Fractional Resources in Unbounded Separation Logic

THIBAULT DARDINIER, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland
ALEXANDER J. SUMMERS, University of British Columbia, Canada

Many separation logics support fractional permissions to distinguish between read and write access to a heap location, for instance, to allow concurrent reads while enforcing exclusive writes. Fractional permissions extend to composite assertions such as (co)inductive predicates and magic wands by allowing those to be multiplied by a fraction. Typical separation logic proofs require that this multiplication has three key properties: it needs to distribute over assertions, it should permit fractions to be factored out from assertions, and two fractions of the same assertion should be combinable into one larger fraction.

Existing formal semantics incorporating fractional assertions into a separation logic define multiplication *semantically* (via models), resulting in a semantics in which distributivity and combinability do not hold for key resource assertions such as magic wands, and fractions cannot be factored out from a separating conjunction. By contrast, existing automatic separation logic verifiers define multiplication *syntactically*, resulting in a different semantics for which it is unknown whether distributivity and combinability hold for all assertions.

In this paper, we present a novel semantics for separation logic assertions that allows states to hold more than a full permission to a heap location during the evaluation of an assertion. By reimposing upper bounds on the permissions held per location at statement boundaries, we retain key properties of separation logic, in particular, the frame rule. Our assertion semantics unifies semantic and syntactic multiplication and thereby reconciles the discrepancy between separation logic theory and tools and enjoys distributivity, factorisability, and combinability. We have formalised our semantics and proved its properties in Isabelle/HOL.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**; **Automated reasoning**; *Concurrency*.

Additional Key Words and Phrases: Fractional permissions, combinability, (co)inductive predicates, magic wands, automatic deductive verifiers

ACM Reference Format:
Thibault Dardinier, Peter Müller, and Alexander J. Summers. 2022. Fractional Resources in Unbounded Separation Logic. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 163 (October 2022), 27 pages. https://doi.org/10.1145/3563326

1 INTRODUCTION

Separation logic [Reynolds 2002] (SL thereafter) is an extension of Hoare logic that enables reasoning about (concurrent) heap-manipulating programs. SL permits reasoning about non-duplicable resources, for example the exclusive ownership of a part of the heap, with *resource assertions*. One simple and important resource assertion is the *points-to* assertion: The resource assertion $l \mapsto v$ ("location l points to v") holds in a state σ iff σ *owns* the heap location l and l contains the value v.

Authors' addresses: Thibault Dardinier, thibault.dardinier@inf.ethz.ch, Department of Computer Science, ETH Zurich, Switzerland; Peter Müller, peter.mueller@inf.ethz.ch, Department of Computer Science, ETH Zurich, Switzerland; Alexander J. Summers, alex.summers@ubc.ca, Department of Computer Science, University of British Columbia, Canada.

2475-1421/2022/10-ART163
https://doi.org/10.1145/3563326

SL permits splitting the resources held by a state, with the *separating conjunction* connective $*$ (also called the *star*): If A and B are SL assertions, the assertion $A * B$ holds in a state σ iff the resources held in σ can be split into two states σ_A and σ_B, written $\sigma = \sigma_A \oplus \sigma_B$, such that A holds in σ_A and B holds in σ_B. Intuitively, $\sigma_A \oplus \sigma_B$ represents the disjoint union of the resources of both states. As an example, the assertion $l_1 \mapsto v_1 * l_2 \mapsto v_2$ describes a state that (separately) owns the two heap locations l_1 (with value v_1) and l_2 (with value v_2).

In all existing variants of SL, states are *bounded*: They cannot own a location l more than once. Consequently, a state σ can be split into $\sigma_A \oplus \sigma_B$ only if σ_A and σ_B own disjoint parts of the heap. Thus, the assertion $l_1 \mapsto v_1 * l_2 \mapsto v_2$ implies that l_1 and l_2 are not aliases. More generally, the assertion $A * B$ implies that A and B describe disjoint parts of the heap. Thanks to the boundedness of states, SL supports the two following famous and important rules:

$$\frac{\{P\}\ C\ \{Q\} \quad mod(C) \cap fv(R) = \varnothing}{\{P * R\}\ C\ \{Q * R\}}\ (\mathit{Frame}) \qquad \frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\ (\mathit{Parallel})$$

The *Frame* rule enables reasoning locally about a program statement C. If C executes safely in a state that satisfies P and results in a state that satisfies Q, then it will also execute safely in a state that satisfies $P * R$, and it will result in a state that satisfies $Q * R$ (provided that R does not mention variables modified by C). This rule is crucial to prove that properties of the uninvolved parts of the heap (described by R) are not affected by executing C; they can be *framed around* C. Similarly, the *Parallel* rule enables reasoning locally about each parallel thread of a parallel composition, given that the two threads operate on disjoint parts of the heap.

To reason about concurrent sharing and the absence of race conditions, SL has been extended with *fractional permissions* [Bornat et al. 2005; Boyland 2003]. In this setting, a state can own a fraction p of a heap location l, written $l \overset{p}{\mapsto} v$, where p is a positive rational number. Fractional ownership ($p < 1$) grants read access to the location l, while exclusive ownership ($p = 1$) grants read and write access. States are also bounded in this setting, in the sense that they cannot own more than a fraction 1 of a heap location l. Two states σ_A and σ_B can be combined iff their fractional ownerships of each heap location l sum to at most 1 and they agree on the values of the heap locations owned by both. Combined with the *Parallel* rule, fractional permissions are particularly suitable for reasoning about concurrent threads that read the same heap locations. Consider an example with two concurrent threads. Exclusive ownership of l can be split into half ownership for each thread, which enables both threads to read l, and exclusive ownership of l (and thus write access) can be regained after the two threads have finished executing.

1.1 Fractional Resources

SL supports resource assertions more general than the points-to assertion, to enable reasoning about arbitrarily large data structures and at a higher level of abstraction. Ownership of arbitrarily large data structures, such as binary trees or linked lists, can for example be described with inductively-defined *predicates* [Parkinson and Bierman 2005]. Moreover, partial data structures can be expressed with the *separating implication* connective $-\!\!*$ (also called *magic wand* or *wand*): The assertion $A -\!\!* B$ describes resources which, combined with any state in which A holds, results in a state in which B holds. It can intuitively typically be seen as expressing the difference in resources between B and A: If B specifies an entire data structure, and A specifies a part of this data structure, then the wand $A -\!\!* B$ can express ownership of B where A has been removed. Specifying partial data structures with wands has proved useful, for example to track the ongoing iteration over a data structure [Maeda et al. 2011; Tuerk 2010] (where the left-hand side of the wand represents the part of the data structure that remains to be traversed), or to formally reason about borrowing references in the Rust programming language [Astrauskas et al. 2019]. Magic wands have also been used to abstractly specify protocols on client calls to an API [Haack and Hurlin 2009; Jensen et al. 2011; Krishnaswami 2006], such as the protocol that governs Java iterators.

Given the importance of these general resource assertions (*resources* hereafter), it is not surprising that the concept of fractional ownership has been generalised to resources (and, in turn, to general assertions): If A is an arbitrary SL assertion and π is a fraction, then A^{π} is a *fractional assertion* that represents a fraction π of A. Fractional assertions have both been studied in theory and applied in automatic SL verifiers. In theory [Brotherston et al. 2020; Le and Hobor 2018], A^{π} holds in a state σ iff there exists a state σ_A such that A holds in σ_A and σ corresponds to σ_A where all permission amounts have been multiplied by π, which we write $\sigma = \pi \odot \sigma_A$. We refer to this definition as the *semantic multiplication*. As an example, if $tree(x)$ represents exclusive ownership of all nodes of a binary tree rooted in x, then $tree(x)^{0.5}$ represents half ownership of all nodes of this binary tree.

Fractional resources are also supported by several automatic SL verifiers, including Chalice [Leino et al. 2009], VerCors [Blom and Huisman 2014], VeriFast [Jacobs et al. 2011], and Viper [Müller et al. 2016]. This support relies on the concept of *syntactic multiplication*: A fraction π of $A * B$ is interpreted as a fraction π of A combined with a fraction π of B, i.e. $(A * B)^{\pi}$ is interpreted as $A^{\pi} * B^{\pi}$. Using this distributivity property, the multiplying fraction can be pushed inside the assertion until it applies to points-to assertions, where $(l \overset{\alpha}{\mapsto} v)^{\pi}$ is interpreted as $l \overset{\pi \cdot \alpha}{\mapsto} v$.

While the semantic and syntactic multiplications look similar, it turns out that they give two distinct meanings to fractional resources! Indeed, while the semantic entailment $(A * B)^{\pi} \models A^{\pi} * B^{\pi}$ holds with both types of multiplication, the dual entailment $A^{\pi} * B^{\pi} \models (A * B)^{\pi}$, which is direct for the syntactic multiplication, does not hold with the semantic multiplication. The reason is that $(A * B)^{\pi}$, interpreted with semantic multiplication, might require stronger non-aliasing guarantees than the ones provided by $A^{\pi} * B^{\pi}$, as shown by the following example:

Example 1. $(\mathtt{x.f} \mapsto v)^{\frac{1}{2}} * (\mathtt{y.f} \mapsto v)^{\frac{1}{2}}$ *does not entail* $(\mathtt{x.f} \mapsto v * \mathtt{y.f} \mapsto v)^{\frac{1}{2}}$ *if interpreted with the semantic multiplication. Indeed,* $(\mathtt{x.f} \mapsto v)^{\frac{1}{2}} * (\mathtt{y.f} \mapsto v)^{\frac{1}{2}}$ *holds in a state* σ *with exclusive ownership of* `x.f` *(with value* v*) and in which* x *and* y *are aliases. However,* $(\mathtt{x.f} \mapsto v * \mathtt{y.f} \mapsto v)^{\frac{1}{2}}$ *does not hold in* σ*, otherwise it would imply (by definition of the semantic multiplication) the existence of a state that exclusively owns* `x.f` *twice, which is not possible since states are bounded.*

Current support for fractional resources in automatic verifiers, based on syntactic multiplication, has never been fully formalised. Worse still, as we show in this paper, the support in these tools is *not* aligned with the formal models considered in theoretical papers on the same topic. Therefore, it is unclear whether the rules they apply, e.g. to recombine two fractions of a recursively-defined predicate (as we explain next), are sound. In this paper we show how to give a fully formal model subsuming the cases supported in practical tools, and going beyond such support to formalise what it can mean to split and recombine more-general resources, such as magic wands.

1.2 Distributivity, Factorisability, and Combinability

As prior work highlights [Brotherston et al. 2020; Le and Hobor 2018], three key properties are needed when reasoning with fractional assertions, which we term *distributivity*, *factorisability*, and *combinability*[1]. We will illustrate shortly on an example why these three properties are necessary. The *distributivity* property holds for a SL connective iff multiplication by any fraction can be

[1]Prior work used a different terminology and referred to both the distributivity and factorisability properties as "distributivity" or the "distribution principle".

```
method processTree(x: Ref) {
  $\{tree(x)^{\pi}\}$
  if (x != null) {
    $\{tree(x)^{\pi} * x \neq null\}$
    $\{(tree(x)^{\frac{\pi}{2}} * x \neq null) * (tree(x)^{\frac{\pi}{2}} * x \neq null)\}$
        | $\{tree(x)^{\frac{\pi}{2}} * x \neq null\}$
        | $\{\exists x_l, x_r.\, x.d \overset{\frac{\pi}{2}}{\mapsto} \_ * x.l \overset{\frac{\pi}{2}}{\mapsto} x_l * x.r \overset{\frac{\pi}{2}}{\mapsto} x_r * tree(x_l)^{\frac{\pi}{2}} * tree(x_r)^{\frac{\pi}{2}}\}$
        | print(x.d)
        | processTree(x.l)
        | processTree(x.r)
        | $\{\exists x_l, x_r.\, x.d \overset{\frac{\pi}{2}}{\mapsto} \_ * x.l \overset{\frac{\pi}{2}}{\mapsto} x_l * x.r \overset{\frac{\pi}{2}}{\mapsto} x_r * tree(x_l)^{\frac{\pi}{2}} * tree(x_r)^{\frac{\pi}{2}}\}$
        | $\{tree(x)^{\frac{\pi}{2}}\}$
      ||
        | $\{tree(x)^{\frac{\pi}{2}} * x \neq null\}$
        | $\{\exists x_l, x_r.\, x.d \overset{\frac{\pi}{2}}{\mapsto} \_ * \ldots\}$
        | print(x.d)
        | processTree(x.l)
        | processTree(x.r)
        | $\{\exists x_l, x_r.\, x.d \overset{\frac{\pi}{2}}{\mapsto} \_ * \ldots\}$
        | $\{tree(x)^{\frac{\pi}{2}}\}$
    $\{tree(x)^{\frac{\pi}{2}} * tree(x)^{\frac{\pi}{2}}\}$
  }
  $\{tree(x)^{\pi}\}$
}
```

Fig. 1. A simple concurrent program that shows why distributivity, factorisability, and combinability are needed when reasoning with fractional resources. The SL predicate $tree(x)$ is recursively-defined as $x \neq null \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * tree(x_l) * tree(x_r)$. A proof outline is shown in blue. In SL with semantic multiplication, factorisability does not hold for separating conjunctions and, thus, the entailments at the end of each parallel branch are not valid! With syntactic multiplication, distributivity holds for the separating conjunction by definition, but it has not been shown that combinability holds: it is unclear whether the proof outline is correct. It *is* correct in the semantics we present in this paper.

distributed over it.[2] Distributivity holds e.g. for the separating conjunction both with semantic multiplication ($(A * B)^{\alpha}$ entails $A^{\alpha} * B^{\alpha}$) and syntactic multiplication (by definition). However, as we show in Sect. 2, distributivity does *not* hold for the magic wand with semantic multiplication. *Factorisability* is the dual property: it holds for a SL connective iff it is always possible to factor a common fraction out over it. As explained above, factorisability does not hold for the separating conjunction with semantic multiplication, i.e. $A^{\alpha} * B^{\alpha}$ does not always entail $(A * B)^{\alpha}$. However, factorisability holds for the separating conjunction with syntactic multiplication by definition. Finally, the *combinability* property holds for an assertion A iff two fractions of this assertion can always be combined, i.e. $A^{\alpha} * A^{\beta}$ entails $A^{\alpha+\beta}$. In this case, we say that the resource assertion A is *combinable*. As we show in Sect. 3, not all assertions are combinable. In particular, even if A and B are combinable, the magic wand $A \mathrel{-\!\!*} B$ is in general not combinable using semantic multiplication.

