Temporary Read-Only Permissions for Separation Logic

Arthur Charguéraud[1,2] and François Pottier[1]

[1] Inria, Paris, France*
[2] ICube – CNRS, Université de Strasbourg, France

Abstract. We present an extension of Separation Logic with a general mechanism for temporarily converting any assertion (or "permission") to a read-only form. No accounting is required: our read-only permissions can be freely duplicated and discarded. We argue that, in circumstances where mutable data structures are temporarily accessed only for reading, our read-only permissions enable more concise specifications and proofs. The metatheory of our proposal is verified in Coq.

1 Introduction

Separation Logic [30] offers a natural and effective framework for proving the correctness of imperative programs that manipulate the heap. It is exploited in many implemented program verification systems, ranging from fully automated systems, such as Infer [9], through semi-interactive systems, such as Smallfoot [4], jStar [15], and VeriFast [22], to fully interactive systems (embedded within a proof assistant), such as the Verified Software Toolchain [1] and Charge! [3], to cite just a few. The CFML system, developed by the first author [10,11], can be viewed as a member of the latter category. We have used it to verify many sequential data structures and algorithms, representing several thousand lines of OCaml code.

1.1 Redundancy in specifications

Our experience with Separation Logic at scale in CFML leads us to observe that many specifications suffer from a somewhat unpleasant degree of verbosity, which results from a frequent need to repeat part of the precondition in the postcondition. This repetition is evident already in the Separation Logic axiom for dereferencing a pointer:

TRADITIONAL READ AXIOM
$$\{l \hookrightarrow v\}\ (\mathsf{get}\, l)\ \{\lambda y.\ [y = v] \star l \hookrightarrow v\}$$

This axiom states that "if initially the memory location l stores the value v, then dereferencing l yields the value v *and, after this operation,* l *still stores* v."

* This research was partly supported by the French National Research Agency (ANR) under the grant ANR-15-CE25-0008.

Arguably, to a human reader, the last part of this statement may seem obvious, even though it is not formally redundant.

Beginning with this axiom, this redundancy contaminates the entire system. It arises not only when a single memory cell is read, but, more generally, every time a data structure is accessed for reading. To illustrate this, consider a function *array_concat* that expects (pointers to) two arrays a_1 and a_2 and returns (a pointer to) a new array a_3 whose content is the concatenation of the contents of a_1 and a_2. Its specification in Separation Logic would be as follows:

$$\begin{array}{l}\{a_1 \rightsquigarrow \mathsf{Array}\, L_1 \mathbin{\star} a_2 \rightsquigarrow \mathsf{Array}\, L_2\} \\ (\mathit{array\_concat}\, a_1\, a_2) \\ \{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\, (L_1 +\!\!+ L_2) \mathbin{\star} a_1 \rightsquigarrow \mathsf{Array}\, L_1 \mathbin{\star} a_2 \rightsquigarrow \mathsf{Array}\, L_2\}\end{array} \tag{1}$$

We assume that $a \rightsquigarrow \mathsf{Array}\, L$ asserts the existence (and unique ownership) of an array at address a whose content is given by the list L. A separating conjunction $\star$ is used in the precondition to require that a_1 and a_2 be disjoint arrays. Its use in the postcondition guarantees that a_3 is disjoint with a_1 and a_2. In this specification, again, the fact that the arrays a_1 and a_2 are unaffected must be explicitly stated as part of the postcondition, making the specification seem verbose.

Ideally, we would like to write a more succinct specification, which directly expresses the idea that the arrays a_1 and a_2 are only read by *array_concat*, even though they are mutable arrays. Such a specification could be as follows, where "RO" is a read-only modality, whose exact meaning remains to be explained:

$$\begin{array}{l}\{\mathsf{RO}(a_1 \rightsquigarrow \mathsf{Array}\, L_1) \mathbin{\star} \mathsf{RO}(a_2 \rightsquigarrow \mathsf{Array}\, L_2)\} \\ (\mathit{array\_concat}\, a_1\, a_2) \\ \{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\, (L_1 +\!\!+ L_2)\}\end{array} \tag{2}$$

The idea is, because only read access to a_1 and a_2 is granted, these arrays cannot be modified or deallocated by the call $\mathit{array\_concat}\, a_1\, a_2$. Therefore, the postcondition need not say anything about these arrays: that would be redundant.

