\!\rangle$
   (b) *if A_1 and A_2 are supported, then:*
   $\mathsf{ExtFrm}(A_1) \cap \langle\!\langle (A_1 \wedge A_2) \to A_3 \rangle\!\rangle \subseteq \langle\!\langle A_1 \to (A_2 \to A_3) \rangle\!\rangle$
   (c) *if both $A_1 \wedge A_2$ and A_3 are self-framing, then:*
   $\mathsf{ExtFrm}(A_1) \cap \langle\!\langle (A_1 \wedge A_2) \to A_3 \rangle\!\rangle \subseteq \langle\!\langle A_1 \to (A_2 \to A_3) \rangle\!\rangle$
(5) *If $A_1 \vDash_{TPL} (A_2 \mathrel{-\!\!*} A_3)$ then $(A_1 * A_2) \vDash_{TPL} A_3$*
(6) *If A_1 is self-framing and $(A_1 * A_2) \vDash_{TPL} A_3$ then $A_1 \vDash_{TPL} (A_2 \mathrel{-\!\!*} A_3)$*

Proof.

(1) Assume $H, P, \sigma \vDash_{TPL} A_1 * (A_1 \mathrel{-\!\!*} A_2)$. We seek to prove that $H, P, \sigma \vDash_{TPL} A_2$. From our assumption, there exist P_1, P_2 such that $P_1 * P_2 = P$ and $H, P_1, \sigma \vDash_{TPL} A_1$ and $H, P_2, \sigma \vDash_{TPL} A_1 \mathrel{-\!\!*} A_2$. From the latter, we have that

$$\forall P_3 \perp P_2, \forall H_3 \overset{rds(P_2) \cup \overline{rds(P_3)}}{\equiv} H.(H_3, \varnothing, \sigma) \lhd P_3 \vDash_{TPL} A_1 \Rightarrow H_3, P_2 * P_3, \sigma \vDash_{TPL} A_2.$$

   From $H, P_1, \sigma \vDash_{TPL} A_1$, by Lemma 3.9, we know that there exists $P_3 \subseteq P_1$ such that $(H, \varnothing, \sigma) \lhd P_3 \vDash_{TPL} A_1$. Combining these facts, we obtain that $H, P_2 * P_3, \sigma \vDash_{TPL} A_2$ holds. By Proposition 3.13, we obtain $H, P_2 * P_1, \sigma \vDash_{TPL} A_2$ as required.

(2) Assume $H, P, \sigma \vDash_{TPL} A_1 \wedge (A_1 \to A_2)$. We seek to prove that $H, P, \sigma \vDash_{TPL} A_2$. From our assumption, we obtain both $H, P, \sigma \vDash_{TPL} A_1$ and $H, P, \sigma \vDash_{TPL} A_1 \to A_2$. From the latter, we have

$$\forall P_1 \perp P, \forall H_1 \overset{rds(P) \cup \overline{rds(P_1)}}{\equiv} H.(H_1, P, \sigma) \lhd P_1 \vDash_{TPL} A_1 \Rightarrow H_1, P * P_1, \sigma \vDash_{TPL} A_2$$

   Taking $H_1 = H$ and $P_1 = \varnothing$ in the above (and noting that from $H, P, \sigma \vDash_{TPL} A_1$ we can easily obtain $(H, P, \sigma) \lhd \varnothing \vDash_{TPL} A_1$), we can obtain $H, P, \sigma \vDash_{TPL} A_2$ as required.