To illustrate why these three properties matter when reasoning with fractional resources, consider the simple concurrent program in Fig. 1, taken from Le and Hobor [2018]. This program manipulates the inductively-defined predicate $tree(x) = (x \neq \texttt{null} \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * tree(x_l) * tree(x_r))$, which expresses ownership of a binary tree stored on the heap: Either x is `null` (which corresponds to an empty tree), or we have ownership of its fields `x.d` (data of the node), `x.l` (pointer to x's left subtree), and `x.r` (pointer to x's right subtree), and we own the trees rooted in `x.l` and `x.r`. The precondition and postcondition of the method `processTree` is `tree(x)`$^{\pi}$ (where π is a ghost parameter omitted from `processTree`'s signature for brevity), which expresses that `processTree` only needs a read access to the tree rooted in x, and guarantees the

[2]Note that, in this paper, *distributivity* refers to this entailment only, and not to the equivalence, while we refer to the dual entailment as *factorisability*.

absence of data races. If x is not `null`, `processTree` forks two threads, and both threads print the data of the node (`x.d`), before recursively calling `processTree` on the left and right subtrees of `x`.

We show in blue a proof outline for this program, which relies on the aforementioned three key properties: distributivity, factorisability, and combinability. This proof outline, explained next, is valid with syntactic multiplication, but *invalid* with semantic multiplication. If `x` is not `null`, we split $\mathit{tree}(x)^{\pi}$ into $\mathit{tree}(x)^{\frac{\pi}{2}} * \mathit{tree}(x)^{\frac{\pi}{2}}$ to give reading permission to each thread, using the *Parallel* rule. Inside each thread we unfold the definition of $\mathit{tree}(x)$ and use the **distributivity** property to distribute the fraction π over the separating conjunction, which, conjoined with the knowledge that $x \neq \texttt{null}$, yields $\exists x_l, x_r.\, x.d \overset{\frac{\pi}{2}}{\mapsto} \_ * x.l \overset{\frac{\pi}{2}}{\mapsto} x_l * x.r \overset{\frac{\pi}{2}}{\mapsto} x_r * \mathit{tree}(x_l)^{\frac{\pi}{2}} * \mathit{tree}(x_r)^{\frac{\pi}{2}}$. This is enough to justify read access to `x.d` and to recursively call `processTree` on both subtrees (with the ghost parameter $\frac{\pi}{2}$). After the two recursive calls, we use the **factorisability** property to recompose the $\frac{\pi}{2}$ ownership of $\mathit{tree}(x)$ from the $\frac{\pi}{2}$ ownership of its fields and subtrees. Note that as explained above, this step is invalid with the semantic multiplication! Finally, after the two threads have finished executing, we use the **combinability** property to recombine the two $\frac{\pi}{2}$ fractions of $\mathit{tree}(x)$ into $\mathit{tree}(x)^{\pi}$. Note that to justify this final step, we need to know that $\mathit{tree}(x)$ is combinable. This proof outline illustrates a typical pattern: Distributivity is necessary when we unfold a fractional resource, while factorisability is necessary to fold back the fractional resource, and combinability is necessary to recombine fractions of a resource that was shared between threads.

This example demonstrates the importance of distributivity, factorisability, and combinability, yet traditional separation logics do not fully support them. In SL semantics based on *semantic* multiplication, distributivity does not hold for magic wands, factorisability does not hold for separating conjunctions, and combinability does not hold for magic wands in general (as we show later). Hence, the entailments at the end of the parallel branches in our example are actually *not valid*, as was already pointed out by Le and Hobor [2018]. By contrast, tools that implement *syntactic* multiplication happily verify the program, but it has never been shown whether combinability actually holds in this setting and, hence, whether the last entailment is valid.

1.3 State of the Art

Several approaches have been proposed to deal with the limitations of the semantic multiplication.

Factorisability for the separating conjunction. According to Le and Hobor [2018], the issue is that assertions such as $\texttt{x.f} \overset{p}{\mapsto} \_ * \texttt{y.f} \overset{p}{\mapsto} \_$, where p is a fractional permission, do not necessarily imply that `x` and `y` are not aliased (for example when $p = 0.5$). They thus use a more complex permission model, the *binary tree share* model [Dockins et al. 2009], which satisfies this *disjointness* property, and define a multiplication over binary tree shares. Going back to Ex. 1, if we replace $\frac{1}{2}$ by any binary tree share τ, then we can prove that $(\texttt{x.f} \mapsto v)^{\tau} * (\texttt{y.f} \mapsto v)^{\tau}$ entails $(\texttt{x.f} \mapsto v * \texttt{y.f} \mapsto v)^{\tau}$. More generally, using the disjointness property, they prove that if A and B are τ*-uniform* for some binary tree share τ (meaning that any state that satisfies A or B must have either no permission or exactly τ permission to each and every heap location), then the factorisability entailment $A^{\pi} * B^{\pi} \models (A * B)^{\pi}$ holds. For example, $\texttt{x.f} \overset{\tau}{\mapsto} v * \texttt{y.f} \overset{\tau'}{\mapsto} v$ is τ-uniform if $\tau' = \tau$ and not otherwise. As well as restricting to τ-uniform assertions, their approach is limited by the complex permission model needed: the notion of multiplication is neither commutative nor left-distributive, and it does not have inverses, which for example prevents factorisability from holding for implication assertions.

Brotherston et al. [2020] retain fractional permissions, but add new variants of the two main SL connectives. Their assertions include the usual (*weak*) star $*$, the usual (*weak*) wand $-\!\!*$ (adjunct of the weak star), a *strong* star $\circledast$, and (its adjunct) a *strong* wand $-\!\!\circledast$[3]. While the weak star $*$ behaves

[3]Note: the notation here is *opposite* to theirs: They denote the strong star $*$ and the weak star $\circledast$, and analogously for wands.

as usual, the strong star $\circledast$ requires strict non-aliasing, e.g. $\mathtt{x.f} \overset{0.5}{\mapsto} v \circledast \mathtt{x.f} \overset{0.5}{\mapsto} v$ is unsatisfiable. They then prove valid the factorisability entailment $A^\pi \circledast B^\pi \models (A \circledast B)^\pi$. To solve the issue in Fig. 1, they thus redefine the $\mathit{tree}(x)$ predicate with the strong star. They also prove a *strong frame rule* for the strong star, which is quite limited, since it can be applied only with a program statement C that does not receive resources. Moreover, while their strong star satisfies factorisability, their strong wand does not satisfy distributivity.

Combinability. Le and Hobor [2018] prove that combinability holds for *precise* assertions [O'Hearn et al. 2004]. An assertion A is *precise* iff, for any state σ, A holds in *at most one* state σ' smaller than σ. They provide formal rules for proving assertions precise, as well as an induction principle to prove that an inductively-defined predicate is precise, based on a well-founded order of heaps decreasing by at least a constant positive permission amount. As an example, to prove that $\mathit{tree}(x)$ is precise (and, thus, combinable), they can assume that $\mathit{tree}(x_l)$ and $\mathit{tree}(x_r)$ are precise, as long as they can prove that $\mathit{tree}(x_l)$ and $\mathit{tree}(x_r)$ represent heaps smaller than $\mathit{tree}(x)$ by at least a constant positive permission amount, e.g. 1. However, their approach does not capture assertions that are combinable, but not precise, which are common in practice.

Brotherston et al. [2020] add *nominal labels* to their assertion language, to track that two fractional assertions have the same *origin*, and thus can be recombined. At any time in a proof, one can conjoin the current assertion A with a fresh label l. Using this label, one can later in the program use the following entailment to recombine two fractions of A: $(l \wedge A)^\alpha * (l \wedge A)^\beta \models (l \wedge A)^{\alpha+\beta}$. They prove the specification $\{l \wedge \mathit{tree}(x)^\pi\}$ `processTree(x)` $\{l \wedge \mathit{tree}(x)^\pi\}$ for some label l for the example in Fig. 1. To prove such preconditions, they also introduce a *jump modality* @ in their assertion language: intuitively, $@_l A$ means that A holds in the heap labelled by l. While this solves the combinability problem for fractions of an assertion that provably have the same origin, it incurs a significant cost in terms of annotation: their proof outline for the simple method `processTree` (Fig. 1) requires managing 10 different labels.

1.4 Approach and Contributions

In this paper, we present a novel assertion semantics for a separation logic that unifies the two kinds of multiplications: syntactic and semantic multiplication are equivalent in our logic, reconciling the discrepancy between SL theory and automatic SL tools. Our logic solves the technical problems explained above: Distributivity holds for the magic wand, factorisability holds for the separating conjunction, and the wand $A \mathrel{-\!\!*} B$ preserves combinability (is combinable if B is combinable).

The key idea of our logic is to allow *unbounded* states (states that can have *more than a full permission* to a heap location) in the underlying *assertion* semantics. Bounds on the held permissions are re-introduced in Hoare triples at statement boundaries, which is sufficient to retain SL's powerful reasoning principles, such as the frame rule. In the following, we will refer to our logic as *unbounded separation logic* (*unbounded logic* for short) and to standard SL as the *bounded* logic.

We make the following contributions:

- We present and formalise a novel separation logic unifying semantic and syntactic multiplication. We prove that it guarantees distributivity and factorisability for all commonly-used SL assertions including the star and the magic wand. We show that reimposing boundedness in Hoare triples is sufficient to justify the frame rule (Sect. 2). Our logic provides the first formal justification for fractional assertions as implemented in automatic SL verifiers.
- We show that the existing approach of characterising combinability indirectly via preciseness is imprecise in general, and prove that commonly-used SL connectives are combinable by

defining and reasoning about the property directly. In particular, we prove that, unlike in the bounded logic, the magic wand is combinable[4] in the unbounded logic (Sect. 3).

- We provide a powerful and novel induction principle for reasoning about (co-)inductively-defined predicates in our logic. In particular, this induction principle allows simple justifications that a particular (co-)inductively-defined predicate is combinable (Sect. 4).
- We show how our unbounded logic can serve as a formal foundation to (1) justify and (2) extend the support of fractional resources in automatic SL verifiers (such as Chalice, VerCors, VeriFast, and Viper). Using the equivalence of syntactic and semantic multiplication, we show how to support fractional magic wands, whose support does not exist in any tool, to our knowledge. Moreover, we identify a syntactic criterion on a (potentially recursive) predicate's definition sufficient to ensure that this predicate is combinable (Sect. 5).

After presenting these technical contributions, we illustrate the advantages of the unbounded logic on two examples of heap-manipulating concurrent programs, one of them from the literature (Sect. 6). We discuss related work in Sect. 7, and conclude in Sect. 8.

All technical results presented in this paper have been formalised and proven in Isabelle/HOL [Nipkow et al. 2002], and our formalisation is publicly available [Dardinier 2022; Dardinier et al. 2022a].

2 UNBOUNDED SEPARATION LOGIC

In this section, we present and formally define an *unbounded* version of SL. The key idea, which we explain in Sect. 2.1 and formalise in Sect. 2.2 and Sect. 2.3, is to allow unbounded states (states that can own a heap location more than once) in the assertion semantics. We show in Sect. 2.4 the distributivity and factorisability rules for our unbounded logic. In particular, distributivity holds for the magic wand and factorisability holds for the separating conjunction, which is not the case for traditional, bounded separation logic. Finally, we show in Sect. 2.5 that reimposing boundedness in Hoare triples is sufficient to preserve key technical results of SL, such as the frame rule.

2.1 Key Idea: 1+1 = 2

As explained in Sect. 1, one key limitation of reasoning with fractional resources in (bounded) SL is that factorising over the star is in general unsound, i.e. the entailment $A^\pi * B^\pi \models (A * B)^\pi$ generally does not hold. As shown with Ex. 1 ($(\texttt{x.f} \overset{1}{\mapsto} v)^{\frac{1}{2}} * (\texttt{y.f} \overset{1}{\mapsto} v)^{\frac{1}{2}} \not\models (\texttt{x.f} \overset{1}{\mapsto} v * \texttt{y.f} \overset{1}{\mapsto} v)^{\frac{1}{2}}$), this entailment fails because of potential aliasing between x and y. The key reason is that states are bounded, and thus the addition of two fractional permissions is a *partial* operation: If a state σ_1 (resp. σ_2) has p_1 (resp. p_2) ownership of the location l, then σ_1 and σ_2 can be combined only if $p_1 + p_2 \leq 1$. In this sense, $1 + 1$ (for example) is undefined. On the left-hand side of this simple example, we add half of a full (1) permission of `x.f` to half of a full permission of `y.f`. If x and y are aliases, this corresponds to a permission amount of $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1 = 1$. Since $\frac{1}{2} + \frac{1}{2} \leq 1$, the addition is defined, and thus the left-hand side is satisfiable. On the other hand, the right-hand side is unsatisfiable when x and y are aliases, because the addition is performed *before* the multiplication. In other words, to satisfy the right-hand side when x and y are the same, a state needs $\frac{1}{2} \cdot (1 + 1)$ permission to `x.f`. But $\frac{1}{2} \cdot (1 + 1)$ is undefined, as $1 + 1$ is undefined as a permission amount.

As explained in Sect. 1.3, Brotherston et al. [2020] solve this issue by *strengthening* the left-hand side with the strong star $\circledast$. In other words, they replace $(\texttt{x.f} \overset{1}{\mapsto} v)^{\frac{1}{2}} * (\texttt{y.f} \overset{1}{\mapsto} v)^{\frac{1}{2}}$ with $(\texttt{x.f} \overset{1}{\mapsto} v)^{\frac{1}{2}} \circledast (\texttt{y.f} \overset{1}{\mapsto} v)^{\frac{1}{2}}$. This new left-hand side is stronger because, by definition of $\circledast$, it enforces that x and y are non-aliases, and thus implies the right-hand side $(\texttt{x.f} \overset{1}{\mapsto} v \circledast \texttt{y.f} \overset{1}{\mapsto} v)^{\frac{1}{2}}$.

[4]If its right-hand side is also combinable.