1.2 Can "RO" be interpreted by macro-expansion?

At this point, the reader may wonder whether the meaning of "RO" could be explained by a simple macro-expansion process. That is, assuming that "RO" is allowed to appear only at the top level of the precondition, the Hoare triple $\{\mathsf{RO}(H_1) \star H_2\}\ t\ \{Q\}$ could be viewed as syntactic sugar for $\{H_1 \star H_2\}\ t\ \{H_1 \star Q\}$. Such sugar is easy to implement; in fact, CFML offers it, under the notation "INV", for "invariant". However, this naïve interpretation suffers from several shortcomings, which can be summarized as follows:

1. It reduces apparent redundancy in specifications,
   but does not eliminate the corresponding redundancy in proofs.
2. It does not allow read-only state to be aliased.
3. It leads to deceptively weak specifications.
4. It can lead to unusably weak specifications.

In the following, we expand on each of these points.

Shortcoming 1: does not reduce proof effort Under the naïve interpretation of "RO" as macro-expansion, the Hoare triple $\{\mathsf{RO}(H_1) \star H_2\}\ t\ \{Q\}$ is just syntactic sugar. The presence or absence of this sugar has no effect on the proof obligations that the user must fulfill. Even though the sugar hides the presence of the conjunct H_1 in the postcondition, it really is still there. So, the user must prove that H_1 holds upon termination of the command t. This might take several proof steps: for example, several predicate definitions might need to be folded. In other words, the issue that we would like to address is not just undesired verbosity; it is also undesired work.

In this paper, we intend to give direct semantic meaning to "RO". In this approach, $\{\mathsf{RO}(H_1) \star H_2\}\ t\ \{Q\}$ is an ordinary Hoare triple, whose postcondition does not mention H_1. Thus, there is no need for the user to argue that H_1 holds upon termination. The proof effort is therefore reduced.

Shortcoming 2: does not allow aliasing read-only state Under the naïve interpretation of "RO", our proposed specification of *array_concat* (2) is just sugar for the obvious specification (1), therefore means that *array_concat* must be applied to two disjoint arrays. Yet, in reality, a call of the form "*array_concat a a*" is safe and makes sense. To allow it, one could prove another specification for *array_concat*, dealing specifically with the case where an array is concatenated with itself:

$$\begin{array}{l}\{a \rightsquigarrow \mathsf{Array}\, L\} \\ (\mathit{array\_concat}\ a\ a) \\ \{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\,(L +\!\!+ L) \star a \rightsquigarrow \mathsf{Array}\, L\}\end{array} \tag{3}$$

However, that would imply extra work: firstly, when *array_concat* is defined, as its code must be verified twice; secondly, when it is invoked, as the user may need to indicate which of the two specifications (1) and (3) should be used.

In this paper, we define the meaning of "RO" in such a way that every read-only assertion is duplicable: that is, $\mathsf{RO}(H)$ entails $\mathsf{RO}(H) \star \mathsf{RO}(H)$. Thanks to this property, our proposed specification of *array_concat* (2) allows justifying the call "*array_concat a a*". In fact, under our reasoning rules, specification (2) subsumes both of the specifications (1) and (3) that one would need in the absence of read-only assertions.

Shortcoming 3: deceptively weak Under the naïve interpretation of "RO" as macro-expansion, the Hoare triple $\{\mathsf{RO}(H_1) \star H_2\}\ t\ \{Q\}$ does not guarantee that the memory covered by H_1 is unaffected by the execution of the command t. Instead, it means only that H_1 still holds upon termination of t. To see this, imagine that the assertion $h \rightsquigarrow \mathsf{HashTable}\, M$ means "the hash table at address h currently represents the dictionary M". A function *population*, which returns the number of entries in a hash table, could have the following specification:

$$\{\mathsf{RO}(h \rightsquigarrow \mathsf{HashTable}\, M)\}\ (\mathit{population}\ h)\ \{\lambda y.\ [y = \mathsf{card}\, M]\} \tag{4}$$

Under the macro-expansion interpretation, this specification guarantees that $h \rightsquigarrow \mathsf{HashTable}\, M$ is preserved, so, after a call to *population*, the table h still

represents the dictionary M. Somewhat subtly, this does not guarantee that the concrete data structure is unchanged. In fact, a function *resize*, which doubles the physical size of the table and profoundly affects its organization in memory, would admit a similar specification:

$$\{\mathsf{RO}(h \rightsquigarrow \mathsf{HashTable}\, M)\}\ (\mathit{resize}\ h)\ \{\lambda().\ []\} \tag{5}$$

In this paper, we define the meaning of "RO" in such a way that it really means "read-only". Therefore, the above specification of *population* (4) acquires stronger meaning, and guarantees that *population* does not modify the hash table. The specification of *resize* (5) similarly acquires stronger meaning, and can no longer be established, since *resize* does modify the hash table. A valid specification of *resize* is $\{h \rightsquigarrow \mathsf{HashTable}\, M\}\ (\mathit{resize}\ h)\ \{\lambda().\ h \rightsquigarrow \mathsf{HashTable}\, M\}$.

