Fictional Separation Logic

Jonas Braband Jensen and Lars Birkedal

IT University of Copenhagen

Abstract. Separation logic formalizes the idea of local reasoning for heap-manipulating programs via the frame rule and the separating conjunction $P * Q$, which describes states that can be split into *separate* parts, with one satisfying P and the other satisfying Q. In standard separation logic, separation means physical separation. In this paper, we introduce *fictional separation logic*, which includes more general forms of fictional separating conjunctions $P * Q$, where $*$ does not require physical separation, but may also be used in situations where the memory resources described by P and Q overlap. We demonstrate, via a range of examples, how fictional separation logic can be used to reason locally and modularly about mutable abstract data types, possibly implemented using sophisticated sharing. Fictional separation logic is defined on top of standard separation logic, and both the meta-theory and the application of the logic is much simpler than earlier related approaches.

Keywords: Separation Logic, Local Reasoning, Modularity.

1 Introduction

Separation logic is a kind of Hoare logic for *local* reasoning about programs with shared mutable state. Locality is achieved by use of the $*$ connective and the frame rule:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

Recall that in standard separation logic, $P * R$ is satisfied by a heap if it can be split into two *separate* (disjoint) parts satisfying P and R respectively. The frame rule expresses that if command C is well-specified with precondition P and postcondition Q, then C will preserve any disjoint invariant R, intuitively (and formally in standard models) because of physical heap *separation*.

In many situations, however, physical separation is too strong a requirement – we would like to be able to reason locally using $*$-connectives and frame rules in situations where we do not have physical separation, but where we do have some form of logical or *fictional separation*[1]. The key idea is that fictional separation should allow us to reason separately about updates to shared resources, as long as the updates follow some kind of discipline to guarantee that updates to one

[1] The term "fictional separation" is derived from the phrase "fiction of disjointness", which, to the best of our knowledge, was introduced by Philippa Gardner [8].

side of the $*$ do not affect the truth of the other side. Permission accounting models [5, 4, 10] provide a familiar simple instance of this idea: they allow us to reason separately about shared heaps as long as we do not update but only read those heaps. In recent work on separation logic for concurrency [7, 11] and for abstraction [8, 9], it is possible to describe more elaborate patterns of sharing. We return to this when we discuss related work in Section 7.

In this paper we introduce *fictional separation logic* and demonstrate, via examples, how the logic can be used to reason locally and modularly about mutable abstract data types, possibly implemented using sophisticated sharing.

Before turning to the technical presentation, we consider a simple example.

Example: Bit Pair. Consider a small library for manipulating pointers to bit pairs. It has a constructor, destructor and some accessors that conform to the following specification in standard higher-order separation logic [2]:

$$
\begin{array}{l}
\exists B_1, B_2 : \mathit{loc} \times \mathit{bool} \to \mathcal{P}(\mathit{heap}). \\
\quad \{\mathit{emp}\}\ \mathsf{bp\_new}()\ \{B_1(\mathsf{ret}, \mathit{false}) * B_2(\mathsf{ret}, \mathit{false})\} \wedge \\
\quad \{B_1(\mathsf{p}, \_) * B_2(\mathsf{p}, \_)\}\ \mathsf{bp\_free}(\mathsf{p})\ \{\mathit{emp}\} \wedge \\
\quad \forall i \in \{1, 2\}.\ (\forall b.\ \{B_i(\mathsf{p}, b)\}\ \mathsf{bp\_get}i(\mathsf{p})\ \{B_i(\mathsf{p}, b) \wedge \mathsf{ret} = b\}) \wedge \\
\qquad\qquad \{B_i(\mathsf{p}, \_)\}\ \mathsf{bp\_set}i(\mathsf{p}, \mathsf{b})\ \{B_i(\mathsf{p}, \mathsf{b})\}.
\end{array}
$$

Note the use of existential quantification over representation predicates B_1 and B_2; they correspond to what Parkinson and Bierman call *abstract predicates* [18]. The special variable ret in postconditions denotes the return value. As usual, the underscore is used for an existentially-quantified variable.

Implementing this naïvely and verifying the implementation is straightforward in standard separation logic. Simply let the constructor allocate two consecutive heap cells and let the accessors dereference either their p parameter or $\mathsf{p} + 1$. For the verification, instantiate $B_i(p, b)$ to $(p + (i - 1)) \mapsto b$.

But this implementation uses at least twice as much heap space as necessary. The least we could do is to allocate only one (integer) heap cell and store the pair of bits in its least significant bits. A possible implementation is the following, where / denotes integer division, and % denotes modulo:

```
bp_new() { p := alloc 1; [p] := 0; return p }
bp_free(p) { free p }
bp_get1(p) { x := [p]; return x % 2 }
bp_get2(p) { x := [p]; return x / 2 }
bp_set1(p,b) { x := [p]; [p] := b + x/2*2 }
bp_set2(p,b) { x := [p]; [p] := 2*b + x%2 }
```

The original specification is unfortunately unprovable for this implementation, even though the two implementations have completely identical behaviour when observed by a client that cannot inspect their internal memory.

The problem is that the abstract module specification is not sufficiently abstract since it requires that the constructor creates two heap chunks that are

physically disjoint. In other words, the abstract module specification reveals patterns of sharing or, as is the case here, lack of sharing, that really ought to be internal to the module implementation. Moving to a heap model with bit-level separation will not solve the essence of this problem. Indeed, a third implementation could store the two bits in an integer that is divisible by 3 when B_1 is true and divisible by 5 when B_2 is true. In this case, the fictional separation comes from arithmetic properties of the integers.

In fictional separation logic, we existentially quantify not only over representation predicates B_1 and B_2, but also over the choice of separation algebra, Σ, and an interpretation map I (explained below). The abstract module specification then looks like this:

$$\begin{aligned}
&\exists \Sigma : \mathit{sepalg}.\ \exists I : \Sigma \setminus \mathit{heap}.\ \exists B_1, B_2 : \mathit{loc} \times \mathit{bool} \to \mathcal{P}(\Sigma).\\
&\quad I.\ \{\mathit{emp}\}\ \mathsf{bp_new}()\ \{B_1(\mathsf{ret}, \mathit{false}) * B_2(\mathsf{ret}, \mathit{false})\} \wedge\\
&\quad I.\ \{B_1(\mathsf{p}, _) * B_2(\mathsf{p}, _)\}\ \mathsf{bp_free}(\mathsf{p})\ \{\mathit{emp}\} \wedge\\
&\quad \forall i \in \{1, 2\}.\ (\forall b.\ I.\ \{B_i(\mathsf{p}, b)\}\ \mathsf{bp_get}i(\mathsf{p})\ \{B_i(\mathsf{p}, b) \wedge \mathsf{ret} = b\}) \wedge\\
&\qquad\qquad I.\ \{B_i(\mathsf{p}, _)\}\ \mathsf{bp_set}i(\mathsf{p}, \mathsf{b})\ \{B_i(\mathsf{p}, \mathsf{b})\}.
\end{aligned}$$

The intention is that the interpretation map I should explain how elements of the separation algebra Σ are represented by predicates on physical heaps.

Note that the Hoare triples are now prefixed by I – we refer to such a predicate $I.\ \{P\}\ C\ \{Q\}$ as an *indirect Hoare triple*. The intention is that I records (1) which separation algebra P and Q should be interpreted over, and (2) how P and Q are translated into physical heap predicates, such that the triple meaningfully corresponds to a suitably translated triple in standard separation logic.

This module specification does not reveal information about sharing or lack of sharing, because Σ and I are abstract, i.e., existentially quantified. Client code can now be verified relative to this abstract module specification and since, as we will show, fictional separation logic supports the standard proof rules (and some additional rules), the verification of client code is as easy as it is in standard separation logic. We return to this example in Section 3.2 and show how both implementations of bit pairs satisfy the above abstract specification. We will consider an example of client code verification in Section 4.1.