(3) In the following, we assume (as in the statement of the Lemma) that $(H, P, \sigma) \in \mathsf{DisExtFrm}(A_1)$

(a) We assume $H, P, \sigma \vDash_{TPL} A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3)$ and seek to prove $H, P, \sigma \vDash_{TPL} (A_1 * A_2) \mathrel{-\!\!*} A_3$. Thus, we can assume

$$(H_3, P_3, \sigma) \in \mathsf{localDisjExts}(H, P, \sigma)$$
$$(H_3, \varnothing, \sigma) \lhd P_3 \vDash_{TPL} A_1 * A_2$$

and must prove

$$H_3, P * P_3, \sigma \vDash_{TPL} A_3$$

By Lemma 3.23 (1), there exist $P_1 \perp P_2$ such that $P_3 = P_1 * P_2$ and both

$$(H_3, \varnothing, \sigma) \lhd P_1 \vDash_{TPL} A_1$$
$$(H_3, \varnothing, \sigma) \lhd P_2 \vDash_{TPL} A_2$$

By the definition of $\mathsf{localDisjExts}(H, P, \sigma)$, we can show

$$(H', P_1, \sigma) \in \mathsf{localDisjExts}(H, P, \sigma)$$
$$(H_3, P_2, \sigma) \in \mathsf{localDisjExts}(H', P * P_1, \sigma)$$

where $H' = (P_1 ? H_3 : H)$. By assumptions, we know $H_3 \overset{P}{\equiv} H$, thus $H' \overset{P}{\equiv} H_3$. By construction, $H' \overset{P_1}{\equiv} H_3$, and thus $H' \overset{P * P_1}{\equiv} H_3$.

By $\mathsf{DisExtFrm}(H, P, \sigma)$ assumption, and Lemma 3.16 (2), we get

$$(H', \varnothing, \sigma) \lhd P_1 \vDash_{TPL} A_1$$

Now using $H, P, \sigma \vDash_{TPL} A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3)$, we get

$$H', P * P_1, \sigma \vDash_{TPL} A_2 \mathrel{-\!\!*} A_3$$

and thus

$$H_3, P * P_1 * P_2, \sigma \vDash_{TPL} A_3$$

as required.

(b) and (c) We prove these two cases together, since they are almost identical. In the proof, we case split on which extra assumption to use: either A_1 and A_2 are supported (for part (b)) or both $A_1 * A_2$ and A_3 are self-framing (for part (c)).

We assume $H, P, \sigma \vDash_{TPL} (A_1 * A_2) \mathrel{-\!\!*} A_3$ and seek to prove $H, P, \sigma \vDash_{TPL} A_1 \mathrel{-\!\!*} (A_2 \mathrel{-\!\!*} A_3)$. Thus, we can assume

$$(H_1, P_1, \sigma) \in \mathsf{localDisjExts}(H, P, \sigma)$$
$$(H_1, \varnothing, \sigma) \lhd P_1 \vDash_{TPL} A_1$$
$$(H_2, P_2, \sigma) \in \mathsf{localDisjExts}(H_1, P * P_1, \sigma)$$
$$(H_2, \varnothing, \sigma) \lhd P_2 \vDash_{TPL} A_2$$

and must prove

$$H_2, P * P_1 * P_2, \sigma \vDash_{TPL} A_3$$

By definition of $\mathsf{localDisjExts}(H, P, \sigma)$ we can show

$$(H_2, P_1 * P_2, \sigma) \in \mathsf{localDisjExts}(H, P, \sigma)$$

By Lemma 3.15, we know

$$(H_1, P, \sigma) \in \mathsf{DisExtFrm}(A_1)$$

and thus, by Lemma 3.16 (2) we know

$$(H_2, \varnothing, \sigma) \lhd P_1 \vDash_{TPL} A_1$$

Now, we case-split on whether we are proving part (b) or (c):

part (b) Then we can assume that A_1 and A_2 are supported. By Lemma 3.25 (1), we can obtain $(H_2, \varnothing, \sigma) \lhd P_1 * P_2 \vDash_{TPL} A_1 * A_2$. Thus, using $H, P, \sigma \vDash_{TPL} (A_1 * A_2) \mathrel{-\!\!*} A_3$ we get

$$H_2, P * P_1 * P_2, \sigma \vDash_{TPL} A_3$$

as required.