Shortcoming 4: unusably weak The weakness of the above specifications is not only somewhat unexpected and deceptive: there are in fact situations where it is problematic.

Imagine that a hash table is internally represented as a record of several fields, among which is a *data* field, holding a pointer to an array. The abstract predicate $h \rightsquigarrow \mathsf{HashTable}\, M$ might then be defined as follows:

$$\begin{array}{l} h \rightsquigarrow \mathsf{HashTable}\, M := \\ \quad \exists a.\, \exists L.\, (h \rightsquigarrow \{\mathit{data} = a; \ldots\} \star a \rightsquigarrow \mathsf{Array}\, L \star \ldots) \end{array} \tag{6}$$

Suppose we wish to verify an operation *foo*, inside the "hash table" module, whose code begins as follows:

```
let foo h =
  let d = h.data in          – read the address of the array
  let p = population h in    – call population
  ...
```

A proof outline for this function must begin as follows:

```
1  let foo h =
2    {h ⇝ HashTable M}                                  – foo's precondition
3    {h ⇝ {data = a; ...} ⋆ a ⇝ Array L ⋆ ...}          – by unfolding
4    let d = h.data in
5    {h ⇝ {data = a; ...} ⋆ a ⇝ Array L ⋆ ... ⋆ [d = a]} – by reading
6    {h ⇝ HashTable M ⋆ [d = a]}                        – by folding
7    let p = population h in
8    {h ⇝ HashTable M ⋆ [d = a] ⋆ [p = #M]}
9    ...
```

At line 3, we unfold $h \rightsquigarrow \mathsf{HashTable}\, M$. Two auxiliary variables, a and L, are introduced at this point; their scope extends to the end of the proof outline. This unfolding step is mandatory: indeed, the read instruction at line 4 requires $h \rightsquigarrow \{\mathit{data} = a; \ldots\}$. This instruction produces the pure assertion $[d = a]$, which

together with the assertion $h \rightsquigarrow \{data = a; \ldots\}$ means that d is the current value of the field $h.data$.

At line 6, we fold $h \rightsquigarrow \mathsf{HashTable}\, M$. This is mandatory: indeed, under the naïve interpretation of "RO", the precondition of the call "*population* h" is $h \rightsquigarrow \mathsf{HashTable}\, M$. Unfortunately, this folding step is harmful: it causes us to lose $h \rightsquigarrow \{data = a; \ldots\}$ and thereby to forget that d is the current value of the field $h.data$. (The equation $d = a$ remains true, but becomes useless.) Yet, in reality, this fact is preserved through the call, which does not modify the hash table.

In summary, because the specification of *population* (4) is too weak, calling *population* at line 7 causes us to lose the benefit of the read instruction at line 4. In this particular example, one could work around the problem by exchanging the two instructions. In general, though, it might not be possible or desirable to modify the code so as to facilitate the proof. Another work-around is to equip *population* with a lower-level specification, where the predicate HashTable is manually unfolded.

We have demonstrated that, under the naïve interpretation, the specification of *population* (4) can be unsuitable for use inside the "hash table" abstraction.

In this paper, we define the meaning of "RO" in such a way that all of the information that is available at line 5 is preserved through the call to *population*. This is explained later on (§2.5).

1.3 Towards true read-only permissions

The question that we wish to address is: what is a simple extension of sequential Separation Logic with duplicable temporary read-only permissions[3] for mutable data?

We should stress that we are primarily interested in a logic of sequential programs. We do discuss structured parallelism and shared-memory concurrency near the end of the paper (§6.2).

We should also emphasize that we are not interested in read permissions for permanently immutable data, which are a different concept. Such permissions can be found, for instance, in Mezzo [2], and could be introduced in Separation Logic, if desired. They, too, grant read access only, and are duplicable. Mezzo allows converting a unique read-write permission to a duplicable read permission, but not the other way around: the transition from mutable to immutable state is irrevocable. Mezzo has no mechanism for obtaining a temporary read-only view of a mutable data structure.