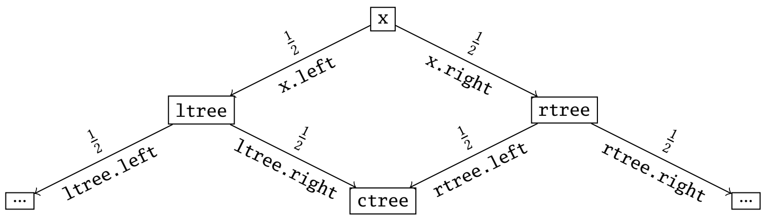

Fig. 2. In any assertion semantics that enjoys factorisability, $tree(x)^{\frac{1}{2}}$ may represent a directed acyclic graph instead of a tree because the upper bound on the permissions held does not prevent sharing (here, of `ctree` via `x.left.right`) and `x.right.left`). In unbounded states, this loss of non-aliasing information occurs even for the full $tree(x)$.

In our new unbounded logic, we go the other way, and make the entailment valid by *weakening* the right-hand side. Concretely, we allow 1 + 1 to equal 2, which makes the right-hand side of Ex. 1 satisfiable, and the entailment valid. We achieve this by considering *unbounded* states, i.e. states that can have more than a full permission to a heap location. Going back to the example and proof outline from Fig. 1, the entailment $\exists x_l, x_r.\, x.d \overset{\frac{\pi}{2}}{\mapsto} \_ * x.l \overset{\frac{\pi}{2}}{\mapsto} x_l * x.r \overset{\frac{\pi}{2}}{\mapsto} x_r * tree(x_l)^{\frac{\pi}{2}} * tree(x_r)^{\frac{\pi}{2}} \models (\exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * tree(x_l) * tree(x_r))^{\frac{\pi}{2}}$ used in the proof is now valid!

Loss of non-aliasing information. Considering unbounded states in the assertion semantics solves the issue of factorisability for the star, but this comes at a cost: As observed by Bornat et al. [2005], any assertion semantics that enjoys factorisability weakens the meaning of $tree(x)^{\pi}$ because $tree(x)^{\frac{1}{2}}$ no longer describes only binary trees, but also admits DAGs (directed acyclic graphs). Fig. 2 shows an illustration of a state σ_d in which $tree(x)^{\frac{1}{2}}$ holds, even though the node `ctree` can be reached from x via two distinct paths, `x.left.right` and `x.right.left`.

This loss of non-aliasing information caused by factorisability occurs in traditional bounded states (like the one depicted in Fig. 2) if the sum of the fractional permissions for each heap location does not exceed a full permission. Since unbounded states do not impose an upper bound on permissions, non-aliasing information is lost even for larger fractions: Even the full $tree(x)$ admits DAGs; e.g. consider a variation of Fig. 2, where each fractional permission is multiplied by 2.

However, a crucial insight of our work is that this loss of non-aliasing information is not an issue in practice. As we will discuss shortly, we re-impose boundedness in a Hoare logic at *statement* boundaries. That is, even in the unbounded logic, $tree(x)$ denotes a tree before and after each statement. This is sufficient to retain the frame rule (as we show in Sect. 2.5), which is by far the most important proof step that relies on non-aliasing information. As an example, if we split the tree into its left and right subtrees and call a method on the right subtree, we still know that the left subtree will remain unchanged. When the call returns and the subtrees are re-combined, execution is at a statement boundary, and we regain all non-aliasing properties of a tree.

In the rare case that non-aliasing information is needed explicitly (e.g. to prove `ltree.right != rtree.left`), it can be obtained via suitable functional specifications. For instance, the *tree* predicate could be extended to take the set of nodes as an argument and to express that the nodes in the left and in the right subtree are disjoint. We note that many concurrent programs with shared data structures have been formalised and proven correct in the automatic SL verifiers Chalice, VerCors, VeriFast, and Viper, even though these verifiers also "suffer" from this loss of non-aliasing information because they use syntactic multiplication and, thus, have factorisability. This empirically supports the claim that explicit non-aliasing information is not crucial for proofs of operations that manipulate data structures to which fractional resources are held.

2.2 State Model and Multiplication

In order to capture different state models and different flavours of SL, our unbounded logic is parameterised by a *separation algebra* $(\Sigma, \oplus)$ [Calcagno et al. 2007; Dockins et al. 2009]:

Definition 1. *A* ***separation algebra*** *is a pair* $(\Sigma, \oplus)$ *where* Σ *is a set of states and* $\oplus$ *is a partial addition that is commutative and associative. In other words,* $(\Sigma, \oplus)$ *is a* partial commutative monoid.

For two states $\sigma, \sigma' \in \Sigma$*, we write* $\sigma \# \sigma'$ *to express that* $\sigma \oplus \sigma'$ *is defined, and* $\sigma' \geq \sigma$ *to express that* σ' *is* greater than σ *in the* $\oplus$*-induced order (i.e. iff* $\exists r \in \Sigma.\ \sigma' = \sigma \oplus r$*).*

As an example, we can represent heaps with fractional permissions as partial functions from a set of heap locations L to a set of pairs of a value (from the set V) and a permission amount (from $\mathbb{Q}^+$), i.e. Σ can be the set of functions of type $L \rightharpoonup V \times \mathbb{Q}^+$. Crucially, note that the permission amounts are not upper-bounded. Two states σ and σ' are compatible, i.e. $\sigma \# \sigma'$, iff they agree on the values they both define, and their combination $\sigma \oplus \sigma'$ is the union of their values and the additions of the permission amounts for each heap location. Thus, a state σ' is greater than a state σ iff σ' contains the same value and has at least as much permission as σ for each heap location where σ is defined.

To express multiplications, our unbounded logic is also parameterised by a *semifield of scalars*:

Definition 2. *A* ***semifield of scalars*** *is a tuple* $(S, +, \cdot, 1)$*, which is a semifield with a multiplicative inverse and without a neutral element for the addition. More precisely, for all* $\alpha, \beta, \pi \in S$*, we require* $(S, +, \cdot, 1)$ *to satisfy the following axioms:*

$$\alpha \cdot 1 = \alpha \qquad \alpha \cdot \alpha^{-1} = 1 \qquad \alpha + \beta = \beta + \alpha$$
$$\alpha \cdot \beta = \beta \cdot \alpha \qquad (\alpha \cdot \beta) \cdot \pi = \alpha \cdot (\beta \cdot \pi) \qquad \pi \cdot (\alpha + \beta) = (\pi \cdot \alpha) + (\pi \cdot \beta)$$

Every scalar is required to have a multiplicative inverse, which is crucial to get some properties, e.g. to factorise a fraction out of an implication. As an example, the set of positive rational numbers $\mathbb{Q}^+$ and set of positive reals $\mathbb{R}^+$ are semifields of scalars. We also require that we can multiply states by scalars with a multiplication operation $\odot$:

Definition 3. *A* ***left*** S***-module*** Σ *is a tuple* $(\Sigma, \oplus, S, +, \cdot, 1, \odot)$ *where* $(\Sigma, \oplus)$ *is a separation algebra,* $(S, +, \cdot, 1)$ *is a semifield of scalars, and* $\odot$ *is a total multiplication from* $S \times \Sigma$ *to* Σ*. More precisely, for all* $\alpha, \beta \in S$ *and* $\sigma, \sigma' \in \Sigma$*, we require* $(\Sigma, \oplus, S, +, \cdot, 1, \odot)$ *to satisfy the the following axioms:*

$$1 \odot \sigma = \sigma \qquad \alpha \odot (\beta \odot \sigma) = (\alpha \cdot \beta) \odot \sigma$$
$$\alpha \odot (\sigma \oplus \sigma') = (\alpha \odot \sigma) \oplus (\alpha \odot \sigma') \qquad (\alpha + \beta) \odot \sigma = (\alpha \odot \sigma) \oplus (\beta \odot \sigma)$$

In our example, if σ is a partial function from L to $V \times \mathbb{Q}^+$, and π is an element of $\mathbb{Q}^+$, then $\pi \odot \sigma$ can be defined as multiplying location-wise the permission amounts of σ by π, and leaving the values unchanged.

Finally, we consider a predicate *valid* on Σ, where *valid*(σ) means that σ is a valid state in the bounded sense. *valid* must be (downward) monotonic, i.e. all states smaller than a valid states must also be valid. In our example, a state is valid iff it has at most 1 permission to each heap location.

2.3 Assertions

To capture different resource models with our unbounded logic, we consider, in our assertion language, semantic assertions (i.e. functions from Σ to Booleans) to abstract over SL assertions that do not contain connectives, such as the usual points-to assertion $\texttt{x.f} \overset{p}{\mapsto} v$. We consider the following assertion language, where A ranges over assertions, x ranges over variable names, and $\mathcal{B}$ ranges over *semantic* assertions:

$$A := \mathcal{B} \mid A * A \mid A \mathrel{-\!\!*} A \mid A^{\pi} \mid A^{*} \mid A \Rightarrow A \mid A \wedge A \mid A \vee A \mid \exists x.\, A \mid \forall x.\, A \mid \mathbf{P} \mid \lceil A \rceil$$

$$
\begin{aligned}
&\sigma, s, \Delta \models \mathcal{B} \iff \mathcal{B}(\sigma)\\
&\sigma, s, \Delta \models A * B \iff (\exists a, b.\, \sigma = a \oplus b \text{ and } a, s, \Delta \models A \text{ and } b, s, \Delta \models B)\\
&\sigma, s, \Delta \models A^{\pi} \iff (\exists a.\, a, s, \Delta \models A \text{ and } \sigma = \pi \odot a)\\
&\sigma, s, \Delta \models A^{*} \iff (\exists a, \pi.\, a, s, \Delta \models A \text{ and } \sigma = \pi \odot a)\\
&\sigma, s, \Delta \models A \mathrel{-\!\!*} B \iff (\forall a.\, (a, s, \Delta \models A \text{ and } \sigma \# a) \implies \sigma \oplus a, s, \Delta \models B)\\
&\sigma, s, \Delta \models A \Rightarrow B \iff (\sigma, s, \Delta \models A \implies \sigma, s, \Delta \models B)\\
&\sigma, s, \Delta \models A \wedge B \iff (\sigma, s, \Delta \models A \text{ and } \sigma, s, \Delta \models B)\\
&\sigma, s, \Delta \models A \vee B \iff (\sigma, s, \Delta \models A \text{ or } \sigma, s, \Delta \models B)\\
&\sigma, s, \Delta \models \exists x.\, A \iff (\exists v.\, \sigma, s(x := v), \Delta \models A)\\
&\sigma, s, \Delta \models \forall x.\, A \iff (\forall v.\, \sigma, s(x := v), \Delta \models A)\\
&\sigma, s, \Delta \models \mathbf{P} \iff \sigma \in \Delta(s)\\
&\sigma, s, \Delta \models \lceil A \rceil \iff (\mathit{valid}(\sigma) \implies \sigma, s, \Delta \models A)
\end{aligned}
$$

Fig. 3. Meaning of unbounded SL assertions. $\sigma \in \Sigma$ is an unbounded state, s is a store of local variables (mapping variable names to values), and Δ is an interpretation (mapping a store to a set of states from Σ).

Given an unbounded state $\sigma \in \Sigma$, a *store* s of local variables (a total function from a set of variable names to a set of values) and an *interpretation context* (which we explain below). The meaning of SL assertions is defined in Fig. 3. Most connectives are defined in the usual SL[5] ($*$, $-\!\!*$) or logical ($\wedge$, $\vee$, $\exists$, $\forall$, $\Rightarrow$) way. The *wildcard* assertion A^* represents an unknown (existentially-quantified) fraction of A (recall that scalars are required to have a multiplicative inverse and thus they cannot be zero). Wildcard assertions are ideal to represent read-only duplicable permissions [Leino et al. 2009]; as an example, $(\texttt{x.f} \mapsto v)^*$ represents *some non-zero* permission of `x.f` (which should contain the value v). Note that, to avoid orthogonal issues such as capture-avoidance and clashes between free and bound names, we use a total store, and thus allow the existential and universal quantifiers to "overwrite" values in the store. For example, the assertion $x = 5 \wedge (\exists x.\, x = 7)$ is *satisfiable*, because the existential quantifier "overwrites" the value of the variable x in the store.

For simplicity of our formalisation, we incorporate recursively-defined predicates (discussed in Sect. 4) via a *single* syntactic predicate symbol $\mathbf{P}$. This is not a mathematical limitation (we can encode multiple predicates in a single one with a dedicated argument to "select" the right predicate definition). The symbol $\mathbf{P}$ represents instances of our (only) predicate; the interpretation context Δ provides the meaning of this predicate: it defines the set of states which correspond to the predicate instance being held. Again for simplicity of our formalisation (avoiding a definition for capture-avoiding substitution), parameterisation of our predicate symbol is *implicit*: we treat the argument names in a predicate's definition as (reserved) variables in our usual store s, and parameterise Δ with such a store from which it can "read off" the values of (only) these parameters. We then encode an instance of a predicate such as $\mathbf{P}(e)$ via the assertion $\exists x.\, x = e \wedge \mathbf{P}$.

As an example (revisited in Sect. 4) assume that Δ_t represents the predicate *tree*, and the name of the argument of P is x. Then $\Delta_t(s)$ depends only on the value of $s(x)$: $\Delta_t(s)$ represents the set of states that own a tree rooted in $s(\mathrm{x})$. An instance e.g. $\mathit{tree}(x_l)$ is represented as $\exists x.\, x = x_l * \mathbf{P}$. We explain how the interpretation context Δ is constructed in Sect. 4.