part (c) Then, we can assume that both $A_1 * A_2$ and A_3 are self-framing. By Lemma 3.9, there exists $P_3 \subseteq (P_1 * P_2)$ such that $(H_2, \varnothing, \sigma) \lhd P_3 \vDash_{TPL} A_1 * A_2$. Define $H_3 = (P_3 \,?\, H_2 : H)$. Then we have $H_3 \overset{P * P_3}{\equiv} H_2$. Since $A_1 * A_2$ is self-framing, by Lemma 3.16 (3), we have $(H_3, \varnothing, \sigma) \lhd P_3 \vDash_{TPL} A_1 * A_2$. We need to show that

$$(H_3, P_3, \sigma) \in \mathsf{localDisjExts}(H, P, \sigma)$$

which follows as $H_3 \overset{P}{\equiv} H$ (since $H \overset{P}{\equiv} H_1 \overset{P}{\equiv} H_2 \overset{P}{\equiv} H_3$). Thus, by assumption, we get $H_3, P * P_3, \sigma \vDash_{TPL} A_3$. Since A_3 is self-framing, and since $H_2 \overset{P * P_3}{\equiv} H_3$, we obtain $H_2, P * P_3, \sigma \vDash_{TPL} A_3$. By Proposition 3.13, we have $H_2, P * P_1 * P_2, \sigma \vDash_{TPL} A_3$ as required.

(4) In the following, we assume (as in the statement of the Lemma) that $(H, P, \sigma) \in \mathsf{ExtFrm}(A_1)$

(a) We assume $H, P, \sigma \vDash_{TPL} A_1 \to (A_2 \to A_3)$ and seek to prove that $H, P, \sigma \vDash_{TPL} (A_1 \wedge A_2) \to A_3$. Thus, we can assume

$$(H_3, P * P_3, \sigma) \in \mathsf{localExts}(H, P, \sigma)$$
$$(H_3, P, \sigma) \lhd P_3 \vDash_{TPL} A_1 \wedge A_2$$

and must prove

$$H_3, P * P_3, \sigma \vDash_{TPL} A_3$$

By Lemma 3.23 (2), there exist $P_1 \perp P_2$ such that $P_3 = P_1 * P_2$ and both

$$(H_3, P, \sigma) \lhd P_1 \vDash_{TPL} A_1$$
$$(H_3, P * P_1, \sigma) \lhd P_2 \vDash_{TPL} A_2$$

By the definition of $\mathsf{localExts}(H, P, \sigma)$, we can show

$$(H', P * P_1, \sigma) \in \mathsf{localExts}(H, P, \sigma)$$
$$(H_3, P * P_1 * P_2, \sigma) \in \mathsf{localExts}(H', P * P_1, \sigma)$$

where $H' = (P_1 ? H_3 : H)$. By assumptions, we know $H_3 \overset{P}{\equiv} H$, thus $H' \overset{P}{\equiv} H_3$. By construction, $H' \overset{P_1}{\equiv} H_3$, and thus $H_3 \overset{P * P_1}{\equiv} H'$.

By $\mathsf{ExtFrm}(H, P, \sigma)$ assumption, and Lemma 3.16 (1), we get

$$(H', P, \sigma) \lhd P_1 \vDash_{TPL} A_1$$

Now using $H, P, \sigma \vDash_{TPL} A_1 \to (A_2 \to A_3)$, we get

$$H', P * P_1, \sigma \vDash_{TPL} A_2 \to A_3$$

and thus

$$H_3, P * P_1 * P_2, \sigma \vDash_{TPL} A_3$$

as required.