Finally, we should say a word of fractional permissions (which are discussed in greater depth in §5.4). Fractional permissions [6] can be used to obtain temporary read-only views of mutable data. A fraction that is strictly less than 1 grants read-only access, and, by joining all shares so as to recover the fraction 1, a unique read-write access permission can be recovered. Nevertheless, fractional permissions are not what we seek. They do not address our shortcoming 1: their

[3] Following Boyland [6], Balabonski *et al.* [2], and others, we use the words "assertion" and "permission" interchangeably.

$$
\begin{array}{rcll}
\mathsf{RO}([P]) & = & [P] & \\
\mathsf{RO}(H_1 \star H_2) & \rhd & \mathsf{RO}(H_1) \star \mathsf{RO}(H_2) & \text{(the reverse is false)} \\
\mathsf{RO}(H_1 \curlyvee H_2) & = & \mathsf{RO}(H_1) \curlyvee \mathsf{RO}(H_2) & \\
\mathsf{RO}(\exists x.\, H) & = & \exists x.\, \mathsf{RO}(H) & \\
\mathsf{RO}(\mathsf{RO}(H)) & = & \mathsf{RO}(H) & \\
\mathsf{RO}(H) & \rhd & \mathsf{RO}(H') & \text{if } H \rhd H' \\
\mathsf{RO}(H) & = & \mathsf{RO}(H) \star \mathsf{RO}(H) &
\end{array}
$$

Fig. 1. Properties of RO

use requires work that we wish to avoid, namely "accounting" (arithmetic reasoning) as well as (universal and existential) quantification over fraction variables. Furthermore, whereas $\mathsf{RO}(h \rightsquigarrow \mathsf{HashTable}\, M)$ is a well-formed permission in our logic, in most systems of fractional permissions, $\frac{1}{2}(h \rightsquigarrow \mathsf{HashTable}\, M)$ is not well-formed. The systems that do allow this kind of "scaling", such as Boyland's [7], do so at the cost of restricting disjunction and existential quantification so that they are "precise".

In this paper, we answer the above question. We introduce a generic assertion transformer, "RO". For any assertion H, it is permitted to form the assertion $\mathsf{RO}(H)$, which offers read-only access to the memory covered by H. For instance, $\mathsf{RO}(x \hookrightarrow v)$ offers read-only access to the memory cell at address x. The temporary conversion from a permission H to its read-only counterpart $\mathsf{RO}(H)$ is performed within a lexically-delimited scope, via a "read-only frame rule". Upon entry into the scope, H is replaced with $\mathsf{RO}(H)$. Within the scope, $\mathsf{RO}(H)$ can be duplicated if desired, and some copies can be discarded; there is no need to keep track of all shares and recombine them so as to regain a full permission. Upon exit of the scope, the permission H re-appears.

The rest of the paper is organized as follows. First, we review our additions to Separation Logic (§2). Then, we give a formal, self-contained presentation of our logic (§3) and of its model, which we use to establish the soundness of the logic (§4). The soundness proof is formalized in Coq and can be found online [12]. Then, we review some of the related work (§5), discuss some potential applications and extensions of our logic (§6), and conclude (§7).

2 Overview

In this section, we describe our additions to Separation Logic, with which we assume a basic level of familiarity. The following sections describe our logic (§3) and its model (§4) in full, and may serve as a reference.

2.1 A "read-only" modality

To begin with, we introduce read-only permissions in the syntax of permissions. Informally, the permission $\mathsf{RO}(H)$ controls the same heap fragment as the permission H, but can be used only for reading. A more precise understanding of the meaning of "RO" is given by the semantic model (§4.1).

The "RO" modality enjoys several important properties, shown in Figure 1, where the symbol $\rhd$ denotes entailment. When applied to a pure assertion $[P]$, "RO" vanishes. It can be pushed into a separating conjunction: $\mathsf{RO}(H_1 \star H_2)$ entails $\mathsf{RO}(H_1) \star \mathsf{RO}(H_2)$. The reverse entailment is false[4]. Because of this, one might worry that exploiting the entailment $\mathsf{RO}(H_1 \star H_2) \rhd \mathsf{RO}(H_1) \star \mathsf{RO}(H_2)$ causes a loss of information. This is true, but if that is a problem, then one can exploit the equality $\mathsf{RO}(H_1 \star H_2) = \mathsf{RO}(H_1 \star H_2) \star \mathsf{RO}(H_1) \star \mathsf{RO}(H_2)$ instead. "RO" commutes with disjunction and existential quantification. "RO" is idempotent[5]. A read-only permission for a single cell of memory, $\mathsf{RO}(l \hookrightarrow v)$, cannot be rewritten into a simpler form, which is why there is no equation for it in Figure 1. It can be exploited via the new read axiom (§2.4). The last two lines of Figure 1 respectively state that "RO" is covariant and that read-only permissions are duplicable.