Outline. The remainder of this paper is organized as follows. We first present some formal preliminaries in Section 2 and then go on to present four sections on fictional separation logic. In each of these sections, we first describe some theory and then present examples that demonstrate how to use the theory in program verification. Basic fictional separation logic and the indirect triple are defined in Section 3. In Section 4 we define separating products of interpretations, which allow clients to use several modules at the same time, and in Section 5 we define a notion of indirect entailment and show how to use it to define fractional permissions within fictional separation logic. We discuss how to stack several levels of abstraction in Section 6, and we conclude and discuss related work in Section 7. To focus on the core ideas, we present fictional separation logic for

a simple sequential imperative programming language with procedures, but it should be clear that the ideas are applicable to richer programming languages.

Proofs and further examples can be found in the online appendix [14].

2 Formal Preliminaries

2.1 Abstract Assertion Logic

The meaning of separation logic assertions is often parametrized on a *separation algebra* (SA) [6], which is an abstraction of the heap model. There are several competing definitions of separation algebra in the literature [6, 10, 12]; we use the one from [12]:[2]

Definition 1. *A* separation algebra *is a partial commutative monoid* $(\Sigma, \circ, 0)$. *We write* $\sigma \doteq \sigma_1 \circ \sigma_2$ *when the* $\sigma_1 \circ \sigma_2$ *is defined and has value* σ.

Given a separation algebra $(\Sigma, \circ, 0)$, the powerset $\mathcal{P}(\Sigma)$ forms a complete boolean BI algebra, i.e., a model of the assertion language of classical separation logic, where the connectives are defined in the standard way [6]:

$$\top \triangleq \Sigma \qquad \bot \triangleq \emptyset$$
$$P \wedge Q \triangleq P \cap Q \qquad P \vee Q \triangleq P \cup Q$$
$$\forall x : A.\ P(x) \triangleq \bigcap_{x:A} P(x) \qquad \exists x : A.\ P(x) \triangleq \bigcup_{x:A} P(x)$$
$$P \Rightarrow Q \triangleq \{\sigma \mid \sigma \in P \Rightarrow \sigma \in Q\} \qquad \mathit{emp} \triangleq \{0\}$$
$$P * Q \triangleq \{\sigma \mid \exists \sigma_1, \sigma_2.\ \sigma \doteq \sigma_1 \circ \sigma_2 \wedge \sigma_1 \in P \wedge \sigma_2 \in Q\}$$
$$P \mathrel{-\!\!*} Q \triangleq \{\sigma_2 \mid \forall \sigma_1.\ \forall \sigma \doteq \sigma_1 \circ \sigma_2.\ \sigma_1 \in P \Rightarrow \sigma \in Q\}$$

As usual, entailment is defined as $P \vdash Q \triangleq P \subseteq Q$. We refer to the elements of $\mathcal{P}(\Sigma)$ as (semantic) assertions.

2.2 Programming Language

The logic we will introduce in the next section is mostly independent of the underlying programming language, but we will fix a particular language here for clarity. It is a simple imperative language in the style of [20], extended with simple procedures:

$$C ::= x := e \mid [e] := e \mid x := [e] \mid x := \mathsf{alloc}\ e \mid \mathsf{free}\ e$$
$$\mid C; C \mid \mathsf{if}\ e\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ e\ \mathsf{do}\ C \mid \mathsf{call}\ x := f(\bar{e})$$

[2] The original definition of SA [6] also required *cancellativity*: that if $\sigma' \doteq \sigma \circ \sigma_1$ and $\sigma' \doteq \sigma \circ \sigma_2$ then $\sigma_1 = \sigma_2$. This is too restrictive for our purposes, so we do not include it in the general definition of a SA.

The commands are, respectively, assignment, heap write, heap read, allocation, deallocation, sequencing, conditional, loop and function call. The argument to alloc specifies how many consecutive heap locations should be allocated.

There is no module system at the language level. When we talk about a module in this paper, it simply refers to a collection of functions.

The operational semantics of the language is defined in a standard way, using the following memory model:

$$\begin{array}{rlrl} C : \mathit{cmd} & \text{(see above)} & v : \mathit{val} & \triangleq \mathit{loc} \uplus \{\mathsf{null}\} \uplus \mathbb{Z} \uplus \{\mathsf{true}, \mathsf{false}\} \\ x, y : \mathit{var} & \triangleq \mathit{string} & s : \mathit{stack} & \triangleq \mathit{var} \to \mathit{val} \\ f : \mathit{func\_name} & \triangleq \mathit{string} & e : \mathit{expr} & \triangleq \mathit{stack} \to \mathit{val} \\ l : \mathit{loc} & \triangleq \mathbb{N} & h : \mathit{heap} & \triangleq \mathit{loc} \xrightarrow[\mathsf{fin}]{} \mathit{val} \end{array}$$
$$\mathit{program} \triangleq \mathit{func\_name} \xrightarrow[\mathsf{fin}]{} \mathit{var}^* \times \mathit{cmd} \times \mathit{expr}$$

Verification always takes place in an implicit global context of type *program* that maps each function name to a parameter list, function body and return expression. The only type of syntactic entities in this paper is *cmd*. Assertions, specifications, inference rules, and even programming language expressions, are semantic. If desired, a syntactic system could be built on top of this, but it would serve no purpose in this paper.

As usual, *heap* is a separation algebra with composition being the union of disjoint maps and the identity being the empty map. In addition to the connectives from Section 2.1, the separation algebra of heaps also has the *points-to* assertion: $l \mapsto v \triangleq \{[l \mapsto v]\}$. We make this more precise in Section 2.4.

2.3 Specification Logic

A specification $S : \mathit{spec}$ is a logical proposition about the program under consideration. The specification logic has the connectives $(\top, \bot, \wedge, \vee, \forall, \exists, \Rightarrow)$ as operators on *spec* and entailment $(\vdash)$ as a relation on *spec*. These interact according to the standard rules of intuitionistic logic.

We assume that there is a definition of the Hoare triple $\{P\}\ C\ \{Q\} : \mathit{spec}$. Intuitively, if $S \vdash \{P\}\ C\ \{Q\}$, then under the assumptions of S, if the command C runs in a state satisfying P, it will not fault, and if it terminates, the resulting state satisfies Q. The Hoare triple is assumed to satisfy the standard structural and command-specific rules of separation logic [20].

The definition of *spec* and the Hoare triple, as well as the proofs that they satisfy the rules of separation logic, are standard and not important here. See, e.g., [1] for a definition of *spec* that allows for (mutually) recursive procedures and is formalized in Coq.

The assertions P, Q used in the Hoare triple are of type $\mathit{asn}(\mathit{heap})$, where $\mathit{asn}(\Sigma) \triangleq \mathit{stack} \to \mathcal{P}(\Sigma)$. Connectives and rules for $\mathcal{P}(\Sigma)$ can be lifted pointwise to $\mathit{asn}(\Sigma)$, so we will conflate the two in the following.

2.4 Constructing Separation Algebras

In this subsection we record some simple ways of constructing separation algebras, which will be useful in the following.

Given a set A and a SA $(\Sigma, \circ, 0)$, we write $A \xrightarrow{\text{fin}} \Sigma$ for the set of total maps $f : A \to \Sigma$ for which only a finite number of values $a : A$ have $f(a) \neq 0$. That is, f has finite support. The set $A \xrightarrow{\text{fin}} \Sigma$ is itself a SA with composition being pointwise and only defined when the composition in Σ is defined at every point.

We let $[a \mapsto \sigma]$ be the map in $A \xrightarrow{\text{fin}} \Sigma$ which maps a to σ and every other element to 0. Observe that $[a \mapsto 0]$ is the constant 0 map.

For $f : A \xrightarrow{\text{fin}} \Sigma$, define $\mathit{supp}(f) = \{a \mid f(a) \neq 0\}$.

A *permission algebra* (PA) [6] is a partial commutative semigroup; i.e., it is like a SA but may not have a unit element. The product of two PAs (SAs) is also a PA (SA); composition is pointwise, and it is defined only when defined on both components. Any set A can be seen as the *empty PA* ($A_\emptyset$) by letting the composition be undefined for all operands. Moreover, any set A can be seen as the *equality PA* ($A_=$) by letting the composition have $x \circ x \doteq x$ for all x and making it undefined for non-equal operands. Finally, any PA Π can be made into a SA $\Pi_\perp$ by adding a unit element.