[5]Note that the difference in the semantics of the bounded and unbounded logics comes from the state model and not from the assertion semantics itself.

$$\frac{}{(A^{\alpha})^{\beta} \equiv_{\Delta} A^{\alpha * \beta}}\ (\mathit{DotDot}) \qquad \frac{\mathit{pure}(A)}{A^{\pi} \equiv_{\Delta} A}\ (\mathit{DotPure}) \qquad \frac{}{A^{1} \equiv_{\Delta} A}\ (\mathit{DotFull}) \qquad \frac{}{(A * B)^{\pi} \equiv_{\Delta} A^{\pi} * B^{\pi}}\ (\mathit{DotStar})$$

$$\frac{}{(A \mathrel{-\!\!*} B)^{\pi} \equiv_{\Delta} A^{\pi} \mathrel{-\!\!*} B^{\pi}}\ (\mathit{DotWand}) \qquad \frac{}{(A \Rightarrow B)^{\pi} \equiv_{\Delta} A^{\pi} \Rightarrow B^{\pi}}\ (\mathit{DotImp}) \qquad \frac{}{A \models_{\Delta} B \Longleftrightarrow A^{\pi} \models_{\Delta} B^{\pi}}\ (\mathit{DotPos})$$

$$\frac{}{(\exists x.\, A)^{\pi} \equiv_{\Delta} \exists x.\, A^{\pi}}\ (\mathit{DotExists}) \frac{}{(\forall x.\, A)^{\pi} \equiv_{\Delta} \forall x.\, A^{\pi}}\ (\mathit{DotForall}) \qquad \frac{}{A^{\alpha+\beta} \models_{\Delta} A^{\alpha} * A^{\beta}}\ (\mathit{Split})$$

$$\frac{}{(A \wedge B)^{\pi} \equiv_{\Delta} A^{\pi} \wedge B^{\pi}}\ (\mathit{DotAnd}) \qquad \frac{}{(A \vee B)^{\pi} \equiv_{\Delta} A^{\pi} \vee B^{\pi}}\ (\mathit{DotOr}) \qquad \frac{}{(A^{*})^{\pi} \equiv_{\Delta} A^{*} \equiv_{\Delta} (A^{\pi})^{*}}\ (\mathit{DotWild})$$

Fig. 4. Distributivity and factorisation rules in the unbounded logic. An assertion A is pure, written *pure*(A), iff it does not depend on the heap and the interpretation context, i.e. $\forall \sigma, \sigma', s, \Delta, \Delta'.\ (\sigma, s, \Delta \models A \leftrightarrow \sigma', s, \Delta' \models A)$.

Finally, we include a *bounding* operator ($\lceil \_ \rceil$) in our language: The bounded assertion $\lceil A \rceil$ trivially holds in invalid states, and in all valid states that satisfy A. This is used to reimpose boundedness in Hoare triples (*cf.* Sect. 2.5), as well as to express the usual magic wand in our unbounded logic.

2.4 Distributivity and Factorisability

We can now prove that all SL connectives satisfy both distributivity and factorisability in our unbounded logic, in contrast to traditional bounded SL. We write $A \models_{\Delta} B$ to express that A semantically entails B for all possible stores and for the interpretation context Δ, i.e. $A, \models_{\Delta} B \Longleftrightarrow (\forall \sigma, s.\ \sigma, s, \Delta \models A \Rightarrow \sigma, s, \Delta \models B)$. We write $A \equiv_{\Delta} B$ iff $A \models_{\Delta} B$ and $B \models_{\Delta} A$.

We formalise the distributivity and factorisability properties for our assertion language via a set of rules (Fig. 4). All rules describe equivalences in our logic, except the *Split* rule. As explained in Sect. 1, the dual entailment, $A^{\alpha} * A^{\beta} \models_{\Delta} A^{\alpha+\beta}$, holds for *combinable* assertions only, as we discuss in Sect. 3. We proved the following theorem in Isabelle/HOL:

Theorem 1. ***Distributivity and factorisability in the unbounded logic.***
All rules shown in Fig. 4 hold in the unbounded logic.

The rules *DotImp*, and *DotPos* are notable, since they rely on the key property that the scalars we consider have a multiplicative inverse. In contrast, Le and Hobor [2018]'s tree-permissions cannot be inverted, and thus they obtain only one direction for these rules.

Comparison to bounded SL. The *DotStar* and *DotWand* rules do not hold in general in the bounded version of SL. As discussed in Sect. 1, *DotStar* does not hold because $A^{\pi} * B^{\pi} \models_{\Delta} (A * B)^{\pi}$ is not true in general. Similarly, *DotWand* does not hold, because $(A \mathrel{-\!\!*} B)^{\pi} \models_{\Delta} A^{\pi} \mathrel{-\!\!*} B^{\pi}$ is invalid in general. Next, we discuss magic wands in the bounded and unbounded logics in more detail.

A magic wand $A \mathrel{-\!\!*} B$ holds in a state σ iff B holds in all states of the form $\sigma \oplus \sigma_A$, where σ_A is a state compatible with σ and in which A holds. Therefore, one can satisfy a wand in two ways: (1) by including enough resources in σ such that combining these resources with the ones specified by A results in the resources required by B, or (2) by ensuring that *any* state σ_A in which A holds is incompatible with σ. The latter can be achieved in the bounded logic by including enough resources in σ such that they cannot be combined (in a bounded state) with those already specified by A.

Example 2. *The magic wand* $W_1 := \mathtt{x.f} \overset{0.5}{\mapsto} \_ \mathrel{-\!\!*} (\mathtt{x.f} \overset{0.5}{\mapsto} \_ * \mathtt{y.g} \mapsto \_)$ *holds in a state* σ *in the bounded logic if (1)* σ *holds full permission to* $\mathtt{y.g}$*, or (2)* σ *holds* more *than half (e.g. full) permission to* $\mathtt{x.f}$*. In the latter case,* σ *combined with any state satisfying the left-hand side of the wand results in a state that holds more than full permission to* $\mathtt{x.f}$*, which is inconsistent in the bounded logic.*

To see why distributivity does not hold for the magic wand in the bounded logic, consider the fractional wand ${W_1}^{0.5}$. According to the semantics of fractional assertions (line 3 in Fig. 3) and strategy (2) above, ${W_1}^{0.5}$ holds in a state σ that holds half permission to `x.f` (and no permission to `y.g`). However, this state does *not* satisfy $(\texttt{x.f} \overset{0.5}{\mapsto} \_)^{0.5} \mathrel{-\!\!*} (\texttt{x.f} \overset{0.5}{\mapsto} \_ * \texttt{y.g} \mapsto \_)^{0.5}$ because it is not necessarily inconsistent with states σ_A that satisfy the right-hand side $(\texttt{x.f} \overset{0.5}{\mapsto} \_)^{0.5}$ and because it does not hold the permission to `y.g` required by the right-hand side: distributivity does not hold.

In contrast, the unbounded logic offers only strategy (1) to satisfy a wand because states that have more than full permission are no longer necessarily inconsistent. This has three important consequences: First, distributivity (*DotWand*) holds for W_1 and for wands in general. Second, it makes wands combinable, as we show in Sect. 3. Third, unbounded states lead to a *stronger* meaning for wands compared to the bounded logic as they must be satisfied by following strategy (1).

The stronger meaning of wands in our unbounded logic is not restrictive for many practical purposes. For example the wands used to specify partial data structures during an ongoing traversal or to model borrowing in Rust need to hold according to strategy (1) since proofs use them to obtain the resources on their right-hand sides. Nonetheless, if the bounded version of a wand is really needed it can be expressed in our unbounded logic using our bounding operator: $A \mathrel{-\!\!*} B$ in the bounded logic corresponds to $A \mathrel{-\!\!*} \lceil B \rceil$ in our logic, since $\lceil B \rceil$ trivially holds in invalid states. Thus, proofs that require the bounded version of magic wands can still be expressed in our logic.

2.5 Frame Rule and Boundedness in Hoare Triples

Considering unbounded states in the assertion semantics might at first glance look surprising or even dangerous. After all, non-aliasing is a key component of separation logic, but lost with unbounded states: The SL entailment $\texttt{x.f} \mapsto \_ * \texttt{y.f} \mapsto \_ \models \texttt{x} \neq \texttt{y}$ does not hold in the unbounded logic. To retain non-aliasing reasoning and key technical results such as the frame rule, we reimpose boundedness on statement boundaries in Hoare triples, as we explain next.

Explicit non-aliasing and Hoare triples. To define the meaning of Hoare triples in our unbounded logic, we assume a big-step semantics given by the relation $\langle ., . \rangle \longrightarrow .$, where $\langle (\sigma, s), C \rangle \longrightarrow (\sigma', s')$ expresses that executing the program statement C in the state described by the heap σ and the store s might lead to the state described by the heap σ' and store s', and we assume a special value $\bot$ to express an error. We assume that this semantics operates only on valid states, i.e. $\langle (\sigma, s), C \rangle \longrightarrow (\sigma', s')$ is false if σ or σ' is invalid. Using this semantics, we define the meaning of Hoare triples to consider valid states only. Intuitively, this means that the unbounded logic permits unbounded states *during* the evaluation of an assertion, but the pre- and postconditions of Hoare triples always describe valid, that is, bounded states. More precisely, the Hoare triple $\{P\}C\{Q\}$ (where P and Q are assertions from our unbounded logic) holds iff[6]

$$\forall \sigma, s.\ valid(\sigma) \wedge \sigma, s \models P \Longrightarrow \neg(\langle (\sigma, s), C \rangle \longrightarrow \bot) \wedge (\forall \sigma', s'.\ \langle (\sigma, s), C \rangle \longrightarrow (\sigma', s') \Rightarrow \sigma', s' \models Q)$$

Thus, the unbounded states that satisfy P and Q do not matter for the validity of the Hoare triple $\{P\}C\{Q\}$. Formally, we have proved that $\{P\}s\{Q\} \equiv_\Delta \{\lceil P \rceil\}s\{Q\}$ and $\{P\}s\{Q\} \equiv_\Delta \{P\}s\{\lceil Q \rceil\}$. These two rules allow us to recover explicit non-aliasing information at statement boundaries. Recall that $\lceil P \rceil$ trivially holds in invalid states; thus $\lceil P \rceil$ can be "strengthened" to $\lceil P' \rceil$, if P entails P' in *valid states.* For instance, in a state with exclusive ownership of both `x.f` and `y.f` we can prove that `x` and `y` are not aliases with the valid entailment $\lceil \texttt{x.f} \mapsto \_ * \texttt{y.f} \mapsto \_ \rceil \models_\Delta \lceil \texttt{x.f} \mapsto \_ * \texttt{y.f} \mapsto \_ * \texttt{x} \neq \texttt{y} \rceil$.

The frame rule holds in the unbounded logic. We now show that, if the frame rule holds for the semantics described by $\langle ., . \rangle \longrightarrow .$ in bounded SL, then it also holds in our unbounded logic. To

[6]To ease reading, we omit the fixed interpretation Δ from this definition.

prove this, we use two properties that have been shown to be equivalent to the frame rule: *safety monotonicity* and the *frame property* [Calcagno et al. 2007]. Safety monotonicity states that if executing C in a (valid) state σ is safe, i.e. does not lead to an error, then it is also safe in larger (valid) states. More formally, C is safety monotonic iff for all valid states σ and σ' and local stores s, $\sigma' \geq \sigma$ and $\neg(\langle(\sigma, s), C\rangle \longrightarrow \bot)$ implies $\neg(\langle(\sigma', s), C\rangle \longrightarrow \bot)$. The frame property states that if executing C in a valid state σ_0 is safe, and executing C in a larger valid state $\sigma_0 \oplus \sigma_1$ leads to a state σ', then there must exist a valid state σ'_0 such that $\sigma' = \sigma'_0 \oplus \sigma_1$ and executing C in σ_0 leads to σ'_0.

We have proven in Isabelle/HOL that the soundness of the frame rule in the bounded logic implies the soundness of the frame rule in the unbounded logic (with the usual definitions).

Theorem 2. ***Frame rule in the unbounded logic.***
Assume that the program statement C is safety monotonic and satisfies the frame property. If $\{P\}C\{Q\}$ holds and $modified(C) \cap freeVars(R) = \varnothing$, then $\{P * R\}C\{Q * R\}$ also holds.

In contrast, this general frame rule does not hold with the strong star $\circledast$ from Brotherston et al. [2020], where the frame rule is restricted to program statements C that do not receive any resources (e.g. by acquiring a lock or receiving a message in a concurrent program).

Parallel rule. The soundness of the *Parallel* rule in the unbounded logic could be proven by adapting the proof from Vafeiadis [2011]. We have not attempted this proof in Isabelle/HOL, since reconstructing the full details in a new formalisation in Isabelle would require lengthy orthogonal work without leading to new insights beyond those from Vafeiadis. However, we are confident that his proof of the CSL soundness can be transferred to the unbounded logic, with almost no change required. Indeed, in this proof (ignoring the invariant and store for simplicity), Hoare triples are defined via a predicate $safe_n(C, h, Q)$, which informally says that it is safe to execute n steps of the command C starting in a *normal* heap (i.e. without fractional permissions, only zero and full permissions) larger than h, and this execution will lead only to final states that satisfy Q. The key point is that if h is an invalid state, then $safe_n(C, h, Q)$ trivially holds: h has more than full permission to one heap location, and thus there is no normal heap larger than h. This matches our interpretation of Hoare triples for invalid states. Moreover, since the CSL rules do not use the magic wand (which has a slightly different meaning in the unbounded logic), they would also have the same meaning in the unbounded logic.

3 COMBINABLE ASSERTIONS

It is often useful to split some resource (with the *Split* rule from Fig. 4) into two (or more) fractions, and to recombine these fractions later. As illustrated by the example in Fig. 1, splitting is typically used to enable threads to concurrently read the same heap data structure. Recombining the fractions is then crucial to get back exclusive ownership, and thus to be able to modify the data structure.