Together, these rules allow pushing "RO" into composite permissions. For instance, if $h \rightsquigarrow \mathsf{HashTable}\, M$ is defined as before (§1, (6)), then the read-only permission $\mathsf{RO}(h \rightsquigarrow \mathsf{HashTable}\, M)$ entails:

$$\exists a.\, \exists L.\, (\mathsf{RO}(h \rightsquigarrow \{data = a; \ldots\}) \star \mathsf{RO}(a \rightsquigarrow \mathsf{Array}\, L) \star \mathsf{RO}(\ldots))$$

In other words, "read-only access to a hash table" implies read-only access to the component objects of the table, as expected.

2.2 A read-only frame rule

To guarantee the soundness of our extension of Separation Logic with read-only permissions, we must enforce one key metatheoretical invariant, namely: a memory location is never governed at the same time by a read-write permission and by a read permission. Indeed, if two such permissions were allowed to coexist, the read-write permission by itself would allow writing a new value to this memory location. The read permission would not be updated (it could be framed out, hence invisible, during the write) and would therefore become stale (that is, carry out-of-date information about the value stored at this location). That would be unsound.

In order to forbid the co-existence of read-write and read-only permissions for a single location, we propose enforcing the following informal rules:

1. Read-only permissions obey a lexical scope (or "block") discipline.
2. Upon entry into a block, a permission H can be replaced with its read-only counterpart $\mathsf{RO}(H)$.
3. Upon exit of this block, the permission H reappears.
4. No read-only permissions are allowed to exit the block.

[4] If it were true, then we would have the following chain of equalities: $\mathsf{RO}(l \hookrightarrow v) = \mathsf{RO}(l \hookrightarrow v) \star \mathsf{RO}(l \hookrightarrow v) = \mathsf{RO}(l \hookrightarrow v \star l \hookrightarrow v) = \mathsf{RO}([\mathsf{False}]) = [\mathsf{False}]$.

[5] In practice, this property should not be useful, because permissions of the form $\mathsf{RO}(\mathsf{RO}(H))$ never appear: the read-only frame rule (§2.2) is formulated in such a way that it cannot give rise to such a permission.

$$\mathsf{normal}\,[P] \qquad \mathsf{normal}\,(x \hookrightarrow v) \qquad \frac{\mathsf{normal}\,H_1 \qquad \mathsf{normal}\,H_2}{\mathsf{normal}\,(H_1 \star H_2)}$$

$$\frac{\mathsf{normal}\,H_1 \qquad \mathsf{normal}\,H_2}{\mathsf{normal}\,(H_1 \curlyvee H_2)} \qquad \frac{\forall x.\ \mathsf{normal}\,H}{\mathsf{normal}\,(\exists x.\ H)}$$

Fig. 2. Properties of normal

Roughly speaking, there is no danger of co-existence between read-write and read permissions when a block is entered, because H is removed at the same time $\mathsf{RO}(H)$ is introduced. There is no danger either when this block is exited, even though H reappears, because no read-only permissions are allowed to exit.

Technically, all four informal rules above take the form of a single reasoning rule, the "read-only frame rule", which subsumes the frame rule of Separation Logic. The frame rule allows an assertion H' to become hidden inside a block: H' disappears upon entry, and reappears upon exit of the block. The read-only frame rule does this as well, and in addition, makes the read-only permission $\mathsf{RO}(H')$ available within the block. In our system, the two rules are as follows:

FRAME RULE
$$\frac{\{H\}\ t\ \{Q\} \qquad \mathsf{normal}\,H'}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

READ-ONLY FRAME RULE
$$\frac{\{H \star \mathsf{RO}(H')\}\ t\ \{Q\} \qquad \mathsf{normal}\,H'}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

The frame rule (above, left) is in fact just a special case of the read-only frame rule (above, right). Indeed, a read-only permission can be discarded at any time (using rule DISCARD-PRE from Figure 4). Thus, the frame rule can be derived by applying the read-only frame rule and immediately discarding $\mathsf{RO}(H')$.