In this terminology, the SA of heaps is $\mathit{heap} = \mathit{loc} \xrightarrow{\text{fin}} \mathit{val}_{\emptyset\perp}$.

3 Fictional Separation Logic

The basic idea of fictional separation logic is that assertions are not just expressed in a single separation algebra, chosen in advance to match the programming language, but instead each module may define its own domain-specific SA. Each such SA is interpreted into another SA and eventually to the SA of heaps. Given separation algebras $(\Sigma, \circ_\Sigma, 0_\Sigma)$ and $(\Sigma', \circ_{\Sigma'}, 0_{\Sigma'})$, an *interpretation* I is of type

$$\Sigma \setminus \Sigma' \triangleq \{I : \Sigma \to \mathcal{P}(\Sigma') \mid I(0_\Sigma) = \{0_{\Sigma'}\}\}.$$

The side condition is not strictly necessary but will ease presentation later.[3]

The logic revolves around the *indirect triple*, defined as

$$I.\ \{P\}\ C\ \{Q\} \triangleq \forall \phi.\ \{\exists \sigma \in P.\ I(\sigma \circ \phi)\}\ C\ \{\exists \sigma \in Q.\ I(\sigma \circ \phi)\}.$$

Here I is an interpretation map of type $\Sigma \setminus \mathit{heap}$, and $P, Q : \mathit{asn}(\Sigma)$, for the same SA Σ. The triple and the all-quantifier on the right-hand side are the ones from the standard specification logic (Section 2.3).

As mentioned in Section 2.3, we implicitly lift operators and constants from $\mathcal{P}(\Sigma)$ into $\mathit{asn}(\Sigma)$. In the definition above, the $(\in)$ operator has been lifted in this way for brevity. Following usual practice, there is also an implicit assumption that the partial composition is well-defined. Written out in full detail, the precondition on the right hand side above is the following element of $\mathit{asn}(\mathit{heap})$:

$$\lambda s : \mathit{stack}.\ \{h \mid \exists \sigma \in P(s).\ \exists \sigma' \doteq \sigma \circ \phi.\ h \in I(\sigma')\}.$$

[3] It simplifies the rule CreateL from Figure 1.

The postcondition is similar, only with Q instead of P.

The quantification over all possible abstract frames ϕ bakes the frame rule into the indirect triple definition, much as in [3], except that here the frame is in a more abstract SA.

The standard specification logic structural rules of Consequence, Exists and Frame hold for the indirect triple. For brevity we just show the frame rule:

$$\frac{\mathit{modifies}(C) \cap \mathit{fv}(R) = \emptyset \qquad S \vdash I.\ \{P\}\ C\ \{Q\}}{S \vdash I.\ \{P * R\}\ C\ \{Q * R\}}\ \textsc{Frame}$$

Here, $\mathit{modifies}(C)$ is the set of program variables possibly assigned to by C [20]. The usual rules for control flow commands (if, while, call and (;)) also hold. Proofs and a discussion of the conjunction rule are in the online appendix [14].

The unit interpretation on Σ is simply $1_\Sigma \triangleq \lambda\sigma : \Sigma.\ \{\sigma\}$; it is used to relate the standard separation logic triple to the indirect triple, as expressed by the following rule:[4]

$$\frac{S \vdash 1_{heap}.\ \{P\}\ C\ \{Q\}}{S \vdash \{P\}\ C\ \{Q\}}\ \textsc{Basic}$$

We typically drop the subscript on 1_Σ since it can be inferred from the context.

3.1 Proof patterns

In this subsection we include a couple of rules and lemmas that are often useful for reasoning about examples in fictional separation logic.

In practice, pre- and postconditions are often singletons, possibly conjoined with a *pure assertion*, i.e., one that either contains every σ or no σ. The following rule relates that special case to standard separation logic. The validity of this rule follows easily from the definition of the indirect triple.

$$\frac{p, q \text{ pure} \qquad S \vdash \forall\phi.\ \{I(\sigma \circ \phi) \land p\}\ C\ \{I(\sigma' \circ \phi) \land q\}}{S \vdash I.\ \{\{\sigma\} \land p\}\ C\ \{\{\sigma'\} \land q\}}\ \textsc{Enter1}$$

The name of this rule, like all other rules in this paper, suggests reading it from the bottom up; i.e., given a proof obligation matching its conclusion, "enter" the abstract scope by exchanging the conclusion for its premise.

We will see in examples that interpretation functions often follow a particular pattern. The following lemma records useful facts about this pattern. It uses the iterated separating conjunction [20] operator ($\forall_*$), defined as

$$\forall_* a \in \{a_1, \ldots, a_n\}.\ P(a) \triangleq P(a_1) * \ldots * P(a_n).$$

Lemma 1. *If $I : (A \xrightarrow{\text{fin}} \Sigma) \searrow heap$ with $I(f) = \forall_* a \in supp(f).\ P(a, f(a))$, then*

a. $I(f) * I(g) \dashv\vdash I(f \circ g)$ *if* $supp(f) \cap supp(g) = \emptyset$.
b. $I(f) * I(g) \vdash I(f \circ g)$ *if* $\forall a.\ (P(a, \_) * P(a, \_) \vdash \bot)$.
c. If p, q are pure, then the following rule is valid.

$$\frac{S \vdash \forall\phi.\ \{I([a \mapsto \sigma \circ \phi]) \land p\}\ C\ \{I([a \mapsto \sigma' \circ \phi]) \land q\}}{S \vdash I.\ \{\{[a \mapsto \sigma]\} \land p\}\ C\ \{\{[a \mapsto \sigma']\} \land q\}}$$

[4] Double lines mean that the rule can be used both from top to bottom and vice-versa.

3.2 Example: Bit Pair

We now return to the example of bit pairs from Section 1 and explain how to prove that the implementation with sharing meets the abstract module specification. (It is obvious how the naïve implementation meets the specification: choose I to be 1_{heap} and apply Basic.)

Choose the witnesses of the existentials as follows (we convert freely between *bool* and $\{0,1\}$).

$$\begin{aligned} \Sigma &= loc \xrightarrow{\text{fin}} (\{1,2\} \xrightarrow{\text{fin}} bool_{\emptyset\perp}) \\ I(f) &= \forall_* p \in supp(f).\ supp(f(p)) = \{1,2\} \wedge p \mapsto (f(p)(1) + 2 \cdot f(p)(2)) \\ B_i(p,b) &= \{[p \mapsto [i \mapsto b]]\} \end{aligned}$$

Composition and unit in Σ follows from the constructions in Section 2.4. This resembles the SA used for object-oriented languages, where each object may have several fields.

The intuition behind the choice of I is that by requiring the support at each point to be the full set, i.e., $\{1,2\}$, we can control what we expect to find in the frame ϕ. For a function such as bp_get1, this means that since $B_1(\mathsf{p}, b)$ is assumed in the precondition, we are sure to find $B_2(\mathsf{p}, b')$ in the frame, for some b'. Showing that the frame is preserved then amounts to showing that the frame also contains $B_2(\mathsf{p}, b')$ after executing the function body – and showing that any other $p' \neq \mathsf{p}$ mentioned in the frame is unaffected, but we will see below that Lemma 1 on pointwise interpretation functions makes that easy.

We now have to show that the implementation of each function matches its specification with this choice of witnesses for the existentials. This is straightforward, because every function in this example has a specification that matches a combination of Enter1 and Lemma 1. These rules reduce the proof obligations to statements in standard separation logic.