(b) and (c) We prove these two cases together, since they are almost identical. In the proof, we case split on which extra assumption to use: either A_1 and A_2 are supported (for part (b)) or both $A_1 \wedge A_2$ and A_3 are self-framing (for part (c)). We assume $H, P, \sigma \vDash_{TPL} (A_1 \wedge A_2) \to A_3$ and seek to prove $H, P, \sigma \vDash_{TPL} A_1 \to (A_2 \to A_3)$. Thus, we can assume

$$\begin{array}{l}(H_1, P_1 * P, \sigma) \in \mathsf{localExts}(H, P, \sigma)\\ (H_1, P, \sigma) \lhd P_1 \vDash_{TPL} A_1\\ (H_2, P * P_1 * P_2, \sigma) \in \mathsf{localExts}(H_1, P * P_1 * P_2, \sigma)\\ (H_2, P * P_1, \sigma) \lhd P_2 \vDash_{TPL} A_2\end{array}$$

and must prove

$$H_2, P * P_1 * P_2, \sigma \vDash_{TPL} A_3$$

By definition of $\mathsf{localExts}(H, P, \sigma)$ we can show

$$(H_2, P * P_1 * P_2, \sigma) \in \mathsf{localExts}(H, P, \sigma)$$

By Lemma 3.15, we know

$$(H_1, P * P_1, \sigma) \in \mathsf{ExtFrm}(A_1)$$

and thus, by Lemma 3.16 (2) we know

$$(H_2, P, \sigma) \lhd P_1 \vDash_{TPL} A_1$$

Now, we case-split on whether we are proving part (b) or (c):

part (b) Then we can assume that A_1 and A_2 are supported. By Lemma 3.25 (2), we can obtain $(H_2, P, \sigma) \lhd P_1 * P_2 \vDash_{TPL} A_1 \wedge A_2$. Thus, using $H, P, \sigma \vDash_{TPL} (A_1 \wedge A_2) \to A_3$ we get

$$H_2, P * P_1 * P_2, \sigma \vDash_{TPL} A_3$$

as required.

part (c) Then, we can assume that both $A_1 \wedge A_2$ and A_3 are self-framing. By Lemma 3.9, there exists $P_3 \subseteq (P_1 * P_2)$ such that $(H_2, P, \sigma) \lhd P_3 \vDash_{TPL} A_1 \wedge A_2$. Define $H_3 = (P_3 \,?\, H_2 : H)$. Then we have $H_3 \overset{P * P_3}{\equiv} H_2$. Since $A_1 \wedge A_2$ is self-framing, by Lemma 3.16 (3), we have $(H_3, P, \sigma) \lhd P_3 \vDash_{TPL} A_1 * A_2$. We need to show that

$$(H_3, P * P_3, \sigma) \in \mathsf{localExts}(H, P, \sigma)$$

which follows as $H_3 \overset{P}{\equiv} H$. By assumption, we get $H_3, P * P_3, \sigma \vDash_{TPL} A_3$. Since A_3 is self-framing, and since $H_2 \overset{P * P_3}{\equiv} H_3$, we obtain $H_2, P * P_3, \sigma \vDash_{TPL} A_3$. By Proposition 3.13, we have $H_2, P * P_1 * P_2, \sigma \vDash_{TPL} A_3$ as required.

(5) We can assume that:

$$\begin{array}{l}\forall H, P, \sigma. (H, P, \sigma \vDash_{TPL} A_1 \Rightarrow\\ \quad \forall P_1 \perp P, \forall H_1 \overset{rds(P) \cup \overline{rds(P_1)}}{\equiv} H.\\ \qquad ((H_1, \varnothing, \sigma) \lhd P_1 \vDash_{TPL} A_2 \Rightarrow H_1, P * P_1, \sigma \vDash_{TPL} A_3))\end{array}$$

We need to know that, assuming that (for some H_2, P_2) $H_2, P_2, \sigma \vDash_{TPL} A_1 * A_2$ holds, we can deduce that $H_2, P_2, \sigma \vDash_{TPL} A_3$ also holds. The former means that there exist P_3 and P_4 such that $P_2 = P_3 * P_4$ and both $H_2, P_3, \sigma \vDash_{TPL} A_1$ and $H_2, P_4, \sigma \vDash_{TPL} A_2$ hold. By Lemma 3.9, there exists $P_5 \subseteq P_4$ such that $(H_2, \varnothing, \sigma) \lhd P_5 \vDash_{TPL} A_2$ holds. Now we apply our original assumption, defining $H = H_2$ and $H_1 = H_2$ and $P = (P_3 * (P_4 - P_5))$ and $P_1 = P_5$ (note that, by Proposition 3.13, we have $H, P, \sigma \vDash_{TPL} A_1$). From the assumption,

we obtain $H, P * P_1, \sigma \vDash_{TPL} A_3$, i.e., $H_2, P_3 * P_4, \sigma \vDash_{TPL} A_3$, i.e., $H_2, P_2, \sigma \vDash_{TPL} A_3$ as required.