Both rules above have the side condition $\mathsf{normal}\,H'$, which requires H' to be "normal". Its role is to ensure that "no read-only permissions are allowed to exit the block". "Normality" can be understood in several ways:

1. A syntactic understanding is that a permission is "normal" if "RO" does not occur in it. This view is supported by the rules in Figure 2. The one thing to remark about these rules is that there is no rule whose conclusion is $\mathsf{normal}(\mathsf{RO}(H))$.
2. A semantic understanding is given when we set up a model of our logic (§4.1). There, we define what it means for a heap, where each memory location is marked either as read-write or as read-only, to satisfy a permission. Then, a permission is "normal" if a heap that satisfies it cannot contain any read-only memory locations.

Because of the normality condition in the read-only frame rule, a Hoare triple $\{H\}\ t\ \{Q\}$ typically[6] has a normal postcondition Q. This means that read-only

[6] If we restricted the rule of CONSEQUENCE in Figure 4 by adding the side condition $\mathsf{normal}\,Q$, then we would be able to prove that every triple $\{H\}\ t\ \{Q\}$ that can

permissions can only be "passed down", from caller to callee. They cannot be "passed back up", from callee to caller.

2.3 A framed sequencing rule

The sequencing rule of Separation Logic (below, left) remains sound in our logic. However, in a setting where postconditions must be normal (or are typically normal), this rule is weaker than desired: it does not allow read-only permissions to be distributed into its second premise. Indeed, suppose Q' is normal, as it is the postcondition of the first premise. Unfortunately, Q' is also the precondition of the second premise. This means that no read-only permissions are available in the proof of t_2. Yet, in practice, it is useful and desirable to be able to thread one or more read-only permissions through a sequence of instructions. We remedy the problem by giving a slightly generalized sequencing rule (below, right), which allows an arbitrary permission H' to be framed out of t_1 and therefore passed directly to t_2.

TRADITIONAL SEQUENCING RULE

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \{Q'\}\ t_2\ \{Q\}}{\{H\}\ (t_1\,;\,t_2)\ \{Q\}}$$

FRAMED SEQUENCING RULE

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \{Q' \star H'\}\ t_2\ \{Q\}}{\{H \star H'\}\ (t_1\,;\,t_2)\ \{Q\}}$$

In Separation Logic, the "framed sequencing rule" can be derived from the sequencing rule and the frame rule. Here, conversely, it is viewed as a primitive reasoning rule; the traditional sequencing rule can be derived from it, if desired.

When a read-only permission is available at the beginning of a sequence of instructions, it is in fact available to every instruction in the sequence. This is expressed by the following rule, which can be derived from the framed sequencing rule, the rule of consequence, and from the fact that read-only permissions are duplicable.

READ-ONLY SEQUENCING RULE

$$\frac{\{H \star \mathsf{RO}(H')\}\ t_1\ \{Q'\} \qquad \{Q' \star \mathsf{RO}(H')\}\ t_2\ \{Q\}}{\{H \star \mathsf{RO}(H')\}\ (t_1\,;\,t_2)\ \{Q\}}$$

2.4 A new read axiom

The axiom for reading a memory cell must be generalized so as to accept a read-only permission (instead of a read-write permission) as proof that reading is permitted. The traditional axiom (below, left) requires a read-write permission $l \hookrightarrow v$, which it returns. The new axiom (below, right) requires a read-only permission $\mathsf{RO}(l \hookrightarrow v)$, which it discards.

TRADITIONAL READ AXIOM

$$\{l \hookrightarrow v\}\ (\mathsf{get}\, l)\ \{\lambda y.\ [y = v] \star l \hookrightarrow v\}$$

NEW READ AXIOM

$$\{\mathsf{RO}(l \hookrightarrow v)\}\ (\mathsf{get}\, l)\ \{\lambda y.\ [y = v]\}$$

The traditional read axiom remains sound, and can in fact be derived from the new read axiom and the read-only frame rule.

be established using the reasoning rules has a normal postcondition Q. We did not restrict the rule of consequence in this way because this is technically not necessary.