However, combining fractions of the same resource is not always sound, i.e. the entailment $A^\alpha * A^\beta \models_\Delta A^{\alpha+\beta}$ is in general not valid. As a simple example, consider the disjunction $A := \texttt{x.f} \mapsto \_ \vee \texttt{x.g} \mapsto \_$. A holds in a state σ_f (resp. σ_g) with full ownership of x.f (resp. x.g) and no other ownership. Thus, by definition, $A^{0.5}$ holds in $0.5 \odot \sigma_f$ and $0.5 \odot \sigma_g$. However, A does *not* hold in the state $(0.5 \odot \sigma_f) \oplus (0.5 \odot \sigma_g)$, because this state has only half ownership of both x.f and x.g, and thus it satisfies neither disjunct of A. A is thus not *combinable*, in the following sense:

Definition 4. *An assertion A is* ***combinable*** *with respect to an interpretation context Δ, written $combinable_\Delta(A)$, iff for all scalars α and β, $A^\alpha * A^\beta \models_\Delta A^{\alpha+\beta}$.*

$$\frac{combinable_\Delta(B)}{combinable_\Delta(A \mathrel{-\!\!*} B)} \qquad \frac{combinable_\Delta(A) \quad combinable_\Delta(B)}{combinable_\Delta(A * B)} \qquad \frac{combinable_\Delta(A) \quad combinable_\Delta(B)}{combinable_\Delta(A \wedge B)}$$

$$\frac{}{combinable_\Delta(A) \Longleftrightarrow combinable_\Delta(A^\pi)} \qquad \frac{pure(A)}{combinable_\Delta(A)} \qquad \frac{pure(A) \quad combinable_\Delta(B)}{combinable_\Delta(A \Rightarrow B)}$$

$$\frac{combinable_\Delta(A)}{combinable_\Delta(\forall x.\, A)} \qquad \frac{combinable_\Delta(A)}{combinable_\Delta(A^*)} \qquad \frac{combinable_\Delta(A) \quad unambiguous_\Delta(A, v)}{combinable_\Delta(\exists v.\, A)}$$

Fig. 5. Rules for reasoning about combinable (non-recursive) assertions in the unbounded logic.

Informally, if we restrict[7] α and β such that $\alpha + \beta = 1$, an assertion A is combinable iff the set of states that satisfy A is convex, in the sense that for any two states σ and σ' satisfying A, the set of all combinations $\alpha \odot \sigma \;\oplus\; (1-\alpha) \odot \sigma'$ (for $0 < \alpha < 1$) all also satisfy A. Intuitively, one can think of the two states σ and σ' as the states satisfying the conjuncts on the left-hand side of combinability, and their combinations as the states satisfying the right-hand side. The set of combinations can be thought of as a line segment between σ and σ'.

As explained in Sect. 1.3, Le and Hobor [2018] have proven that *precise* assertions [O'Hearn et al. 2004] are combinable. Informally, an assertion A is precise iff, for any heap σ, A holds in *at most one* heap smaller than σ. In practice, many useful assertions are combinable but not precise, which shows that checking combinability indirectly via preciseness is too approximate. As a simple example, consider wildcard assertions A^*, introduced in Sect. 2. Because wildcard assertions are ideal to represent read-only duplicable permissions, they are pervasive in automatic SL verifiers such as VeriFast [Jacobs et al. 2011] (see for example Jacobs and Piessens [2011]) and Viper [Müller et al. 2016] (see for example Summers and Müller [2018]). Using our definition of combinability, we can simply prove that a wildcard assertion A^* is combinable if A is combinable, and this property is effectively assumed by both verifiers. However, wildcard assertions are not precise. Therefore, we focus, in this work, on the combinability property itself instead of using preciseness as a (strictly less useful) proxy. The rules in Fig. 5 formalise the combinability of non-recursive assertions. We have proved the following lemma in Isabelle/HOL:

Theorem 3. *All rules presented in Fig. 5 hold in the unbounded logic.*

These rules can be used to prove that an assertion is combinable. As an example, to prove that $A * B$ is combinable, it suffices to prove that A and B are combinable. Notice that the assertion $\exists v.\, A$ is combinable if A is combinable *and* if A is unambiguous in v. Intuitively, this means that, for a given state σ, there is at most one value of v such that A holds in σ, otherwise the existential could act like a (potentially unbounded) disjunction. This rule is crucial to prove that assertions such as $\exists v.\, \texttt{x.f} \mapsto v * A$ are combinable, provided that A is combinable. Formally, given an interpretation Δ, an assertion A is unambiguous in v, written $unambiguous_\Delta(A, v)$, iff the following holds:

$$\forall \sigma_1, \sigma_2, s, v_1, v_2.\, \sigma_1 \# \sigma_2 \wedge \sigma_1, s(v := v_1), \Delta \models A \wedge \sigma_2, s(v := v_2), \Delta \models A \Rightarrow v_1 = v_2$$

$\exists v.\, \texttt{x.f} \mapsto v$ is trivially unambiguous in v. Moreover, if A is unambiguous in v, then $A * B$ is also unambiguous in v. We can thus derive the following useful rule:

$$\frac{combinable_\Delta(A)}{combinable_\Delta(\exists v.\, l \overset{p}{\mapsto} v * A)}$$

[7] We have proven in Isabelle/HOL that the two definitions are equivalent.

The first rule of Fig. 5 shows a key result: the magic wand is combinable in the unbounded logic, whereas it is not in bounded SL. Consider again the wand $W_1 = \texttt{x.f} \overset{0.5}{\mapsto} \_ \mathrel{-\!\!*} (\texttt{x.f} \overset{0.5}{\mapsto} \_ * \texttt{y.g} \mapsto \_)$. W_1 can be satisfied in bounded SL by providing (1) full permission to `y.g` (such that the right-hand side of the wand holds), or (2) more than half permission to `x.f` (such that we obtain an inconsistent state when combined with the left-hand side of the wand). A combination of half a state that satisfies (1) and half a state that satisfies (2), i.e. a state with half permission to both `x.f` and `y.g`, does *not* satisfy W_1, which shows that W_1 is not combinable in the bounded logic. However, in the unbounded logic, W_1 can be satisfied only by satisfying (1), which ensures that W_1 is combinable.

4 COMBINABLE (CO)INDUCTIVE PREDICATES

Sect. 3 provides rules to prove that *non-recursive* assertions are combinable. For example, using the rules from Fig. 5, it is easy to prove that the assertion $x \neq \texttt{null} \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r$ is combinable. However, these rules are not sufficient on their own to prove that (co)inductively-defined predicates are combinable, but this property is required to prove practical examples. For instance, the proof outline in Fig. 1 is valid (in the unbounded logic) only if $\mathit{tree}(x)$ is combinable. Recall that $\mathit{tree}(x)$ is defined inductively via the following equation: $\mathit{tree}(x) = (x \neq \texttt{null} \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * \mathit{tree}(x_l) * \mathit{tree}(x_r))$. Our goal is to provide the formal foundations to justify a proof that $\mathit{tree}(x)$ is combinable by induction: Assuming that $\mathit{tree}(x_l)$ and $\mathit{tree}(x_r)$ are combinable, it would then be straightforward to prove that $\mathit{tree}(x)$ is also combinable, using the recursive definition of $\mathit{tree}(x)$ and the rules from Fig. 5.

In this section, we formalise the mathematics necessary to enable such intuitive proofs, which turns out to be non-trivial for general SL assertions. In Sect. 4.1, we formalise the meaning of (co)inductive predicates in our assertion language via the concepts of least and greatest fixed points of a recursive equation, and use Knaster-Tarski's theorem to prove that (under some conditions) these fixed points exist. We also explain why the standard induction principle derived from this theorem is not sufficient to prove that these fixed points are combinable. We then define, in Sect. 4.2, a class of *set-closure properties*, which captures properties such as combinability and (an assertion) being intuitionistic. Moreover, we formalise and prove a novel, simple, and powerful induction principle for set-closure properties and fixed points: If a non-decreasing (defined in Sect. 4.1) function f preserves a set-closure property P, then the least and the greatest fixed point of f satisfy P. This novel induction principle captures the intuition described above. Proving this induction principle requires transfinite induction: We show in Sect. 4.3 why Kleene's fixed point theorem (which does not require transfinite induction) is not sufficient to prove this induction principle for some recursive predicate definitions that can be expressed in our assertion language.

4.1 Preliminaries: Monotonic Functions and Existence of Fixed Points

A recursive equation might have zero, some, or infinitely many fixed points. For example, any interpretation for $\mathbf{P}$ is a fixed point of the recursive equation $\mathbf{P} = \mathbf{P}$, and thus, fixed points of this simple recursive equation are in general not combinable. Two types of fixed points are typically used in SL: The *least fixed point*, and the *greatest fixed point*. Predicates interpreted as a least (resp. greatest) fixed point are referred to as *inductive* (resp. *coinductive*) predicates. Inductive predicates are particularly suitable to describe finite data structures. As an example, the least fixed point of the recursive equation for $\mathit{tree}(x)$ describes all finite binary trees. On the other hand, coinductive predicates can describe infinite data structures, and are useful to describe infinite sets of permissions, for instance, to specify the input/output behaviour of reactive programs [Penninckx et al. 2015].

The fixed points we are interested in are *interpretation contexts*, i.e. functions mapping a store of local variables to the set of states satisfying the predicate instance $\mathbf{P}$ (see Sect. 2.3). As an example, *tree* can be seen as an interpretation context Δ_t, which takes as input a store of local variables (containing in particular a value for the variable x), and outputs the set of states that satisfy $tree(x)$. Moreover, a recursive definition can be described with an assertion, using the symbol **P** for recursive calls. More precisely, given an assertion A (which might contain the symbol **P**) that represents the potentially recursive definition of our predicate, we define the interpretation of our predicate as the least or greatest fixed point of the function f_A, which we define as follows:

$$f_A := \lambda\Delta.\, \lambda s.\, \{\sigma \mid \sigma, s, \Delta \models A\}$$

The function f_A takes an interpretation context Δ and constructs a new interpretation context, by constructing, for any local store s (defining values for the variables corresponding to predicate parameters), the set of states that satisfy A (recall that an interpretation context maps a local store to a set of states), where the meaning of **P** is given by the interpretation context Δ. As an example, we can define the interpretation context Δ_t for our *tree* predicate as the least fixed point of f_A, for $A := (x \neq \texttt{null} \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * \mathbf{P}(x_l) * \mathbf{P}(x_r)))$.[8]

To formally define the meaning of the *least* and *greatest* fixed point of such a function, we need to define an order on interpretation contexts. Informally, an interpretation context Δ is smaller than another interpretation context Δ' iff Δ "semantically entails" Δ'. More precisely:

Definition 5. *An **interpretation context** is a function mapping a local store of variables to a set of states from Σ. An interpretation context Δ is **smaller than** another interpretation context Δ', written $\Delta \sqsubseteq \Delta'$, iff $\forall s.\, \Delta(s) \subseteq \Delta'(s)$.*

Lemma 1. *The set of interpretation contexts equipped with the partial order relation $\sqsubseteq$ is a* complete lattice. *In particular, for a family of interpretation contexts S:*

- *The* supremum *(or* join*) of S, written $\sqcup S$, can be obtained as $\sqcup S := \lambda s.\, \{\sigma \mid \exists \Delta \in S.\, \sigma \in \Delta(s)\}$.*
- *The* infimum *(or* meet*) of S, written $\sqcap S$, can be obtained as $\sqcap S := \lambda s.\, \{\sigma \mid \forall \Delta \in S.\, \sigma \in \Delta(s)\}$.*

A *fixed point* of a function f is an interpretation Δ such that $f(\Delta) = \Delta$. The *least* (resp. *greatest*) fixed point of a function f is a fixed point that is smaller (resp. larger) than all other fixed points of f, with respect to the partial order $\sqsubseteq$. Knaster-Tarski's theorem states that any *non-decreasing* function f has a least and a greatest fixed point [Tarski 1955].

Definition 6. *A function f is **non-decreasing**, written $mono^+(f)$, iff $\forall \Delta, \Delta'.\, \Delta \sqsubseteq \Delta' \Rightarrow f(\Delta) \sqsubseteq f(\Delta')$.*

We have proven in Isabelle/HOL that the function f_A is non-decreasing if **P** occurs only in positive positions.

Theorem 4. ***Knaster-Tarski fixed point construction***
Let $LFP(f) := \sqcap\{\Delta \mid f(\Delta) \sqsubseteq \Delta\}$ and $GFP(f) := \sqcup\{\Delta \mid \Delta \sqsubseteq f(\Delta)\}$. If $mono^+(f)$, then $LFP(f)$ is the least fixed point of f and $GFP(f)$ is the greatest fixed point of f.

In addition to the existence of a least (and a greatest) fixed point of a function f, this theorem gives us an induction principle for this fixed point: If an interpretation Δ satisfies $f(\Delta) \sqsubseteq \Delta$, then it is greater or equal to $LFP(f)$, because $LFP(f)$ is the infimum of the set of such interpretations (a similar induction principle can be derived for $GFP(f)$). This induction principle allows one to prove properties about each individual state of $LFP(f)$, by choosing a relevant Δ. For example, let $\Delta_P(s)$ (for all s) be the set of all states that satisfy a property P (e.g. owning `x.f` with value 5). If f preserves the property P, i.e. $f(\Delta_P) \subseteq \Delta_P$, then $LFP(f) \subseteq \Delta_P$; in other words, all states in $LFP(f)(s)$ (for all s) satisfy P.

[8]Recall that **P** does not take explicit arguments in our syntax; $\mathbf{P}(x_l)$ (resp. $\mathbf{P}(x_r)$) is syntactic sugar for $\exists x.\, x = x_l * \mathbf{P}$ (resp. $\exists x.\, x = x_r * \mathbf{P}$).

However, it does not appear possible to apply this induction principle to the combinability property, since combinability is not a property of individual states in a set of states (such as P above), but rather of unboundedly large subsets of such a set. Combinability concerns the (infinite) space of all combinations of two states (similar to a convexity property, as explained in Sect. 3).

4.2 An Induction Principle for (Co)Inductive Predicates and Set-Closure Properties

Given a non-decreasing function f, Theorem 4 expresses that f has a least fixed point ($\mathit{LFP}(f)$) and a greatest fixed point ($\mathit{GFP}(f)$), and provides induction principles to reason about these fixed points. However, as explained above, these induction principles do not appear sufficient to prove that these fixed points are combinable. On the other hand, Cousot and Cousot [1979] have proven that, if $\mathit{mono}^{+}(f)$, then $\mathit{LFP}(f)$ (resp. $\mathit{GFP}(f)$) can be expressed as the stationary limit of $f^{\alpha}(\Delta_{\bot})$ (resp. $f^{\alpha}(\Delta_{\top})$), where α ranges over ordinals, f^{α} is defined by transfinite recursion, and $\Delta_{\bot}$ (resp. $\Delta_{\top}$) is defined as the *empty* (resp. *full*) interpretation, as given by the following definition.