(6) We can assume that

$$\forall H, P, \sigma.(H, P, \sigma \vDash_{TPL} A_1 * A_2 \Rightarrow H, P, \sigma \vDash_{TPL} A_3)$$

i.e., we (equivalently) assume that:

$$\forall H, P_1, P_2, \sigma.(P_1 \perp P_2 \wedge H, P_1, \sigma \vDash_{TPL} A_1 \wedge H, P_2, \sigma \vDash_{TPL} A_2 \Rightarrow H, P_1 * P_2, \sigma \vDash_{TPL} A_3)$$

We need to show that, if we assume (for some H_1 and P_1) that $H_1, P_1, \sigma \vDash_{TPL} A_1$, then we can deduce that $H_1, P_1, \sigma \vDash_{TPL} A_2 \mathrel{-\!\!*} A_3$ holds, i.e., that:

$$\forall P_2 \perp P_1, \forall H_2 \overset{rds(P_1) \cup \overline{rds(P_2)}}{\equiv} H_1. ((H_2, \varnothing, \sigma) \lhd P_2 \vDash_{TPL} A_2 \Rightarrow H_2, P_1 * P_2, \sigma \vDash_{TPL} A_3)$$

To show this, we assume $P_2 \perp P_1$ and $H_2 \overset{rds(P_1) \cup \overline{rds(P_2)}}{\equiv} H_1$ and $(H_2, \varnothing, \sigma) \lhd P_2 \vDash_{TPL} A_2$ and need to prove $H_2, P_1 * P_2, \sigma \vDash_{TPL} A_3$. Since A_1 is self-framing, and since $H_2 \overset{P_1}{\equiv} H_1$, we know that $H_2, P_1, \sigma \vDash_{TPL} A_1$. Then, letting $H = H_2$, we can apply our original assumption to obtain $H_2, P_1 * P_2, \sigma \vDash_{TPL} A_3$ as required.

Lemma 4.5.

(1) *If* $H, P, \sigma \vDash_{TPL} \mathsf{sframed}(E)$, *and* $H' \overset{P}{\equiv} H$ *then* $[\![E]\!]_{H,\sigma} = [\![E]\!]_{H',\sigma}$.

(2) $\mathsf{sframed}(E)$ *is self-framing*

(3) *If* $H, P, \sigma \vDash_{TPL} \mathsf{sframed}(B)$, *and* $H' \overset{P}{\equiv} H$ *then* $H, P, \sigma \vDash_{TPL} B$ *if and only if* $H', P, \sigma \vDash_{TPL} B$.

(4) $\mathsf{sframed}(B)$ *is self-framing.*

Proof.

(1) Follows by straightforward induction on E.

(2) Follows by induction on E, and using previous property. The base cases hold trivially. For the inductive case $(E.f)$, we assume

$$H, P, \sigma \vDash_{TPL} \mathsf{sframed}(E) \qquad P[[\![E]\!]_{H,\sigma}, f] \geq \pi \qquad H \overset{P}{\equiv} H'$$

and need to show that

$$H', P, \sigma \vDash_{TPL} \mathsf{sframed}(E) \qquad P[[\![E]\!]_{H',\sigma}, f] \geq \pi$$

The first part follows from the inductive hypothesis. The second part follows as we know $[\![E]\!]_{H,\sigma} = [\![E]\!]_{H',\sigma}$ by the previous part of the lemma.