2.5 Illustration

Recall the hypothetical operation *foo* that was used earlier (§1.2) to illustrate our claim that faking read-only permissions by macro-expansion is unsatisfactory. Let us carry out this proof again, this time with our read-only permissions. As before, we assume that *population* requires read access to the hash table. This is expressed by specification (4), which we repeat here:

$$\{\mathsf{RO}(h \rightsquigarrow \mathsf{HashTable}\, M)\}\ (\mathit{population}\ h)\ \{\lambda y.\ [y = \#M]\}$$

We will use the following derived rule, which follows from the read-only frame rule, the rule of consequence, and the fact that "RO" is covariant:

$$\text{READ-ONLY FRAME RULE (WITH CONSEQUENCE)}\quad \frac{H' \rhd H'' \qquad \{H \star \mathsf{RO}(H'')\}\ t\ \{Q\} \qquad \mathsf{normal}\ H'}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

This rule differs from the read-only frame rule in that, instead of introducing $\mathsf{RO}(H')$, this rule introduces $\mathsf{RO}(H'')$, where H'' is logically weaker than H'. Nevertheless, upon exit, the permission H' is recovered.

We can now give a new proof outline for *foo*:

1 $\mathsf{let}\ \mathit{foo}\ h =$
2 $\quad \{h \rightsquigarrow \mathsf{HashTable}\, M\}$
3 $\quad \{h \rightsquigarrow \{\mathit{data} = a; \ldots\} \star a \rightsquigarrow \mathsf{Array}\, L \star \ldots\}$
4 $\quad \mathsf{let}\ d = h.\mathit{data}\ \mathsf{in}$
5 $\quad \{h \rightsquigarrow \{\mathit{data} = a; \ldots\} \star a \rightsquigarrow \mathsf{Array}\, L \star \ldots \star [d = a]\}$
6 $\quad \{\mathsf{RO}(h \rightsquigarrow \mathsf{HashTable}\, M) \star [d = a]\}$
$\quad$ – by the read-only frame rule (with consequence)
7 $\quad \mathsf{let}\ p = \mathit{population}\ h\ \mathsf{in}$
8 $\quad \{h \rightsquigarrow \{\mathit{data} = a; \ldots\} \star a \rightsquigarrow \mathsf{Array}\, L \star \ldots \star [d = a] \star [p = \#M]\}$
9 $\quad \ldots$

Up to and including line 5, the outline is the same as in our previous attempt. At this point, in order to justify the call to *population*, we wish to obtain the read-only permission $\mathsf{RO}(h \rightsquigarrow \mathsf{HashTable}\, M)$. This is done by applying the read-only frame rule (with consequence) around the call. We exploit the entailment $(h \rightsquigarrow \{\mathit{data} = a; \ldots\} \star a \rightsquigarrow \mathsf{Array}\, L \star \ldots) \rhd (h \rightsquigarrow \mathsf{HashTable}\, M)$, which corresponds to folding the HashTable predicate. Thus, for the duration of the call, we obtain the read-only permission $\mathsf{RO}(h \rightsquigarrow \mathsf{HashTable}\, M)$. After the call, the more precise, unfolded, read-write permission $h \rightsquigarrow \{\mathit{data} = a; \ldots\} \star a \rightsquigarrow \mathsf{Array}\, L \star \ldots$ reappears. The loss of information in our first proof outline (§1.2) no longer occurs here.

3 Logic

In this section, we give a formal presentation of Separation Logic with read-only permissions. We present the syntax of programs, the syntax of assertions, and the reasoning rules. This is all a user of the logic needs to know. The semantic model and the proof of soundness come in the next section (§4).

$$\textsc{eval-val}\quad v/m \Downarrow v/m$$

$$\textsc{eval-if}\quad \frac{n \neq 0 \ \wedge\ t_1/m \Downarrow v'/m' \quad \vee \quad n = 0 \ \wedge\ t_2/m \Downarrow v'/m'}{(\mathsf{if}\ n\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/m \Downarrow v'/m'}$$

$$\textsc{eval-let}\quad \frac{t_1/m \Downarrow v_1/m' \quad ([v_1/x]\,t_2)/m' \Downarrow v/m''}{(\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2)/m \Downarrow v/m''}$$

$$\textsc{eval-app}\quad \frac{v_1 = \mu f.\lambda x.t \quad ([v_1/f]\,[v_2/x]\,t)/m \Downarrow v'/m'}{(v_1\,v_2)/m \Downarrow v'/m'}$$

$$\textsc{eval-ref}\quad \frac{l \notin \mathsf{dom}\, m \qquad m' = m \uplus (l \mapsto v)}{(\mathsf{ref}\, v)/m \Downarrow l/m'}$$

$$\textsc{eval-get}\quad \frac{l \in \mathsf{dom}\, m \qquad v = m[l]}{(\mathsf{get}\, l)/m \Downarrow v/m}$$

$$\textsc{eval-set}\quad \frac{l \in \mathsf{dom}\, m \qquad m' = m[l := v]}{(\mathsf{set}\, l\, v)/m \Downarrow ()/m'}$$