Definition 7. *The* ***empty interpretation****, written $\Delta_{\bot}$, maps all stores to the empty set $\emptyset$ (representing the assertion false), i.e. $\forall s.\, \Delta_{\bot}(s) := \emptyset$. The* ***full interpretation****, written $\Delta_{\top}$, maps all stores to the universal set Σ (representing the assertion true), i.e. $\forall s.\, \Delta_{\top}(s) := \Sigma$.*
Given a function f, f^{α} (where α is an ordinal) is defined by transfinite recursion as follows:

- *For an ordinal α, $f^{\alpha+1} := \lambda\Delta.\, f(f^{\alpha}(\Delta))$.*
- *For a limit ordinal γ, $f^{\gamma} := \lambda\Delta.\, \sqcup\{f^{\beta}(\Delta) \mid \beta < \gamma\}$.*

We show in Sect. 4.3 why Kleene's fixed point theorem cannot be applied to prove that a fixed point of some recursive predicate definitions in our assertion language is combinable, which justifies our use of ordinals and transfinite induction. Using these constructive definitions of $\mathit{LFP}(f)$ and $\mathit{GFP}(f)$, we can express our induction principle for *set-closure properties*, after we define the latter.

Definition 8. *A predicate P on interpretation contexts (i.e. a function from interpretation contexts to Booleans) is a* ***set-closure property*** *iff P satisfies the following:*

$$\exists M.\, \forall \Delta.\, (P(\Delta) \iff \forall s.\, (\forall a, b \in \Delta(s).\, M(a, b) \subseteq \Delta(s)))$$

Intuitively, a set-closure property corresponds to being closed under some operation M, which constructs (from two states) a set of states. As an example, an assertion A is combinable iff it is closed under the operation M that informally constructs the line segment between two states, or, more formally, under the operation $M(a, b) := \{\sigma \mid \exists p, q.\, p + q = 1 \wedge \sigma = p \odot a \oplus q \odot b\}$; combinability is thus a set-closure property. As another example, an assertion A is intuitionistic iff it is upward closed (formally corresponding to the operation $M(a, b) := \{\sigma \mid \sigma \succeq a\}$), which shows that the property of (an assertion) being intuitionistic is also a set-closure property.

We have proven the following induction principle in Isabelle/HOL.

Theorem 5. ***Induction principle for set-closure properties.***
Let f be a non-increasing function (i.e. $\mathit{mono}^{+}(f)$) and P a set-closure property. If f preserves P, i.e. $\forall \Delta.\, P(\Delta) \Rightarrow P(f(\Delta))$, then $P(\mathit{LFP}(f))$ and $P(\mathit{GFP}(f))$ hold.

This theorem justifies the intuitive induction described at the beginning of this section when P is the combinability property: To prove that $\mathit{tree}(x)$ is combinable, we simply have to prove that the assertion $x \neq \texttt{null} \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * \mathit{tree}(x_l) * \mathit{tree}(x_r)$ is combinable (corresponding to $P(f(\Delta))$), while assuming that $\mathit{tree}(y)$ is combinable for all y (corresponding to $P(\Delta)$), which we can do using the rules from Fig. 5. Moreover, this theorem can be easily leveraged in the context of automatic SL verifiers, as we show in Sect. 5: If the assertion language for defining predicates recursively is restricted in ways that are standard in such tools, we directly get that all (co)inductive predicates are combinable.

4.3 Kleene's Fixed Point Theorem is too Restrictive for SL

Some readers might wonder why we used ordinals and transfinite induction to prove Theorem 5, instead of (the simpler) Kleene's fixed point theorem. The reason is that Kleene's theorem forces a stronger assumption on f, namely *Scott-continuity.* The theorem states that, if a function f is *Scott-continuous*, then its least fixed point can be computed as the supremum of $f^n(\Delta_\bot)$, where n ranges over natural numbers, and $\Delta_\bot$ is the empty interpretation. Unfortunately, the rich connectives commonly-employed in separation logics easily violate this requirement. Using the universal quantifier or the magic wand in our recursive definition A is enough to make f_A not Scott-continuous. Worse, using the existential quantifier or the separating conjunction in A is enough for f_A to not satisfy the dual property of Scott-continuity, which is required to prove that $GFP(f_A)$ is the infimum of ${f_A}^n(\Delta_\top)$ (where $\Delta_\top$ is the full interpretation). The following example illustrates the problem.

Example 3. *Consider the recursive definition* $A := (\texttt{x.g} \mapsto \_)^* \mathrel{-\!\!*} (\texttt{x.g} \mapsto \_ \vee A^{0.5})$, *interpreted in an intuitionistic manner.*[9] *Recall that* $(\texttt{x.g} \mapsto \_)^*$ *represents an unspecified positive permission amount. Let* f_A *denote the function associated with this recursive definition, and* Δ_p *an interpretation context such that, for all stores* s*, a state* σ *is in* $\Delta_p(s)$ *iff* σ *has at least* p *permission to* $\texttt{x.g}$*.*

f_A *is* not *Scott-continuous, and thus Kleene's theorem does not apply. To see why, let us nonetheless compute* $f^n(\Delta_\bot)$*. Starting from the empty interpretation* $\Delta_\bot$*, we get* $f_A(\Delta_\bot) = \Delta_1$*: We need* 1 *permission of* $\texttt{x.g}$ *to prove the right-hand side of the wand* $\texttt{x.g} \mapsto \_ \vee A^{0.5} \equiv \texttt{x.g} \mapsto \_$*. Then,* ${f_A}^2(\Delta_\bot) = f_A(\Delta_1) = \Delta_{0.5}$*, since, in this case, we can prove the disjunct* $A^{0.5}$ *to prove the right-hand side. Similarly,* ${f_A}^3(\Delta_\bot) = f_A(\Delta_{0.5}) = \Delta_{0.25}$*. By induction, we get* ${f_A}^{n+1}(\Delta_\bot) = \Delta_{\frac{1}{2^n}}$*.*

We can now apply Kleene's formula to obtain a potential *least fixed point: The supremum of* $f^n(\Delta_\bot)$ *is* $\Delta_{>0}$*, where a state is in* $\Delta_{>0}(s)$ *(for all* s*) iff it has non-zero permission to* $\texttt{x.g}$*. However,* $\Delta_{>0}$ *is* not *a fixed point of* f_A*, since* $f_A(\Delta_{>0}) = \Delta_\top$ *(the full interpretation). Indeed, in this case, the wand is always trivially satisfied, since the left-hand side implies the right-hand side.*

This example shows that Kleene's theorem is too restrictive to justify the existence of a least fixed point for some recursive SL predicate definitions. The situation is similar for Kleene's dual theorem (existence of a greatest fixed point), because of the existential quantifier and the separating conjunction. As an example, the greatest fixed point of the recursive equation $A := \texttt{x.g} \overset{0.5}{\mapsto} \_ * (\texttt{x.g} \mapsto \_)^* * A^{0.5}$ is $\Delta_\bot$. However, Kleene's dual formula for the greatest fixed point (i.e. the infimum of $f^n(\Delta_\top)$) yields Δ_1, which is not a fixed point of this equation, because $f_A(\Delta_1) = \Delta_\bot$. The richer mathematical foundations we provide in this section are needed to enable direct proofs of combinability over general recursively-defined SL predicates.

5 FORMAL FOUNDATIONS FOR FRACTIONAL PREDICATES AND MAGIC WANDS IN AUTOMATIC SL VERIFIERS

Fractional resources, in the form of fractional predicates, are supported by several automatic SL verifiers, such as VerCors [Blom and Huisman 2014], VeriFast [Jacobs et al. 2011], and Viper [Müller et al. 2016]. As explained in Sect. 1, this support relies on the concept of a *syntactic multiplication.* For example, the semantics of fractional resources in VeriFast is explicitly defined as follows: "applying a coefficient f to a user-defined predicate is equivalent to multiplying the coefficient of each chunk mentioned in the predicate's body by f" [Jacobs et al. 2011]. VerCors and Viper perform a similar

[9]By "interpreted in an intuitionistic manner", we mean that the assertion A holds in any state that satisfies *at least* the magic wand, i.e. A holds in any state that owns the magic wand and possibly other resources. In classical SL, this interpretation can be obtained by considering $A * \mathit{true}$ instead of A, where the left conjunct captures the wand, and the right conjunct captures the other resources.

$$
\begin{array}{ll}
\pi \cdot (A * B) := (\pi \cdot A) * (\pi \cdot B) & \pi \cdot (A \wedge B) := (\pi \cdot A) \wedge (\pi \cdot B) \\
\pi \cdot (A \mathrel{-\!\!*} B) := (\pi \cdot A) \mathrel{-\!\!*} (\pi \cdot B) & \pi \cdot (A \vee B) := (\pi \cdot A) \vee (\pi \cdot B) \\
\pi \cdot (A \Rightarrow B) := (\pi \cdot A) \Rightarrow (\pi \cdot B) & \pi \cdot (\exists x.\, A) := \exists x.\, (\pi \cdot A) \\
\pi \cdot A^{\alpha} := (\pi * \alpha) \cdot A & \pi \cdot (\forall x.\, A) := \forall x.\, (\pi \cdot A) \\
\pi \cdot (A^{*}) := A^{*} & \pi \cdot A := A^{\pi} \qquad \text{(otherwise)}
\end{array}
$$

Fig. 6. Definition of the syntactic multiplication over assertions.

syntactic multiplication when *unfolding* (exchanging a predicate instance with its definition, also called *opening*) or *folding* (the reverse operation, also called *closing*) a fractional predicate.

However, as shown by Ex. 1, there is a mismatch between the syntactic and the semantic multiplication in the bounded logic. Consider $P(x, y) := \texttt{x.g} \mapsto \_ * \texttt{y.g} \mapsto \_$. While $P(x, x)^{0.5}$ is equivalent to false in bounded SL if interpreted with the semantic multiplication, the three verifiers allow the user to obtain this fractional predicate instance in exchange for full permission of `x.g`; this behaviour is compatible with the semantic multiplication in our novel unbounded logic.

In this section, we show that our unbounded logic can serve as a formal foundation for fractional predicates in automatic SL verifiers, since it gives a meaning to the syntactic multiplication performed by these verifiers, and justifies that fractions of the same predicate can be soundly recombined (under some restrictions). Moreover, using the unbounded logic as a formal foundation enables sound extensions of these verifiers, for example to handle fractional magic wands (which, to our knowledge, no verifier supports yet).

In Sect. 5.1, we define a syntactic multiplication over assertions, and show that it is equivalent to the semantic one in the unbounded logic. From this, we derive rules for fractional magic wands, which could easily be automated in VerCors and Viper. We then define, in Sect. 5.2, a simple syntactic restriction on recursive predicate definitions, which ensures the existence of a least and a greatest fixed point. This allows us to derive *fold* and *unfold* rules for fractional predicates, based on the syntactic multiplication, which formally justifies what VerCors, VeriFast, and Viper actually do. Finally, we define, in Sect. 5.3, a syntactic restriction on the definition of a predicate, which ensures that this predicate is combinable, using the results from Sect. 3 and Sect. 4.

5.1 Syntactic Multiplication and Fractional Magic Wands

Fig. 6 shows the definition of the syntactic multiplication over assertions, which we write $\pi \cdot A$ for a scalar π and an assertion A. The idea of this syntactic multiplication, which corresponds to what the three verifiers do, is straightforward: We push the multiplication inside, until we reach semantic assertions $\mathcal{B}$ or predicate **P**. The following theorem follows from the distributivity and factorisation rules shown in Fig. 4.

Theorem 6. *Syntactic and semantic multiplication are equivalent in unbounded logic:* $A^{\pi} \equiv_{\Delta} \pi \cdot A$

This result justifies the syntactic multiplication performed by the verifiers. Moreover, it can also be leveraged to improve the support for magic wands in automatic SL verifiers. Both VerCors and Viper support magic wands [Blom and Huisman 2015; Schwerhoff and Summers 2015], via two operations *package* and *apply*. Packaging a wand $A \mathrel{-\!\!*} B$ amounts to exchanging resources that satisfy the wand with an instance of the wand. Applying a wand $A \mathrel{-\!\!*} B$ boils down to giving up an instance of the wand $A \mathrel{-\!\!*} B$ and resources that satisfy A, in exchange for resources that satisfy B. However, neither VerCors nor Viper support packaging and applying fractions of wands.

These rules, describing how to package and apply fractional wands, hold in the unbounded logic:

$$\frac{F * (\pi \cdot A) \models_\Delta \pi \cdot B}{F \models_\Delta (A \mathrel{-\!\!*} B)^\pi}\ (PackageWand) \qquad \frac{}{(\pi \cdot A) * (A \mathrel{-\!\!*} B)^\pi \models_\Delta \pi \cdot B}\ (ApplyWand)$$

The rule *PackageWand* states that it is sound to give up the resources specified by F, which satisfy $\pi \cdot B$ when combined with $\pi \cdot A$, in exchange for a fraction π of the wand $A \mathrel{-\!\!*} B$. On the other hand, the rule *ApplyWand* states that it is sound to give up a fraction π of a wand $A \mathrel{-\!\!*} B$ and resources that satisfy $\pi \cdot A$, in exchange for resources that satisfy $\pi \cdot B$. Since these two rules rely on the syntactic multiplication, they could be easily added to VerCors and Viper, which have algorithms to compute F.

5.2 Folding and Unfolding Fractions of Recursively-Defined Predicates

To formally justify the way VeriFast, VerCors, and Viper handle recursively-defined predicates, we need to ensure the existence of a fixed point for all predicate definitions accepted by these tools. Indeed, the three verifiers assume that an instance of a recursively-defined predicate is a fixed-point of its recursive definition. All three verifiers enforce recursive calls to appear in positive positions when A is a recursive predicate definition, since (1) none supports implications whose left-hand side are not pure (i.e. specify resources, including predicate instances), and (2) VerCors and Viper do not allow magic wands inside predicate definitions; again, our work presents foundations for extending this support.

We write *correctRec*(A) iff recursive calls to $\mathbf{P}$ in A happen in positive positions only. To avoid duplicating rules, we write *FP* to refer indiscriminately to either *LFP* or *GFP*. We have proved in Isabelle/HOL the following theorem:

Theorem 7. *If correctRec(A) holds, then* $\forall \sigma, s.\ \sigma, s, FP(f_A) \models A \iff \sigma \in FP(f_A)(s)$.