(3) By induction on B. The base cases hold trivially. For the inductive case, assume

$$H, P, \sigma \vDash_{TPL} \mathsf{sframed}(B_1)$$
$$H, P, \sigma \vDash_{TPL} B_1 \to \mathsf{sframed}(B_2)$$
$$H \overset{P}{\equiv} H'$$

By inductive hypothesis, we know

$$H, P, \sigma \vDash_{TPL} B_1 \iff H', P, \sigma \vDash_{TPL} B_1$$

We case split on whether or not B_1 holds. For the first case, assume $H, P, \sigma \vDash_{TPL} B_1$. Therefore, by Lemma 3.18 we know

$$H, P, \sigma \vDash_{TPL} \mathsf{sframed}(B_2)$$

and thus, by inductive hypothesis

$$H, P, \sigma \vDash_{TPL} B_2 \iff H', P, \sigma \vDash_{TPL} B_2$$

Hence, we know

$$H, P, \sigma \vDash_{TPL} B_1 * B_2 \iff H', P, \sigma \vDash_{TPL} B_1 * B_2$$

as required.

For the second case, assume $H, P, \sigma \nvDash_{TPL} B_1$. Therefore

$$H', P, \sigma \nvDash_{TPL} B_1$$

and thus we know

$$H, P, \sigma \vDash_{TPL} B_1 * B_2 \iff H', P, \sigma \vDash_{TPL} B_1 * B_2$$

as required.

(4) By induction on B. The base cases follow directly from previous parts of this lemma. For the inductive case, we assume

$$\begin{array}{l} H, P, \sigma \vDash_{TPL} \mathsf{sframed}(B_1) \\ H, P, \sigma \vDash_{TPL} B_1 \to \mathsf{sframed}(B_2) \\ H \overset{P}{\equiv} H' \end{array}$$

and we seek to prove

$$\begin{array}{l} H', P, \sigma \vDash_{TPL} \mathsf{sframed}(B_1) \\ H', P, \sigma \vDash_{TPL} B_1 \to \mathsf{sframed}(B_2) \end{array}$$

The first obligation follows by inductive hypothesis. Using Lemma 3.18, we can assume $H', P, \sigma \vDash_{TPL} B_1$, and must prove $H', P, \sigma \vDash_{TPL} \mathsf{sframed}(B_2)$. Thus, by previous part, we know $H, P, \sigma \vDash_{TPL} B_1$, and by Lemma 3.18 we know

$$H, P, \sigma \vDash_{TPL} \mathsf{sframed}(B_2)$$

By inductive hypothesis, we obtain

$$H', P, \sigma \vDash_{TPL} \mathsf{sframed}(B_2)$$

as required.

Lemma 4.6.

(1) $\mathsf{sframed}(p_1) \wedge ((p_1 * p_2) \mathrel{-\!\!*} p) \vDash_{TPL} p_1 \mathrel{-\!\!*} (p_2 \mathrel{-\!\!*} p)$
(2) $\mathsf{sframed}(p_1) \wedge (p_1 \mathrel{-\!\!*} (p_2 \mathrel{-\!\!*} p)) \vDash_{TPL} (p_1 * p_2) \mathrel{-\!\!*} p$

Proof. We break this proof into two steps. First we prove that the $\mathsf{sframed}(p_1)$ condition has a semantic meaning in terms of $\mathsf{DisExtFrm}(p_1)$, and then show this semantic meaning allows the restructuring of the assertion. That is, we show that

$$\forall p.\ \langle\!\langle \mathsf{sframed}(p) \rangle\!\rangle \subseteq \mathsf{DisExtFrm}(p) \tag{A.1}$$

and then show

$$\mathsf{DisExtFrm}(p_1) \cap \langle\!\langle (p_1 * p_2) \mathrel{-\!\!*} p \rangle\!\rangle \subseteq \langle\!\langle p_1 \mathrel{-\!\!*} (p_2 \mathrel{-\!\!*} p) \rangle\!\rangle \tag{A.2}$$

and

$$\mathsf{DisExtFrm}(p_1) \cap \langle\!\langle p_1 \mathrel{-\!\!*} (p_2 \mathrel{-\!\!*} p) \rangle\!\rangle \subseteq \langle\!\langle (p_1 * p_2) \mathrel{-\!\!*} p \rangle\!\rangle \tag{A.3}$$

To prove (A.2) we use Proposition 3.26(3)(b) and Lemma 4.3, and (A.3) is just a restatement of Proposition 3.26(3)(a).