First note that the following *saturation lemma* holds for the interpretation map for bit pairs:

$$I([\_ \mapsto [i \mapsto b] \circ \phi]) \vdash \exists b'.\ \phi = [3-i \mapsto b'].$$

We present here the proof of bp_get1. Let $C = (\mathsf{x} := [\mathsf{p}])$, i.e., the body of bp_get1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall b_2.\ \{\mathsf{p} \mapsto (b + 2 \cdot b_2)\}\ C\ \{\mathsf{p} \mapsto (b + 2 \cdot b_2) \wedge \mathsf{x}\%2 = b\}}\text{(trivial)}}{\forall b_2.\ \{I([\mathsf{p} \mapsto [1 \mapsto b, 2 \mapsto b_2]])\}\ C\ \{I([\mathsf{p} \mapsto [1 \mapsto b, 2 \mapsto b_2]]) \wedge \mathsf{x}\%2 = b\}}\text{(definition)}}{\forall \phi.\ \{I([\mathsf{p} \mapsto [1 \mapsto b] \circ \phi])\}\ C\ \{I([\mathsf{p} \mapsto [1 \mapsto b] \circ \phi]) \wedge \mathsf{x}\%2 = b\}}\text{Saturation}}{I.\ \{\{[\mathsf{p} \mapsto [1 \mapsto b]]\}\}\ C\ \{\{[\mathsf{p} \mapsto [1 \mapsto b]]\} \wedge \mathsf{x}\%2 = b\}}\text{Lemma 1c}}{I.\ \{B_1(\mathsf{p}, b)\}\ C\ \{B_1(\mathsf{p}, b) \wedge \mathsf{x}\%2 = b\}}\text{(definition)}$$

Notice that the overhead of showing that the abstract specification is met is fairly small and straightforward.

3.3 Example: Monotonic Counter

A monotonic counter is an integer stored in the heap with operations for reading it and incrementing it but not for decrementing it. The implementation could look like this:

$$\begin{array}{l}\mathsf{mc\_new()}\ \{\ \mathsf{c} := \mathsf{alloc}\ 1;\ [\mathsf{c}] := 0;\ \mathsf{return}\ \mathsf{c}\ \}\\ \mathsf{mc\_read(c)}\ \{\ \mathsf{x} := [\mathsf{c}];\ \mathsf{return}\ \mathsf{x}\ \}\\ \mathsf{mc\_inc(c)}\ \{\ \mathsf{x} := [\mathsf{c}];\ [\mathsf{c}] := \mathsf{x}{+}1\ \}\end{array}$$

Reasoning about monotonic counters was posed as a verification challenge by Pilkiewicz and Pottier [19]. They discussed the challenge in a type-and-capability system, so the presentation is somewhat different than for separation logic, but the idea is the same. The counter should have a representation predicate $MC(c, i)$ that can be freely duplicated; i.e., $MC(c, i) \vdash MC(c, i) * MC(c, i)$. It should be possible to frame out one of the copies while the other copy is used to call the increment function; when the first copy is later framed back in, it can soundly be used to call the read function since its postcondition only guarantees that the returned value is *at least* the value from the representation predicate.

The specification in fictional separation logic looks like this:

$$\begin{array}{l}\exists \Sigma : \mathit{sepalg}.\ \exists I : \Sigma \searrow \mathit{heap}.\ \exists MC : \mathit{loc} \times \mathbb{Z} \to \mathcal{P}(\Sigma).\\ \quad (\forall c, j.\ \forall i \le j.\ (MC(c, j) \dashv\vdash MC(c, j) * MC(c, i))) \wedge\\ \quad I.\ \{\mathit{emp}\}\ \mathsf{mc\_new()}\ \{MC(\mathsf{ret}, 0)\} \wedge\\ \quad (\forall i.\ I.\ \{MC(\mathsf{c}, i)\}\ \mathsf{mc\_read(c)}\ \{MC(\mathsf{c}, i) \wedge \mathsf{ret} \ge i\}) \wedge\\ \quad (\forall i.\ I.\ \{MC(\mathsf{c}, i)\}\ \mathsf{mc\_inc(c)}\ \{MC(\mathsf{c}, i + 1)\}).\end{array}$$

The fact about MC has several corollaries that are useful for clients:

$$\begin{array}{rl} MC(c, i) & \dashv\vdash MC(c, i) * MC(c, i)\\ i \le j \wedge MC(c, j) & \vdash MC(c, i) * \top\\ MC(c, i) * MC(c, j) & \vdash MC(c, \mathit{max}(i, j))\end{array}$$

Compared to the solution by Pilkiewicz and Pottier, this solution has several advantages. Our solution can be specified and verified without changing the implementation to account for limitations in the verification system [19, end of Sect. 4]. Moreover, it can be verified in the simple system of fictional separation logic, whereas there exists no soundness proof yet for the very complicated type system used by Pilkiewicz and Pottier.

To verify our specification against the implementation shown above, choose the existentials as follows:

$$\begin{array}{rl} \Sigma = \mathit{loc} \xrightarrow{\text{fin}} \mathbb{Z}_\perp & \text{where composition in } \mathbb{Z} \text{ is } \mathit{max}\\ I(f) = \forall_*\, c \in \mathit{supp}(f).\ \exists j \ge f(c).\ c \mapsto j & \\ MC(c, i) = \{[c \mapsto i]\} & \end{array}$$

The property about MC is straightforward to verify in the assertion logic. Verification of the three functions is shown in the online appendix [14].

$$\frac{S \vdash I * J.\ \{P^{\mathsf{L}}\}\ C\ \{Q^{\mathsf{L}}\}}{S \vdash I.\ \{P\}\ C\ \{Q\}}\ \textsc{CreateL} \qquad \frac{S \vdash I.\ \{P\}\ C\ \{Q\}}{S \vdash I * J.\ \{P^{\mathsf{L}}\}\ C\ \{Q^{\mathsf{L}}\}}\ \textsc{ForgetL}$$

$$\frac{S \vdash I * J.\ \{P^{\mathsf{L}}\}\ C\ \{Q \times \top\}}{S \vdash I * 1.\ \{P^{\mathsf{L}}\}\ C\ \{Q \times \top\}}\ \textsc{LeakL} \qquad \frac{p \text{ pure}}{(p \wedge P) \times Q \dashv\vdash p \wedge (P \times Q)}\ \textsc{Prod-Pure}$$

Fig. 1. Selected inference rules for separating products, using notation $P^{\mathsf{L}} \triangleq P \times emp$

There are some limitations in the specification. There can be no function to deallocate a counter because its representation predicate can be freely shared. The absence of deallocation means that this specification is more suited for a garbage-collected language. Also, the specification does not guarantee that consecutive calls to mc_read will return the same value; it would be valid to implement mc_read such that it has the side effect of incrementing the counter. These limitations are also present in the specification by Pilkiewicz and Pottier.

4 Clients and Separating Products

To allow clients of multiple libraries to know about more than one separation algebra and interpretation function, we introduce *separating products* of interpretations.

Given interpretations $I_1 : \Sigma_1 \searrow \Sigma$ and $I_2 : \Sigma_2 \searrow \Sigma$, their separating product $I_1 * I_2$ has type $\Sigma_1 \times \Sigma_2 \searrow \Sigma$ and is defined as

$$I_1 * I_2 \triangleq \lambda(\sigma_1, \sigma_2).\ I_1(\sigma_1) * I_2(\sigma_2).$$

Figure 1 shows a collection of inference rules about separating products. At the bottom of a proof tree, just above application of Basic, a client should use CreateL for each module that will be used. In that rule, $P^{\mathsf{L}} \triangleq P \times emp$, where $(\times)$ is simply the Cartesian product lifted into *asn*. To write that out, $P_1 \times P_2 \triangleq \lambda s.\ \{(\sigma_1, \sigma_2) \mid \sigma_1 \in P_1(s) \wedge \sigma_2 \in P_2(s)\}$.

The CreateL rule requires the command C to clean up the state abstracted by J completely; i.e., that state must satisfy *emp* in the postcondition. When this is not possible, for example in the Monotonic Counters example, we can instead use the LeakL rule.

Before calling a library function, a client will, as usual, have to frame out irrelevant facts. There, it can be useful to know that $(P * Q)^{\mathsf{L}} \dashv\vdash P^{\mathsf{L}} * Q^{\mathsf{L}}$ and that $P \times Q \dashv\vdash P^{\mathsf{L}} * Q^{\mathsf{R}}$, where $P^{\mathsf{R}} \triangleq emp \times P$. After applying the frame rule, the client can then ignore the irrelevant separation algebras using the ForgetL rule, which is just the CreateL rule inverted.