Combining this result with Theorem 6, we can now prove that the two following rules, used by the three verifiers to fold and unfold fractions of predicates, are valid in the unbounded logic:

$$\frac{correctRec(A)}{\pi \cdot A \models_{FP(A)} \mathbf{P}^\pi}\ (Fold) \qquad \frac{correctRec(A)}{\mathbf{P}^\pi \models_{FP(A)} \pi \cdot A}\ (Unfold)$$

Fold allows one to give up resources that satisfy $\pi \cdot A$ in exchange for a fraction π of the predicate instance $\mathbf{P}$, which is defined (co)inductively by the equation $\mathbf{P} = A$. *Unfold* permits the reverse operation: to exchange a fraction π of the predicate instance $\mathbf{P}$ with resources that satisfy $\pi \cdot A$.

5.3 Combinability

Finally, we want to leverage results from Sect. 3 and Sect. 4 to prove that the rules used by VerCors, VeriFast, and Viper to combine fractions of predicates are valid in the unbounded logic. Both VerCors and Viper automatically combine fractions of the same predicate instance, which is currently sound (1) because of their restricted assertion languages and (2) because they forbid magic wands inside predicate definitions. Indeed, VerCors and Viper allow disjunctions, existentially-quantified assertions, and negations only of pure assertions. As explained in Sect. 3, the magic wand interpreted in the bounded logic is not combinable in general, and thus allowing wands inside predicate definitions *and* combining fractions of such a predicate instance would be unsound. Note that restriction (2) could be removed by interpreting wands in the unbounded logic.

In contrast, it is possible to write VeriFast predicates that are *not* combinable, e.g. using existential quantifiers. VeriFast thus performs a static analysis on a predicate definition to detect whether this predicate is combinable, and, if it is, VeriFast emits a lemma that permits combining two fractions of this predicate, which is formally justified by our unbounded logic.

$$
\begin{array}{ll}
comb(\mathbf{P}) \iff \top & comb(A \Rightarrow B) \iff pure(A) \land comb(B) \\
comb(\mathcal{B}) \iff \mathcal{B} \text{ is combinable} & comb(A \land B) \iff comb(A) \land comb(B) \\
comb(A^{\pi}) \iff comb(A) & comb(\exists x.\, A) \iff comb(A) \land unambiguous(A, x) \\
comb(A^{*}) \iff comb(A) & comb(\forall x.\, A) \iff comb(A) \\
comb(A * B) \iff comb(A) \land comb(B) & comb(A) \iff \bot \qquad \text{(otherwise)} \\
comb(A \mathrel{-\!\!*} B) \iff comb(B) &
\end{array}
$$

Fig. 7. Syntactic condition to ensure that an assertion is combinable.

To formally justify the behaviours of the three verifiers, we define in Fig. 7 a syntactic condition for an assertion A, $comb(A)$, which ensures that the assertion A is combinable. *comb* forbids semantic assertions that are not combinable, as well as disjunctions and implications with an impure left-hand side. Moreover, $unambiguous(A, x)$ can be conservatively checked syntactically, using the fact that $\exists v.\, \texttt{x.f} \mapsto v$ is trivially unambiguous in v, and the entailment $unambiguous(A, x) \implies unambiguous(A * B, x)$ for all A, B, and x. This is, in essence, what VeriFast does.

Finally, note that $comb(\mathbf{P})$ always holds. This way, we can leverage the induction principle from Sect. 4 (Theorem 5) to prove that predicates (co)inductively-defined with the recursive equation $\mathbf{P} = A$ such that $comb(A)$ holds are combinable. In particular, we have proven in Isabelle/HOL that the following rule, used by all three verifiers in some form, is valid in the unbounded logic:

$$\frac{comb(A) \quad correctRec(A)}{\mathbf{P}^{\alpha} * \mathbf{P}^{\beta} \models_{FP(A)} \mathbf{P}^{\alpha+\beta}}\ (Combinability)$$

6 EXAMPLES

The example from Fig. 1 is one illustration of the power and the simplicity of the unbounded logic; with our logic such direct concurrent-separation-logic proofs *just work*, without additional connectives and with essentially no restriction on the assertions involved in the specifications. The entailments inside the two parallel branches are justified by the rules *Unfold* and *Fold*, and the last entailment ($tree(x)^{\frac{\pi}{2}} * tree(x)^{\frac{\pi}{2}}$ entails $tree(x)^{\pi}$) is justified by the rule *Combinability* (since the recursive definition of $tree(x)$ satisfies the syntactic condition *comb* defined in Fig. 7).

In this section, we further illustrate the flexibility and the simplicity of the unbounded logic on two additional examples. The first example motivates the need for factorisability for the magic wand, while the second example, taken from Brotherston et al. [2020], shows that the unbounded logic provides an easy and intuitive way to reason about cross-thread data transfer.

6.1 Concurrently Reading a Subtree and a Tree

Consider the concurrent method `readBoth` in Fig. 8, which takes as input a reference `x` and an integer `key`. In this simple example, we start with a fraction π of a tree rooted in `x`. We then sequentially look for a subtree of `x` that matches `key`, using the method `find`, where `find` is specified as follows [Brotherston et al. 2020; Cao et al. 2019]

$$\{tree(x)^{\alpha}\}\ \texttt{find(x, key)}\ \{\lambda y.\, (tree(y) * (tree(y) \mathrel{-\!\!*} tree(x)))^{\alpha}\}$$

where y is bound to the return variable. $tree(y) \mathrel{-\!\!*} tree(x)$ expresses the ownership of all nodes of $tree(x)$, except the nodes from its subtree $tree(y)$. Combined with the subtree $tree(y)$, this magic wand gives us a simple way to get back the ownership of the entire tree, $tree(x)$.

Method `readBoth` then forks two threads, and each thread performs some action that first requires read access to the subtree `sub`, and then read access to the whole tree `x`. Thus, the

```
method readBoth(x: Ref, key: Int) {
  $\{tree(x)^{\pi}\}$
  sub := find(x, key)
  $\{(tree(sub) * (tree(sub) -* tree(x)))^{\pi}\}$
  $\{(tree(sub) * (tree(sub) -* tree(x)))^{\frac{\pi}{2}} * (tree(sub) * (tree(sub) -* tree(x)))^{\frac{\pi}{2}}\}$
        | $\{(tree(sub) * (tree(sub) -* tree(x)))^{\frac{\pi}{2}}\}$        || $\{(tree(sub) * (tree(sub) -* tree(x)))^{\frac{\pi}{2}}\}$        |
        | $\{tree(sub)^{\frac{\pi}{2}} * (tree(sub) -* tree(x))^{\frac{\pi}{2}}\}$ || $\{tree(sub)^{\frac{\pi}{2}} * (tree(sub) -* tree(x))^{\frac{\pi}{2}}\}$ |
        | readTree(sub)                                            || readTree(sub)                                            |
        | $\{tree(sub)^{\frac{\pi}{2}} * (tree(sub) -* tree(x))^{\frac{\pi}{2}}\}$ || $\{tree(sub)^{\frac{\pi}{2}} * (tree(sub) -* tree(x))^{\frac{\pi}{2}}\}$ |
        | $\{tree(x)^{\frac{\pi}{2}}\}$                               || $\{tree(x)^{\frac{\pi}{2}}\}$                               |
        | readTree(x)                                              || readTree(x)                                              |
        | $\{tree(x)^{\frac{\pi}{2}}\}$                               || $\{tree(x)^{\frac{\pi}{2}}\}$                               |
  $\{tree(x)^{\frac{\pi}{2}} * tree(x)^{\frac{\pi}{2}}\}$
  $\{tree(x)^{\pi}\}$
}
```

Fig. 8. A simple concurrent program that looks for a subtree of x that matches key, and then concurrently reads from both the tree rooted in x and in the subtree.

method readTree requires some fractional ownership of the tree it reads, i.e. it is specified as $\{tree(y)^{\alpha}\}$ readTree(y) $\{tree(y)^{\alpha}\}$. In this specification, α can be thought of as a ghost parameter; the method can be called for any (non-zero) fractional amount α. Finally, method readBoth joins the two threads, and returns the fractional ownership of $tree(x)$ it started with.

Proving that method readBoth satisfies its specification is straightforward in our unbounded logic. After the call to find, we split the fraction π of $tree(\mathtt{sub}) * (tree(\mathtt{sub}) -\!* \; tree(x))$ into two fractions $\frac{\pi}{2}$, and we give one fraction to each thread, using the rule *Parallel*. In each thread, we then distribute the fraction $\frac{\pi}{2}$ over the star, to justify the call readTree(sub).

After this call, we need to justify that we can read the tree rooted in x, which we achieve by first *distributing the fraction* $\frac{\pi}{2}$ over the wand $tree(\mathtt{sub}) -\!* \; tree(x)$, and then by applying the wand. Crucially, note that this step is invalid in the bounded logic, since the distributivity property does not hold for the wand! Moreover, this step would also be invalid with the weak or the strong wand from Brotherston et al. [2020], and even if we used the binary tree share model from Le and Hobor [2018]. Finally, since we (syntactically) know that $tree(x)$ is *combinable*, we recombine the two fractions $\frac{\pi}{2}$ of $tree(x)$ after the threads have finished executing, which concludes the proof.

6.2 Cross-thread Data Transfer

We also illustrate our unbounded logic on an example from Brotherston et al. [2020], which involves message-passing concurrency, with simplified Hoare rules [Bell et al. 2010; Hobor and Gherghina 2012; Leino et al. 2010; Villard et al. 2009]: Given a channel c, a message number i, and an associated message invariant R_i^c, the rule to send message i via channel c is $\{R_i^c(x)\}$ send(c, x) $\{emp\}$, whereas the rule to receive this message is $\{emp\}$ y := receive(c) $\{R_i^c(y)\}$.

The method transfer first creates a binary tree by calling the method makeTree(), and then forks two threads. The first thread calls the same method find as in the previous example, to find a subtree rooted in s that matches the key, and sends the reference s to the second thread via the channel ch. The second thread receives reference s, and then modifies the tree rooted in s by calling modify, which thus requires exclusive ownership of the tree rooted in s. After the modification, the second thread notifies the first one, and both terminate. Finally, the tree rooted in x is deleted (alternatively, full access could be returned, but this code is from Brotherston et al. [2020]).

```
method transfer(key: Int) {
  $\{emp\}$
  x := makeTree()
  $\{tree(x, \tau_x)\}$
  $\{tree(x, \tau_x)^{\frac{1}{2}} * tree(x, \tau_x)^{\frac{1}{2}}\}$
    | $\{tree(x, \tau_x)^{\frac{1}{2}}\}$
    | s := find(x, key)
    | $\{(tree(s, \tau_s) * (tree(s, \tau_s) \mathrel{-\!\!*} tree(x, \tau_x)))^{\frac{1}{2}} * sub(s, \tau_x)\}$
    | $\{(tree(s, \tau_s)^{\frac{1}{2}} * sub(s, \tau_x)) * (tree(s, \tau_s) \mathrel{-\!\!*} tree(x, \tau_x))^{\frac{1}{2}}\}$
    | send(ch, s)
    | $\{(tree(s, \tau_s) \mathrel{-\!\!*} tree(x, \tau_x))^{\frac{1}{2}}\}$
    | $\{(tree(s, \tau_s) \mathrel{-\!\!*} tree(x, \tau_x))^{\frac{1}{2}}\}$
    | receive(ch)
    | $\{tree(s, \tau_s)^{\frac{1}{2}} * (tree(s, \tau_s) \mathrel{-\!\!*} tree(x, \tau_x))^{\frac{1}{2}}\}$
    | $\{tree(x, \tau_x)^{\frac{1}{2}}\}$
  ||
    | $\{tree(x, \tau_x)^{\frac{1}{2}}\}$
    | $\{tree(x, \tau_x)^{\frac{1}{2}}\}$
    | s := receive(ch)
    | $\{(tree(s, \tau_s)^{\frac{1}{2}} * sub(s, \tau_x)) * tree(x, \tau_x)^{\frac{1}{2}}\}$
    | $\{tree(s, \tau_s) * (tree(s, \tau_s)^{\frac{1}{2}} \mathrel{-\!\!*} tree(x, \tau_x)^{\frac{1}{2}})\}$
    | modify(s)
    | $\{tree(s, \tau_s) * (tree(s, \tau_s)^{\frac{1}{2}} \mathrel{-\!\!*} tree(x, \tau_x)^{\frac{1}{2}})\}$
    | $\{tree(s, \tau_s)^{\frac{1}{2}} * tree(x, \tau_x)^{\frac{1}{2}}\}$
    | send(ch, ())
    | $\{tree(x, \tau_x)^{\frac{1}{2}}\}$
    | $\{tree(x, \tau_x)^{\frac{1}{2}}\}$
  $\{tree(x, \tau_x)^{\frac{1}{2}} * tree(x, \tau_x)^{\frac{1}{2}}\}$
  $\{tree(x, \tau_x)\}$
  deleteTree(x)
  $\{emp\}$
}
```

Fig. 9. Cross-thread data transfer from Brotherston et al. [2020]. `find` is specified as in the previous example. `modify(s)` requires exclusive ownership of the tree rooted in s. The first message invariant is $tree(s, \tau_s)^{0.5} * sub(s, \tau_x)$, and the second message invariant is $tree(s, \tau_s)^{0.5}$.

To verify method `transfer`, we need to transmit from the first to the second thread the knowledge that s is a node that belongs to the tree rooted in x. It is standard to express such information about heap values by adding a second parameter to the predicate *tree*, representing a mathematical abstraction of the tree structure; for a tree rooted at x we will write τ_x for the corresponding mathematical tree. We require our tree abstraction to include the reference identities of each node. The pure function $sub(s, \tau_x)$, which is then easy to write inductively over these mathematical trees, expresses that the reference s belongs to the tree τ_x.

The second thread needs this piece of knowledge to prove that $tree(x, \tau_x)^{\frac{1}{2}}$ can be decomposed into $tree(s, \tau_s)^{\frac{1}{2}} * (tree(s, \tau_s)^{\frac{1}{2}} \mathrel{-\!\!*} tree(x, \tau_x)^{\frac{1}{2}})$ (using a simple inductive lemma), in order to prove it has exclusive ownership of $tree(s, \tau_s)$. Therefore, our first message invariant is $tree(s, \tau_s)^{\frac{1}{2}} * sub(s, \tau_x)$. After the second thread has received the first message, it can use the aforementioned entailment to justify exclusive ownership of $tree(x, \tau_x)$, and thus call `modify`. After this call, the second thread applies the magic wand to get back half ownership of both $tree(x, \tau_x)$ and $tree(s, \tau_s)$, and it sends $tree(s, \tau_s)^{\frac{1}{2}}$ via the channel; hence our second message invariant is $tree(s, \tau_s)^{\frac{1}{2}}$. The first thread then receives $tree(s, \tau_s)^{\frac{1}{2}}$, and uses the distributivity of the magic wand to get back half ownership of $tree(x, \tau_x)$. Finally, the two threads terminate, and method `transfer` deletes the tree.