To prove (A.1) we use induction on p.

($p \equiv acc(E, f, \pi)$) This requires that we prove

$$\langle\!\langle \mathsf{sframed}(E) \rangle\!\rangle \subseteq \mathsf{DisExtFrm}(acc(E.f, \pi))$$

By expanding the definition of $\mathsf{DisExtFrm}(acc(E.f, \pi))$ and the semantics of $acc(E.f, \pi)$ we can assume

$$H, P, \sigma \vDash_{TPL} \mathsf{sframed}(E) \qquad H \overset{P}{\equiv} H' \qquad P \perp P'$$
$$P'[[\![E]\!]_{H',\sigma}, f] \geq \pi \qquad H' \overset{P*P'}{\equiv} H''$$

and are required to prove $P'[[\![E]\!]_{H'',\sigma}, f] \geq \pi$. By definition of $\equiv$, we can get $H' \overset{P}{\equiv} H''$, and thus use Lemma 4.5, to give $[\![E]\!]_{H',\sigma} = [\![E]\!]_{H'',\sigma}$ as required.

($p \equiv B$) This case requires that we prove

$$\langle\!\langle \mathsf{sframed}(B) \rangle\!\rangle \subseteq \mathsf{DisExtFrm}(B)$$

By expanding the definition of $\mathsf{DisExtFrm}(B)$ we can assume

$$H, P, \sigma \vDash_{TPL} \mathsf{sframed}(B) \qquad H \overset{P}{\equiv} H' \qquad P \perp P'$$
$$H', P', \sigma \vDash_{TPL} B \qquad H' \overset{P*P'}{\equiv} H''$$

and are required to prove $H'', P', \sigma \vDash_{TPL} B$. By definition of $\equiv$, we know $H' \overset{P}{\equiv} H''$, and thus use Lemma 4.5 to give $H', P', \sigma \vDash_{TPL} B \iff H'', P', \sigma \vDash_{TPL} B$ as required.

($p \equiv p_1 * p_2$) We assume

$$\langle\!\langle \mathsf{sframed}(p_1) \rangle\!\rangle \subseteq \mathsf{DisExtFrm}(p_1)$$
$$\langle\!\langle \mathsf{sframed}(p_2) \rangle\!\rangle \subseteq \mathsf{DisExtFrm}(p_2)$$

and, expanding the definition of $\mathsf{sframed}(p_1 * p_2)$, we must show

$$\langle\!\langle \mathsf{sframed}(p_1) \wedge (p_1 \mathrel{-\!\!*} \mathsf{sframed}(p_2)) \rangle\!\rangle \subseteq \mathsf{DisExtFrm}(p_1 * p_2)$$

We can assume, by expanding the definition of $\mathsf{DisExtFrm}(p_1 * p_2)$, and the definition of the semantics of $*$:

$$H, P, \sigma \vDash_{TPL} \mathsf{sframed}(p_1)$$
$$H, P, \sigma \vDash_{TPL} p_1 \mathrel{-\!\!*} \mathsf{sframed}(p_2)$$
$$H \overset{P}{\equiv} H'$$
$$P_1 \perp P \wedge P_2 \perp P \wedge P_1 \perp P_2$$
$$H', P_1, \sigma \vDash_{TPL} p_1$$
$$H', P_2, \sigma \vDash_{TPL} p_2$$
$$H' \overset{P*P_1*P_2}{\equiv} H''$$

and we are left with proving:

$$H'', P_1 * P_2, \sigma \vDash_{TPL} p_1 * p_2$$

Let $H_1 = (rds(P) \cup \overline{rds(P_1)} \ ?\ H : H')$. By inductive hypothesis, we know $(H, P, \sigma) \in \mathsf{DisExtFrm}(p_1)$, and thus we know $H_1, P_1, \sigma \vDash_{TPL} p_1$.