Pure assertions can move in and out of products as described by the Prod-Pure rule. There are of course rules CreateR, ForgetR and LeakR symmetric to the ones in Figure 1, and further structural rules can be defined to handle commutativity and associativity with separating products.

4.1 Example: Client of Two Modules

Assume we have a client program C that uses both the bit pair and the monotonic counter modules, and we want to show that it has precondition *emp* and postcondition $\top$. We suggest $\top$ in the postcondition because there is no deallocation function for monotonic counters as mentioned earlier.

The standard pattern for this is to assume the module specifications at the bottom of the tree and then move from the standard triple to the appropriate indirect triple by applying Basic once and then Create or Leak for each module, reading from the bottom up. Abbreviate the bit pair and monotonic counter specifications, minus the existentials, as S_{bp} and S_{mc} respectively. The bottom of the proof for C then looks like this.

$$\dfrac{\dfrac{\dfrac{\dfrac{S_{\mathrm{bp}} \wedge S_{\mathrm{mc}} \vdash I_{\mathrm{bp}} * I_{\mathrm{mc}}.\ \{emp \times emp\}\ C\ \{emp \times \top\}}{S_{\mathrm{bp}} \wedge S_{\mathrm{mc}} \vdash I_{\mathrm{bp}} * 1.\ \{emp \times emp\}\ C\ \{emp \times \top\}}\ \textsc{LeakL}}{S_{\mathrm{bp}} \wedge S_{\mathrm{mc}} \vdash 1.\ \{emp\}\ C\ \{\top\}}\ \textsc{CreateR}}{(\exists \Sigma, I_{\mathrm{bp}}, B_1, B_2.\ S_{\mathrm{bp}}) \wedge (\exists \Sigma, I_{\mathrm{mc}}, MC.\ S_{\mathrm{mc}}) \vdash \{emp\}\ C\ \{\top\}}\ \textsc{Basic}}\ \exists\mathrm{L}$$

The bottom proof step applies the standard existential-left rule from sequent calculus twice.

If C uses the heap directly, not just through the two modules, it should apply Create once more to get the interpretation $I_{\mathrm{bp}} * I_{\mathrm{mc}} * 1$ on the indirect triple.

Further up in the proof tree, there will eventually be a point where it is necessary to call a function belonging to one of the modules, e.g., the bit pairs. The following pattern is used to ignore irrelevant modules during the call.

$$\dfrac{\dfrac{\dfrac{S \vdash I_{\mathrm{bp}}.\ \{P\}\ \mathsf{call}\ f\ \{Q\}}{S \vdash I_{\mathrm{bp}} * I_{\mathrm{mc}}.\ \{P^{\mathsf{L}}\}\ \mathsf{call}\ f\ \{Q^{\mathsf{L}}\}}\ \textsc{ForgetL}}{S \vdash I_{\mathrm{bp}} * I_{\mathrm{mc}}.\ \{P^{\mathsf{L}} * {R_1}^{\mathsf{L}} * {R_2}^{\mathsf{R}}\}\ \mathsf{call}\ f\ \{Q^{\mathsf{L}} * {R_1}^{\mathsf{L}} * {R_2}^{\mathsf{R}}\}}\ \textsc{Frame}}{S \vdash I_{\mathrm{bp}} * I_{\mathrm{mc}}.\ \{(P * R_1) \times R_2\}\ \mathsf{call}\ f\ \{(Q * R_1) \times R_2\}}\ \textsc{Consequence}$$

Note that this kind of reasoning will not be so explicit in practice; a simple tool can easily elide these steps.

In this section we considered a generic client; see the online appendix [14] for a concrete example client using bit pairs and monotonic counters.

4.2 Example: Wrapper

This example demonstrates how a module can extend the abstraction of another module by using a separating product. We will see that this example gives a compelling argument against solving the fiction-of-disjointness problem by letting the client carry around an explicit but opaque frame as done in [16, Chapter 5].

Consider first the following specification of a collection data structure.

$$
\begin{aligned}
S_{\mathrm{Coll}}(\Sigma : \mathit{sepalg}, I : \Sigma \setminus \mathit{heap}, \mathit{Coll} : \mathit{loc} \times \mathcal{P}_{\mathsf{fin}}(\mathit{val}) \to \mathcal{P}(\Sigma)) \triangleq \forall c, V.\\
&(\mathit{Coll}(c, \_) * \mathit{Coll}(c, \_) \vdash \bot) \wedge\\
&I.\, \{\mathit{emp}\}\ \mathsf{coll\_new}()\ \{\mathit{Coll}(\mathsf{ret}, \emptyset)\} \wedge\\
&I.\, \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_free}(\mathsf{c})\ \{\mathit{emp}\} \wedge\\
&I.\, \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_clone}(\mathsf{c})\ \{\mathit{Coll}(\mathsf{c}, V) * \mathit{Coll}(\mathsf{ret}, V)\} \wedge\\
&I.\, \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_contains}(\mathsf{c}, \mathsf{v})\ \{\mathit{Coll}(\mathsf{c}, V) \wedge \mathsf{ret} = (\mathsf{v} \in V)\} \wedge\\
&I.\, \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_add}(\mathsf{c}, \mathsf{v})\ \{\mathit{Coll}(\mathsf{c}, V \cup \{\mathsf{v}\})\} \wedge\\
&I.\, \{\mathit{Coll}(\mathsf{c}, V)\}\ \mathsf{coll\_remove}(\mathsf{c}, \mathsf{v})\ \{\mathit{Coll}(\mathsf{c}, V \setminus \{\mathsf{v}\})\}.
\end{aligned}
$$

This is a standard specification of a finite collection, except for the coll_clone function. This function could be implemented by simply copying the contents of the collection to a new data structure; in standard separation logic, that would be the only possible implementation because of the $(*)$ in the postcondition.

In fictional separation logic, it might also be implemented by using *copy on write* – the reference-counting technique in which the contents are initially shared between two collections and only copied when the need arises because one of them is modified [17]. The purpose of including coll_clone here is to have a reason why this library should be specified with fictional separation logic.

Consider now a wrapper module of indirect references to collections. The implementation could be this:

```
wcoll_new(c) { w := alloc 1; [w] := c; return w }
wcoll_contains(w,v) { c := [w]; return coll_contains(c,v) } ...
```

Functions wcoll_add, wcoll_remove and wcoll_free would be implemented analogously to wcoll_contains, forwarding the call. A more useful wrapper module would, of course, add some functionality, such as caching the last query to wcoll_contains or counting the number of calls to wcoll_add, but the essence remains the same.

We can give the following specification to this code.

$$
\begin{aligned}
&\forall \Sigma, I, \mathit{Coll}.\ S_{\mathrm{Coll}}(\Sigma, I, \mathit{Coll}) \Rightarrow\\
&\quad \exists \mathit{WColl} : \mathit{loc} \times \mathcal{P}_{\mathsf{fin}}(\mathit{val}) \to \mathcal{P}(\mathit{heap} \times \Sigma).\ \forall V.\\
&\quad\ 1 * I.\, \{\mathit{Coll}(\mathsf{c}, V)^{\mathsf{R}}\}\ \mathsf{wcoll\_new}(\mathsf{c})\ \{\mathit{WColl}(\mathsf{ret}, V)\} \wedge\\
&\quad\ 1 * I.\, \{\mathit{WColl}(\mathsf{w}, V)\}\ \mathsf{wcoll\_contains}(\mathsf{w}, \mathsf{v})\ \{\mathit{WColl}(\mathsf{w}, V) \wedge \mathsf{ret} = (\mathsf{v} \in V)\} \wedge \ldots
\end{aligned}
$$

Observe that this is an example of one specification depending on another, by being universal in the parameters of the S_{Coll} specification from above. (See [1] for more general cases of this design pattern, in standard higher-order separation logic.) For the example implementation above, the proof of the specification should instantiate the existential as $\mathit{WColl}(w, V) = \exists c.\ w \mapsto c \times \mathit{Coll}(c, V)$. As a side remark, this specification could be made more abstract, so that it would

reveal less implementation detail, by hiding the use of the 1-interpretation behind an existential.