Comparison. In contrast to the approach from Brotherston et al. [2020], our unbounded logic requires us to add a mathematical tree abstraction to the predicate *tree*, transfer the knowledge that s points to a node in τ_x, and prove a simple inductive lemma about tree decomposition. These kinds of reasoning steps are standard in separation logic proofs and required anyway to prove

richer functional specifications such as sortedness. Instead, Brotherston et al. use custom *assertion labels* and a *jump modality* in their logic. The first thread transmits the information that the tree rooted in x has not been modified since the beginning of the method using the label l_0 introduced in the precondition of `transfer`, via the following invariant for the first message:
$(l_1 \land \mathit{tree}(s))^{0.5} \land \left(@_{l_0}^{0.5} \left((l_1 \land \mathit{tree}(s)) \circledast (l_2 \land (\mathit{tree}(s) \mathbin{-\!\circledast} \mathit{tree}(x)))\right)^{0.5}\right)$.

The left conjunct specifies half of the ownership of the tree rooted in s, while the right conjunct contains some knowledge about the initial heap (labelled with l_0).

Our unbounded logic has the advantage that the message invariants are more concise and do not require non-standard connectives in specifications; our first message invariant is $\mathit{tree}(s, \tau_s)^{\frac{1}{2}} * \mathit{sub}(s, \tau_x)$. This advantage is even better illustrated with the second message invariant, which they specify as $(l_0 \land \mathit{tree}(s))^{0.5} \land l_2 \perp l_3 \land \left(@_2^{0.5} ((l_3 \land \mathit{tree}(s)) \mathbin{-\!\circledast} (l_4 \land \mathit{tree}(x)))^{0.5}\right)$, where $l_2 \perp l_3$ is another clever but non-standard construct which expresses that the heaps represented by l_2 and l_3 are disjoint. By contrast, our second message invariant is simply $\mathit{tree}(s, \tau_s)^{\frac{1}{2}}$.

7 RELATED WORK

Multiplication with permissions. SL has been extended with different permission models, including fractional permissions [Bornat et al. 2005; Boyland 2003], counting permissions [Bornat et al. 2005], named permissions [Parkinson 2005], and binary tree shares [Dockins et al. 2009]. Although these interoperate well with simple points-to predicates, when considering general fractional resources, none of them provides the key properties of distributivity, factorisability and combinability. Some of the weaknesses have been previously identified [Brotherston et al. 2020; Le and Hobor 2018], but as we discuss in detail in Sect. 1.3 and Sect. 6.2, the alternatives presented there introduce new complexities to specifications without providing these three properties for their logics in general.

More-general fractional ownership was (to our knowledge) first explored by Boyland [2010], who defines the concept of *nesting*. Nesting enables a heap location l to own some fraction π of a resource A; owning a fraction α of the location l then results in owning a fraction $\alpha \cdot \pi$ of A. Moreover, Boyland permits fractions above 1 in intermediate fractional heaps to get the useful equality $\sigma = \pi \odot (\frac{1}{\pi} \odot \sigma)$. However, his work is fundamentally incompatible with SL, because nesting is a static notion in the type system, and because logical and SL connectives such as negations, disjunctions, unrestricted existentials, and magic wands interpreted in the usual way would lead to unsoundness in his framework.

Combinability. As explained in Sect. 1.3, Le and Hobor [2018] handle combinability indirectly via preciseness. However, as explained in Sect. 3, preciseness is too restrictive and, for example, does not capture wildcard assertions. Brotherston et al. [2020] add labels and jump modalities to the assertion language, which solves the issue of combinability when it can be proven that the two fractions of a resource have the same origin. However, these additional features substantially complicate proofs in the logic. By contrast, our approach provides simple syntactic rules to prove that an assertion is combinable.

Restricted definitions of magic wands. In previous work [Dardinier et al. 2022b], we explore a restricted definition of the magic wand (in the bounded logic) that satisfies combinability, but not distributivity. Boyland [2010] defines a connective $\mathbin{-\!+}$, which is a syntactic connective similar to the magic wand, and which satisfies the analogous combinable property. However, the connective $\mathbin{-\!+}$ is much more restricted than a magic wand: e.g. $(a = b) \mathbin{-\!+} (b = a)$ cannot be proven. Chang and Rival [2008] also define a restricted version $A = \!\!* B$ of the magic wand, which is defined inductively and where A and B must be inductive predicates. Intuitively, $A = \!\!* B$ holds in a state σ if one can obtain A via a finite unfolding of predicate instances in B such that resources other than A obtained via the unfolding hold in σ. This restricted wand may satisfy combinability in general (although we have not proved this), but is not as expressive as the general magic wand supported by our work, in particular for expressing arbitrary method contracts.

Fixed points. Le and Hobor [2018] provide an induction principle for heaps with fractional permissions, based on the well-founded order of heaps that decrease by at least a fixed positive permission amount. This induction principle is strictly weaker than the one we present in Sect. 4, since the latter can deal, for example, with recursive predicate instances on the right-hand side of magic wands. Moreover, Sect. 5 shows how to leverage our induction principle to ensure combinability from a simple syntactic condition.

To give a semantics to abstract predicates, Parkinson and Bierman [2005] indirectly construct a semantic predicate environment from an abstract one, by generating a fixed point for a function *step*. As in Sect. 4, it turns out that this *step* function is monotonic but not Scott-continuous, and thus Kleene's fixed-point theorem cannot be applied.

Step-indexing [Appel and McAllester 2001] ensures the monotonicity of recursive definitions by guarding recursive calls with a *later* modality, which is useful for example to deal with recursive types [Ahmed 2006]. Step-indexing has been integrated into SL to reason about impredicative protocols [Svendsen and Birkedal 2014], and is at the core of Iris [Jung et al. 2018], a framework for higher-order concurrent SL. It would be interesting to explore how multiplication and the paradigm of unbounded logic presented here can be integrated into a framework such as Iris.

8 CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel semantics for separation logic, where states are temporarily unbounded. This logic reconciles the two existing views on multiplication: The syntactic one, used by automatic SL verifiers, and the semantic one, studied in theory. Our logic eliminates important shortcomings of the semantic multiplication: Distributivity holds for the magic wand, factorisability holds for the separating conjunction, and the wand $A \mathrel{-\!\!*} B$ is combinable if B is combinable. Moreover, our logic justifies the reasoning steps performed by existing automatic SL verifiers and paves the way to improve them further, for example by adding support for fractional wands.

As future work, we plan to extend Viper with support for fractional magic wands. We also plan to explore further existing discrepancies between the theory of SL and the way SL is automated in existing verifiers, in order to provide strong formal foundations for all features used in these verifiers. Formal foundations for automatic SL verifiers are crucial to increase trust in these verifiers, for example by generating certificates of correctness for successful runs.

DATA AVAILABILITY STATEMENT

All technical results presented in this paper have been formalised and proven in Isabelle/HOL, and our formalisation is publicly available [Dardinier 2022; Dardinier et al. 2022a].

ACKNOWLEDGMENTS

We thank Jérôme Dohrau for noticing an interesting connection with the problem of inferring annotations, and Gaurav Parthasarathy for numerous helpful discussions about this work. This work was partially funded by the Swiss National Science Foundation (SNSF) under Grant No. 197065.

REFERENCES

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proceedings of the 15th European Conference on Programming Languages and Systems* (Vienna, Austria) *(ESOP'06)*. Springer-Verlag, Berlin, Heidelberg, 69–83. https://doi.org/10.1007/11693024_6

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (sep 2001), 657–683. https://doi.org/10.1145/504709.504712

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification, In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). *Proc. ACM Program. Lang.* 3, OOPSLA, 147:1–147:30. https://doi.org/10.1145/3360573

Christian J. Bell, Andrew W. Appel, and David Walker. 2010. Concurrent Separation Logic for Pipelined Parallelization. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166.

Stefan Blom and Marieke Huisman. 2014. The VerCors Tool for Verification of Concurrent Programs. In *FM 2014: Formal Methods*, Cliff Jones, Pekka Pihlajasaari, and Jun Sun (Eds.). Springer International Publishing, Cham, 127–131.

Stefan Blom and Marieke Huisman. 2015. Witnessing the elimination of magic wands. *International Journal on Software Tools for Technology Transfer (STTT)* 17, 6 (2015), 757–781. https://doi.org/10.1007/s10009-015-0372-3

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *Principle of Programming Languages (POPL)*, Jens Palsberg and Martín Abadi (Eds.). ACM, 259–270.

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis (SAS)*, Radhia Cousot (Ed.). 55–72.

John Tang Boyland. 2010. Semantics of fractional permissions with nesting. *Transactions on Programming Languages and Systems (TOPLAS)* 32, 6 (2010), 22:1–22:33. https://doi.org/10.1145/1749608.1749611

James Brotherston, Diana Costa, Aquinas Hobor, and John Wickerson. 2020. Reasoning over Permissions Regions in Concurrent Separation Logic. In *Computer Aided Verification (CAV)*, Shuvendu K. Lahiri and Chao Wang (Eds.).

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local action and abstract separation logic. In *Logic in Computer Science (LICS)*. 366–375.

Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W. Appel. 2019. Proof Pearl: Magic Wand as Frame. arXiv:cs.PL/1909.08789

Bor-Yuh Evan Chang and Xavier Rival. 2008. Relational inductive shape analysis. *ACM SIGPLAN Notices* 43, 1 (2008), 247–260.

Patrick Cousot and Radhia Cousot. 1979. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific J. Math.* 81, 1 (1979), 43–57.

Thibault Dardinier. 2022. Unbounded Separation Logic. *Archive of Formal Proofs* (September 2022). https://isa-afp.org/entries/Separation_Logic_Unbounded.html, Formal proof development.

Thibault Dardinier, Peter Müller, and Alexander J. Summers. 2022a. Fractional Resources in Unbounded Separation Logic (artifact). https://doi.org/10.5281/zenodo.7072457

Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. 2022b. Sound Automation of Magic Wands. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 130–151.

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Programming Languages and Systems*, Zhenjiang Hu (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 161–177.

C. Haack and C. Hurlin. 2009. Resource Usage Protocols for Iterators. *Journal of Object Technology (JOT)* 8, 4 (June 2009), 55–83.

Aquinas Hobor and Cristian Gherghina. 2012. Barriers in Concurrent Separation Logic: Now With Tool Support! *Logical Methods in Computer Science* Volume 8, Issue 2 (April 2012). https://doi.org/10.2168/LMCS-8(2:2)2012

Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *Principles of Programming Languages (POPL)*. 271–282. https://doi.org/10.1145/1926385.1926417

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (NFM) (Lecture Notes in Computer Science)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.), Vol. 6617. Springer, 41–55.

Jonas Jensen, Lars Birkedal, and Peter Sestoft. 2011. Modular Verification of Linked Lists with Views via Separation Logic. *Journal of Object Technology (JOT)* 10 (January 2011), 2: 1–20. https://doi.org/10.1145/1924520.1924524

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Neelakantan R. Krishnaswami. 2006. Reasoning about Iterators with Separation Logic. In *Specification and Verification of Component-Based Systems (SAVCBS)*. https://doi.org/10.1145/1181195.1181213

Xuan-Bach Le and Aquinas Hobor. 2018. Logical Reasoning for Disjoint Permissions. In *European Symposium on Programming (ESOP)*, Amal Ahmed (Ed.).

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2009. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V (Lecture Notes in Computer Science)*, Vol. 5705. Springer, 195–222. http://www.springerlink.com

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-free Channels and Locks. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, A. D. Gordon (Ed.), Vol. 6012. Springer, 407–426. http://www.springerlink.com

Toshiyuki Maeda, Haruki Sato, and Akinori Yonezawa. 2011. Extended Alias Type System Using Separating Implication *(Workshop on Types in Language Design and Implementation (TLDI))*. https://doi.org/10.1145/1929553.1929559

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science)*, B. Jobstmann and K. R. M. Leino (Eds.), Vol. 9583. Springer, 41–62.

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic.* Vol. 2283. Springer Science & Business Media.

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. 2004. Separation and Information Hiding. *SIGPLAN Not.* 39, 1 (jan 2004), 268–280. https://doi.org/10.1145/982962.964024

Matthew Parkinson. 2005. *Local Reasoning for Java.* Ph.D. Dissertation. https://www.microsoft.com/en-us/research/publication/local-reasoning-for-java/ http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-654.html.

Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In *Principle of Programming Languages (POPL)*, J. Palsberg and M. Abadi (Eds.). ACM, 247–258.

Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs, Vol. 9032. 158–182. https://doi.org/10.1007/978-3-662-46669-8_7

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. IEEE, 55–74.

Malte Schwerhoff and Alexander J. Summers. 2015. Lightweight Support for Magic Wands in an Automatic Verifier. In *European Conference on Object-Oriented Programming (ECOOP) (LIPIcs)*, J. T. Boyland (Ed.), Vol. 37. Schloss Dagstuhl, 614–638.

Alexander J. Summers and Peter Müller. 2018. Automating Deductive Verification for Weak-Memory Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science)*. Springer, 190–209.

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–168.

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285 – 309. https://doi.org/pjm/1103044538

Thomas Tuerk. 2010. Local reasoning about while-loops. In *Verified Software: Theories, Tools and Experiments - Theory Workshop (VS-Theory)*.

Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. *Electronic Notes in Theoretical Computer Science* 276 (2011), 335–351. https://doi.org/10.1016/j.entcs.2011.09.029 Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

Jules Villard, Étienne Lozes, and Cristiano Calcagno. 2009. Proving Copyless Message Passing. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems* (Seoul, Korea) *(APLAS '09)*. Springer-Verlag, Berlin, Heidelberg, 194–209. https://doi.org/10.1007/978-3-642-10672-9_15