By Lemma 3.9, we can prove that there exist P_1', P_1'' such that $P_1' * P_1'' = P_1$ and $(H_1, \varnothing, \sigma) \lhd P_1' \vDash_{TPL} p_1$. Therefore, we know $H_1, P_1' * P, \sigma \vDash_{TPL} \mathsf{sframed}(p_2)$, and thus $(H_1, P_1' * P, \sigma) \in \mathsf{DisExtFrm}(p_2)$. As $H \overset{P}{\equiv} H'$, we know $H_1 \overset{P}{\equiv} H'$ by construction. Moreover,

by construction we know $H_1 \overset{rds(P_1) \setminus rds(P)}{\equiv} H'$, which with the previous gives $H_1 \overset{P_1}{\equiv} H'$. Thus, we know $H_1 \overset{P_1 * P}{\equiv} H'$, which we can weaken to $H_1 \overset{P_1' * P}{\equiv} H'$. Thus, we know $(H', P_1' * P, \sigma) \in \mathsf{DisExtFrm}(p_2)$

As all assertions are weakening-closed (cf. Proposition 3.13), we have $H', P_1'' * P_2, \sigma \vDash_{TPL} p_2$. We know $P_1'' * P_2 \perp P_1' * P$, thus using $\mathsf{DisExtFrm}(p_2)$, we know $H'', P_1'' * P_2, \sigma \vDash_{TPL} p_2$.

By $(H, P, \sigma) \in \mathsf{DisExtFrm}(p_1)$ and $H \overset{P}{\equiv} H'$, we know $(H', P, \sigma) \in \mathsf{DisExtFrm}(p_1)$. By weakening assumption we know $H' \overset{P * P_1'}{\equiv} H''$. As $P_1' \perp P$, we know $H'', P_1', \sigma \vDash_{TPL} p_1$ and thus

$$H'', P_1 * P_2, \sigma \vDash_{TPL} p_1 * p_2$$

as required.

($p \equiv B \to p'$) This case requires

$$\{\!|\mathsf{sframed}(B) \wedge (B \to \mathsf{sframed}(p'))|\!\} \subseteq \mathsf{DisExtFrm}(B \to p')$$

By expanding the definition of $\mathsf{DisExtFrm}(B \to p')$ and using Lemma 3.18, we can assume

$$\begin{array}{l} H, P, \sigma \vDash_{TPL} \mathsf{sframed}(B) \\ H, P, \sigma \vDash_{TPL} B \Rightarrow H, P, \sigma \vDash_{TPL} \mathsf{sframed}(p') \\ H \overset{P}{\equiv} H' \\ P \perp P' \\ H', P', \sigma \vDash_{TPL} B \Rightarrow H', P', \sigma \vDash_{TPL} p' \\ H'' \overset{P * P'}{\equiv} H' \\ H'', P', \sigma \vDash_{TPL} B \end{array}$$

and must prove

$$H'', P', \sigma \vDash_{TPL} p'$$

By the $\mathsf{sframed}(B)$ assumption and by Lemma 4.5, and since B is pure, we can obtain that $H', P', \sigma \vDash_{TPL} B$ and $H, P, \sigma \vDash_{TPL} B$. Therefore, we know

$$\begin{array}{l} H, P, \sigma \vDash_{TPL} \mathsf{sframed}(p') \\ H', P', \sigma \vDash_{TPL} p' \end{array}$$

and must show

$$H'', P', \sigma \vDash_{TPL} p'$$

which follows directly by definition of $\mathsf{sframed}(p')$ and the inductive hypothesis.