The specification of the constructor, wcoll_new, is an example of ownership transfer: ownership of memory described by an abstract predicate ($Coll(\mathsf{c}, V)$) is transferred from the caller to the module. The specification intentionally does not reveal whether the transfer happened simply by storing a pointer, as in our example implementation, or whether the constructor allocated a new collection (or another container data structure), manually copied the contents of the given collection to that, and then freed the given collection.

For comparison, to mimic this in standard separation logic, one could take inspiration from Krishnaswami's design pattern [16, Chapter 5] and let the representation predicate of the collection module describe all the collections that may share data; i.e., $H : \mathcal{P}_{\mathsf{fin}}(loc \times \mathcal{P}_{\mathsf{fin}}(val)) \to \mathcal{P}(heap)$. The constructor specification would then be along the lines of the following, where $\uplus$ denotes union of sets of tuples with disjoint first components, and $Coll'(c, V) \triangleq \{(c, V)\}$.

$$\{H(Coll'(\mathsf{c}, V) \uplus \phi)\}\ \mathsf{wcoll\_new}(\mathsf{c})\ \{\exists\sigma.\ H(\sigma \uplus \phi) * WColl'(\mathsf{ret}, \sigma, V)\}.$$

This specification allows the same implementation freedom as the fictional separation logic version does, and the caller is guaranteed that the abstract frame ϕ is preserved. But it is a completely undesirable specification in practice because the $WColl'$ predicate can never be detached from the H predicate that it may share data with. This means that all the accessor functions must have both $WColl'$ and H in their pre- and postconditions. Even worse, clients need to keep track of the opaque σ that links the two together.

5 Indirect Entailment

There is no restriction that a physical heap can only be in the image of a single abstract σ. Therefore we can sometimes change abstract pre- and postconditions in a more powerful way than what the rule of consequence allows; we present an application of this idea in the next subsection. First, we define *indirect entailment*:

$$P \models_I Q \triangleq \forall\phi.\ \big((\exists\sigma \in P.\ I(\sigma \circ \phi)) \vdash (\exists\sigma \in Q.\ I(\sigma \circ \phi))\big).$$

We can now state the *indirect rule of consequence*:

$$\frac{P \models_I P' \qquad S \vdash I.\ \{P'\}\ C\ \{Q'\} \qquad Q' \models_I Q}{S \vdash I.\ \{P\}\ C\ \{Q\}}\ \text{ROC-Indirect}$$

Its correctness is immediate from the definitions.

The definition of indirect entailment is quite similar to the indirect triple. In fact, if $I : \Sigma \setminus heap$ for some Σ, then $P \models_I Q$ if and only if $\vdash I.\ \{P\}\ skip\ \{Q\}$.

For any I, the relation $(\models_I)$ is reflexive and transitive and is a superrelation of $(\vdash)$. Judgements $P \models_I Q$ can also be studied as a kind of degenerate assertion logic; in that case, the standard natural-deduction introduction and elimination rules for $(\top, \bot, \vee, \exists)$ hold, and so do $(\Rightarrow)$-introduction and $(\wedge, \forall)$-elimination. It

is also possible to reason locally on both sides of a separating conjunction, and the existential-left rule holds; i.e.,

$$\frac{P \models_I P' \qquad Q \models_I Q'}{P * Q \models_I P' * Q'} \qquad\qquad \frac{\forall x.\ (P(x) \models_I Q)}{\exists x.\ P(x) \models_I Q}$$

We discuss more rules for $(\models_I)$ in the online appendix [14].

5.1 Example: Fractional Permissions

Permission accounting [5, 4, 10] is a solution to simple sharing problems where just read-only data is shared. The points-to predicate is generalized to carry a permission, so $l \overset{z}{\mapsto} v$ denotes a z-permission to access heap location l. If z is a read-only permission, then there are no write permissions to l available to others, and therefore its value stays v. If z is a write permission, then there are no other read or write permissions for l available to others.

A write permission can be split across the $*$ into several read-only permissions. If it is known that all read-only permissions have been accounted for, then they can be re-assembled into a write permission. Permissions are clearly useful for sharing data read-only between threads in concurrent programs, but it also has uses in a sequential setting [13, 15].

We will now show how fractional permissions, a particular permission accounting scheme, can be encoded in fictional separation logic. This allows us to use fractional permissions where we need it, without having fractional permissions in the base logic! A permission z is a rational number satisfying $0 < z \leq 1$. The write permission is 1, and all smaller numbers are read-only permissions. We will define the assertion $l \overset{z}{\mapsto} v$ such that the splitting and joining of permissions can be described by the following inference rule.

$$\frac{}{l \overset{z_1}{\mapsto} v_1 * l \overset{z_2}{\mapsto} v_2 \dashv\vdash v_1 = v_2 \land z_1 + z_2 \leq 1 \land l \overset{z_1+z_2}{\longmapsto} v_1} \textsc{Fractions}$$

We first define the SA of heaps with fractional permissions as usual [4]:

$$\begin{aligned} \Sigma_{\text{fp}} &\triangleq \mathit{Ptr} \overset{\text{fin}}{\rightharpoonup} (\mathit{Val}_= \times \mathit{Perm})_\bot, \text{ where} \\ \mathit{Perm} &\triangleq \{z : \mathbb{Q} \mid 0 < z \leq 1\} \\ z_1 \dot{+} z_2 &\triangleq \begin{cases} z_1 + z_2 & \text{if } z_1 + z_2 \leq 1 \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

Since $(\mathit{Perm}, \dot{+})$ is a permission algebra, Σ_{fp} is a separation algebra (see Section 2.4). We define the fractional points-to predicate by $l \overset{z}{\mapsto} v \triangleq \{[l \mapsto (v, z)]\}$ and can then easily verify the Fractions rule.

To make use of all this, we define an interpretation $I_{\text{fp}} : \Sigma_{\text{fp}} \searrow \mathit{heap}$. The idea is the same as for the interpretation function in the bit pair example (Section 3.2): assume we have the full knowledge (permission) for each described heap location.

$$I_{\text{fp}}(f) \triangleq \forall_* l \in \mathit{supp}(f).\ \exists v.\ f(l) = (v, 1) \land l \mapsto v.$$

We can now prove a specification of the heap read command for fractional permissions. For clarity, let us just consider the case where the variable name being assigned to is fresh (formally we treat free variables as in [1]):

$$\frac{x \notin fv(e, e')}{\vdash I_{\text{fp}}.\ \{e \overset{z}{\mapsto} e'\}\ x := [e]\ \{e \overset{z}{\mapsto} e' \land x = e'\}}$$

Let us sketch the proof of this rule. We first expand the definition of the fractional points-to predicate in the conclusion:

$$\vdash I_{\text{fp}}.\ \{\{[e \mapsto (e', z)]\}\}\ x := [e]\ \{\{[e \mapsto (e', z)]\} \land x = e'\}.$$

By applying Lemma 1c, we can reduce this to the proof obligation

$$\vdash \forall \phi.\ \{I_{\text{fp}}([e \mapsto (e', z) \circ \phi])\}\ x := [e]\ \{I_{\text{fp}}([e \mapsto (e', z) \circ \phi]) \land x = e'\},$$

which is now a statement in standard separation logic that can be discharged using a saturation lemma like in Section 3.2. Intuitively, I_{fp} in the precondition gives us the points-to predicate needed for applying the standard read rule. The postcondition requires us to prove that I_{fp} holds for the same parameter, which is easy since the heap did not change.

We can also prove the write and allocation rules using the above approach, but we will instead show how to use indirect entailment to get even simpler proofs. The following indirect bi-entailment expresses that having the full permission to a location ($z = 1$) is the same as having a standard points-to predicate for it:

$$\frac{}{(l \overset{1}{\mapsto} v)^{\mathsf{L}} \dashv\models_{I_{\text{fp}} * 1} (l \mapsto v)^{\mathsf{R}}}$$

With this lemma, proved in the online appendix [14], the write and allocation rules follow almost immediately from their standard separation logic versions. For instance, the fractional write rule is derived as follows.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash 1.\ \{e \mapsto \_\}\ [e] := e'\ \{e \mapsto e'\}}\ \textsc{Basic, Write-std}}{\vdash I_{\text{fp}} * 1.\ \{(e \mapsto \_)^{\mathsf{R}}\}\ [e] := e'\ \{(e \mapsto e')^{\mathsf{R}}\}}\ \textsc{ForgetR}}{\vdash I_{\text{fp}} * 1.\ \{(e \overset{1}{\mapsto} \_)^{\mathsf{L}}\}\ [e] := e'\ \{(e \overset{1}{\mapsto} e')^{\mathsf{L}}\}}\ \textsc{ROC-Indirect}}{\vdash I_{\text{fp}}.\ \{e \overset{1}{\mapsto} \_\}\ [e] := e'\ \{e \overset{1}{\mapsto} e'\}}\ \textsc{CreateL}$$

6 Stacking

Intuitively, fictional separation logic allows us to pretend we are working in a high-level memory model Σ if we show how to interpret that high-level memory model down to *heap*. It is then natural to investigate whether we can stack an even higher-level memory model Σ' on top of that construction and interpret Σ' down to Σ. Of course, this should generalize to arbitrary levels of stacking.

In this section, we present a theory of stacking that allows this while interacting well with the features introduced in previous sections and not being

a burden on the logic when not in use. It is important to stress that it is in many cases possible for one module to depend on and extend the abstraction of another module *without* using stacking; c.f. the wrapper example in Section 4.2.

The most basic way to combine two interpretations is to compose them as relations. Given interpretations $I : \Sigma_1 \setminus \Sigma_2$ and $J : \Sigma_2 \setminus \Sigma_3$, their relational composition $(I\,;J)$ has type $\Sigma_1 \setminus \Sigma_3$ and is defined as

$$I\,;J \triangleq \lambda\sigma_1.\ \exists\sigma_2' \in I(\sigma_1).\ J(\sigma_2').$$

That is, $\sigma_3 \in (I\,;J)(\sigma_1)$ if and only if $\exists\sigma_2' \in I(\sigma_1).\ \sigma_3 \in J(\sigma_2')$.

We can show the following rule for working with relational composition:

$$\frac{S \vdash \forall\phi.\ J.\ \{\exists\sigma \in P.\ I(\sigma \circ \phi)\}\ C\ \{\exists\sigma \in Q.\ I(\sigma \circ \phi)\}}{S \vdash (I\,;J).\ \{P\}\ C\ \{Q\}}\ \textsc{RelComp}$$

Reading the rule from the bottom up, RelComp allows peeling off the top layer of a multi-layered interpretation, making its frame explicit. This is desirable when verifying an implementation that extends upon the J-interpretation using I. Perhaps J is opaque at the point where this rule is applied.

Relational composition masks the J-interpretation to the outside; in particular, it masks the frame in Σ_2 that goes into J, which means that the converse of RelComp does not hold. In many situations, including the next example, we want to give specifications that expose both the Σ_1-algebra and the Σ_2-algebra in order to be useful to clients that do not have their data exclusively in Σ_1. This discussion motivates our definition of the *stacking* composition. Given interpretations $I : \Sigma_1 \setminus \Sigma_2$ and $J : \Sigma_2 \setminus \Sigma_3$, their stacking $I \succ J$ has type $\Sigma_1 \times \Sigma_2 \setminus \Sigma_3$ and is defined as

$$I \succ J \triangleq (I * 1)\,;J.$$

With this definition, we now get a generalization of RelComp in the form of the following rule, which holds in both directions:

$$\frac{S \vdash \forall\phi.\ J.\ \{\exists\sigma \in P_1.\ I(\sigma \circ \phi) * P_2\}\ C\ \{\exists\sigma \in Q_1.\ I(\sigma \circ \phi) * Q_2\}}{S \vdash I \succ J.\ \{P_1 \times P_2\}\ C\ \{Q_1 \times Q_2\}}\ \textsc{StackComp}$$

The special case of this rule where $P_2 = Q_2 = \mathit{emp}$ is similar to RelComp. The special case where $P_1 = Q_1 = \mathit{emp}$ leads to a rule that is more like a stacking version of Forget and Create (Section 4).

There is a generalization of the Enter1 rule to stacking:

$$\frac{S \vdash \forall\phi.\ J.\ \{I(\sigma \circ \phi) * P\}\ C\ \{I(\sigma' \circ \phi) * Q\}}{S \vdash I \succ J.\ \{\{\sigma\} \times P\}\ C\ \{\{\sigma'\} \times Q\}}\ \textsc{Enter1Stack}$$

This rule is simply a special case of StackComp. Notice that $(I \succ 1) = (I * 1)$, so these inference rules can also be applied to separating products in some cases.

A module may use stacking internally but hide that fact if the stacking does not need to be visible to its clients. This can be achieved by collapsing the stacking composition to a relational composition by the following rule:

$$\frac{S \vdash I \succ J.\ \{P^{\mathsf{L}}\}\ C\ \{Q^{\mathsf{L}}\}}{S \vdash (I\,;J).\ \{P\}\ C\ \{Q\}}\ \textsc{StackRel}$$

6.1 Example: Abstract Fractional Permissions

We saw the example of fractional permissions in Section 5.1, where the points-to predicate was extended to carry a permission. With stacking, we can use essentially the same construction to extend the *Coll* predicate from Section 4.2 to carry a permission. Just like the heap-read command could execute with any partial permission, while heap write required full permission, we can prove that coll_clone and coll_contains can execute with any partial permission, while the other functions require full permission.

Formally, we can prove the validity of the following specification.

$$
\begin{array}{l}
\forall \Sigma, I, \mathit{Coll}.\ S_{\mathrm{Coll}}(\Sigma, I, \mathit{Coll}) \Rightarrow \\
\quad \exists \Sigma' : \mathit{sepalg}.\ \exists I' : \Sigma' \searrow \Sigma. \\
\quad \exists \mathit{FColl} : \mathit{Perm} \times \mathit{loc} \times \mathcal{P}_{\mathsf{fin}}(\mathit{val}) \to \mathcal{P}(\Sigma').\ \forall V, V', c, z, z'. \\
\qquad (\mathit{FColl}^{z}(c, V) * \mathit{FColl}^{z'}(c, V') \dashv\vdash V = V' \wedge z + z' \leq 1 \wedge \mathit{FColl}^{z+z'}(c, V)) \wedge \\
\qquad (\mathit{FColl}^{1}(c, V)^{\mathsf{L}} \mathrel{=\!\!|\!\models}_{I' \succ I} \mathit{Coll}(c, V)^{\mathsf{R}}) \wedge \\
\qquad I' \succ I.\ \{\mathit{FColl}^{z}(\mathsf{c}, V)^{\mathsf{L}}\}\ \mathsf{coll\_contains}(\mathsf{c}, \mathsf{v})\ \{\mathit{FColl}^{z}(\mathsf{c}, V)^{\mathsf{L}} \wedge \mathsf{ret} = (\mathsf{v} \in V)\} \wedge \\
\qquad I' \succ I.\ \{\mathit{FColl}^{z}(\mathsf{c}, V)^{\mathsf{L}}\}\ \mathsf{coll\_clone}(\mathsf{c})\ \{\mathit{FColl}^{z}(\mathsf{c}, V)^{\mathsf{L}} * \mathit{FColl}^{1}(\mathsf{ret}, V)^{\mathsf{L}}\}.
\end{array}
$$

The elements of the specification are all the same as for the standard fractional permissions in Section 5.1. It is written such that the stacking is revealed to clients, allowing them to use the fractional and non-fractional collections together and convert between them using the indirect bi-entailment in the specification. There is thus no need for specifying fractional versions of the remaining functions since the indirect bi-entailment allows reusing the original specifications.

Notice that we can define *FColl* and prove fractional versions of all the functions without knowing their implementation or how *Coll*, I or Σ are defined. In particular, the implementations of coll_clone and coll_contains are allowed to modify the underlying heap, but they still appear read-only through the indirect specification.

If it is not necessary for fractional and non-fractional collections to coexist and share footprints from the perspective of clients, the StackRel rule could be used to hide the stacking in this specification. Then the specification can be made to look as simple as the original specification in Section 4.2 by hiding $(I' ; I)$ behind an existential.

We can verify the specification by choosing the existentials as follows:

$$
\begin{aligned}
\Sigma' &= \mathit{loc} \xrightarrow{\mathsf{fin}} (\mathcal{P}_{\mathsf{fin}}(\mathit{val})_{=} \times \mathit{Perm})_{\bot} \\
I'(f) &= \forall_{*}\, c \in \mathit{supp}(f).\ \exists V.\ f(c) = (V, 1) \wedge \mathit{Coll}(c, V) \\
\mathit{FColl}^{z}(c, V) &= \{[c \mapsto (V, z)]\}
\end{aligned}
$$

This is very similar to the original fractional permissions example, and the proofs are also similar.

7 Discussion and Related Work

Simplicity has been a major goal for this theory, particularly in three places: (1) clients of a module that uses fictional separation internally should be able to reason with the same ease as in standard separation logic; (2) the overhead in verifying an implementation with fictional separation should be minimal; and (3) correctness of the meta-theory should be easy to prove. The three goals are listed in order of importance since they represent tasks to be carried out by a decreasing number of people.

We believe that the first simplicity goal has been achieved in most situations, though clients of multiple modules with complex stacking patterns may benefit from tool support for composing the interpretations. The second goal has been achieved in the sense that it is easy to verify examples like those presented in this paper and, moreover, the separation algebras needed can be assembled from standard constructions. The third goal has been reached through judicious choice of definitions, especially by defining the indirect triple in terms of the standard triple – it has been possible to conduct all meta-theoretical proofs without unfolding the definition of the standard triple [14]. Because we work directly in the semantics of the logic, it should be natural to encode this theory in the Coq proof assistant, extending our existing Coq formalization of separation logic [1].

A major inspiration for fictional separation logic has been the design pattern used by Krishnaswami [16, Chapter 5] for specifying data structures with fictional disjointness in standard separation logic. The technique is to define a per-module custom separation logic (not separation *algebra*) and let the client manage the abstract frame, which is explicitly present in every function specification. Fictional separation logic makes the essential part of this design pattern formal, allowing the abstract frame to be managed implicitly by the indirect triple and enabling a general and comprehensive theory on these custom separation logics, instead of scattering the theory across modules on an ad-hoc basis. See also the discussion in Section 4.2. We ignore Krishnaswami's concept of a *ramification operator* since it would make the resulting logic too different from separation logic.

The work on *locality-breaking transformations* (LBT) for context logic, a kind of non-commutative separation logic, by Dinsdale-Young, Gardner and Wheelhouse [8, 9] can also be seen as a formalization of Krishnaswami's design pattern, though they were developed independently. LBT is in the field of program refinement, which means that not only are pre- and postconditions of a triple transformed across abstraction layers, like in fictional separation logic, but the command is also transformed. Despite that difference, the intuition and proof obligations are similar to fictional separation logic: verifying a module implementation involves showing that all atomic operations preserve an abstract frame from a per-module context algebra. Reasoning in LBT is fundamentally in two stages, though: a client program and proof are always created at the high level and are subsequently transformed to the low level outside the logic. In fictional separation logic, moving between the levels is done within the logic itself, and the

separation algebras are first-class entities in the logic. Hence, as we have seen, we can take advantage of all the features in the specification logic, e.g., to hide the definition of a separation algebra behind an existential quantifier. The soundness proofs of the meta-theory in [9] are much more complicated than ours, despite their much less expressive specification logic; it appears to be caused by their proof-theoretic approach to soundness as opposed to our semantic approach.

In terms of what examples can be encoded, fictional separation logic is quite close to the *concurrent abstract predicates* (CAP) framework [7, 11] restricted to sequential programs. CAP has been developed for reasoning about concurrent programs in which several threads may work on the same shared memory; when restricting attention to sequential programs, CAP thus allows to specify and reason about modules that are implemented using sharing. The CAP approach, seen from our perspective, is to fix one particular separation algebra for all modules, which is sufficiently powerful to handle most cases of sharing. The algebra is specialized to each module by giving a per-module protocol definition, with access to the various stages in the protocol controlled by permission accounting. These explicit protocols, describing what atomic modifications may be performed on shared memory regions, give verification tasks a completely different flavour and intuition compared to fictional separation logic. In a sequential setting, the two systems are therefore very different solutions to the same problem. Concurrent abstract predicates is fundamentally rooted in a concurrent setting, though, which complicates the proof system. In particular, program verification requires showing *stability* of all intermediate pre- and postconditions in a proof.

Future work includes extending fictional separation logic to richer programming languages. Our preliminary investigations suggest that it is straightforward to extend the logic to a language with function pointers, by using the idea of nested triples [21] to specify such pointers. In fictional separation logic we will, of course, use *indirect* nested triples. To make it possible to call a function f with a function argument that uses an interpretation stacked on top of f's own interpretation, one can specify both f and its argument through a stacking of interpretations.

We are also interested in extending fictional separation logic to a concurrent language in order to find out whether it can retain its simplicity.

Acknowledgements. We would like to thank Jesper Bengtson, Thomas Dinsdale-Young, Filip Sieczkowski, Kasper Svendsen, Peter Sestoft and Jacob Thamsborg for helpful feedback and discussions.

References

1. Bengtson, J., Jensen, J.B., Sieczkowski, F., Birkedal, L.: Verifying object-oriented programs with higher-order separation logic in Coq. In: Proceedings of ITP (2011)
2. Biering, B., Birkedal, L., Torp-Smith, N.: BI-hyperdoctrines, higher-order separation logic, and abstraction. ACM Transactions on Programming Languages and Systems 29(5) (2007)

3. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules for algol-like languages. Logical Methods in Computer Science 2(5:1) (Aug 2006)
4. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: Proceedings of POPL. pp. 259–270 (2005)
5. Boyland, J.: Checking interference with fractional permissions. In: Proceedings of SAS (2003)
6. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Proceedings of LICS (2007)
7. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M., Vafeiadis, V.: Concurrent abstract predicates. In: Proceedings of ECOOP (2010)
8. Dinsdale-Young, T., Gardner, P., Wheelhouse, M.: Abstraction and refinement for local reasoning. In: Proceedings of VSTTE (2010)
9. Dinsdale-Young, T., Gardner, P., Wheelhouse, M.: Abstraction and refinement for local reasoning (February 2011), journal submission
10. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: Proceedings of APLAS (2009)
11. Dodds, M., Jagannathan, S., Parkinson, M.J.: Modular reasoning for deterministic parallelism. In: Proceedings of POPL (2011)
12. Gotsman, A., Berdine, J., Cook, B.: Precision and the conjunction rule in concurrent separation logic. In: Proceedings of MFPS (2011)
13. Haack, C., Hurlin, C.: Resource usage protocols for iterators. In: Proceedings of IWACO (2008)
14. Jensen, J.B., Birkedal, L.: Fictional separation logic: Appendix (2011), http://itu.dk/ jobr/research/fsl-appendix.pdf
15. Jensen, J.B., Birkedal, L., Sestoft, P.: Modular verification of linked lists with views via separation logic. Journal of Object Technology 10, 2:1–20 (2011)
16. Krishnaswami, N.R.: Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic. Ph.D. thesis, Carnegie Mellon University (2011)
17. Meyers, S.: More Effective C++: 35 New Ways to Improve Your Programs and Designs. Addison-Wesley Professional, first edn. (1996)
18. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: Proceedings of POPL. pp. 247–258 (2005)
19. Pilkiewicz, A., Pottier, F.: The essence of monotonic state. In: Proceedings of TLDI (2011)
20. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of LICS. pp. 55–74 (2002)
21. Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested Hoare triples and frame rules for higher-order store. In: Proceedings of CSL (2009)