Fig. 3. Big-step evaluation

3.1 Calculus

Our programming language is a λ-calculus with references. Its syntax is as follows:

$$v := x \mid () \mid n \mid l \mid \mu f.\lambda x.t$$
$$t := v \mid \mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \mid \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \mid (v\,v) \mid \mathsf{ref}\, v \mid \mathsf{get}\, v \mid \mathsf{set}\, v\, v$$

A value v is either a variable x, the unit value $()$, an integer constant n, a memory location l, or a recursive function $\mu f.\lambda x.t$. A term t is either a value, a conditional construct, a sequencing construct, a function call, or a primitive instruction for allocating, reading, or writing a reference.

The big-step evaluation judgement (Figure 3) takes the form $t/m \Downarrow v/m'$, and asserts that the evaluation of the term t in the memory m terminates and produces the value v in the memory m'. A memory is a finite map of locations to values.

3.2 Permissions

The syntax of permissions, also known as assertions, is as follows:

$$H := [P] \mid l \hookrightarrow v \mid H_1 \star H_2 \mid H_1 \curlyvee H_2 \mid \exists x.\, H \mid \mathsf{RO}(H)$$

All of these constructs are standard, except for $\mathsf{RO}(H)$, which represents a read-only form of the permission H. The pure assertion $[P]$ is true of an empty heap, provided the proposition P holds. P is expressed in the metalanguage; in our Coq formalisation, it is an arbitrary proposition of type Prop. In particular, a Hoare triple $\{H\}\ t\ \{Q\}$ (defined later on) is a proposition: this is important, as it allows reasoning about first-class functions. The empty permission $[\,]$ can be viewed as syntactic sugar for $[\mathsf{True}]$. The permission $l \hookrightarrow v$ grants unique read-write access to the reference cell at address l, and asserts that this cell currently

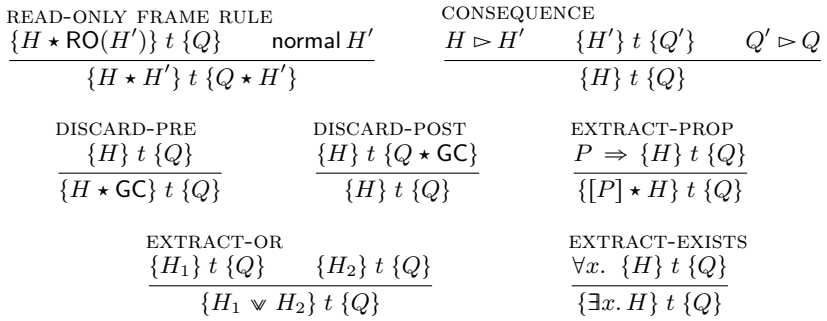

Fig. 4. Reasoning rules (structural)

contains the value v. Separating conjunction $\star$, disjunction $\curlyvee$, and existential quantification are standard. We omit ordinary conjunction $\curlywedge$, partly because we do not use it in practice when carrying out proofs in CFML, partly because we did not have time to study whether the rule of conjunction holds in our logic.

From a syntactic standpoint, a "normal" permission is one that does not contain any occurrences of "RO". (Recall Figure 2.)

As usual in Separation Logic, permissions are equipped with an entailment relation, written $H_1 \rhd H_2$ ("H_1 entails H_2"). It is a partial order. (In particular, it is antisymmetric: we view two propositions that entail each other as equal.) The standard connectives of Separation Logic enjoy their usual properties, which, for the sake of brevity, we do not repeat. (For instance, separating conjunction is associative, commutative, and admits $[\,]$ as a unit.) In addition, read-only permissions satisfy the laws of Figure 1, which have been explained earlier (§2.1).

3.3 Reasoning rules

As usual in Separation Logic, a Hoare triple takes the form $\{H\}\ t\ \{\lambda y.\, H'\}$ where H and H' are permissions and t is a term. The precondition H expresses requirements on the initial state; the postcondition $\lambda y.\, H'$ offers guarantees about the result y of the computation and about the final state. We write Q for a postcondition $\lambda y.\, H'$. For greater readability, we write $Q \star H'$ for $\lambda y.\, (Q\, y \star H')$. We write $Q \curlyvee Q'$ as a shorthand for $\lambda y.\, (Q\, y) \curlyvee (Q'\, y)$. We write $Q \rhd Q'$ as a shorthand for $\forall y.\, (Q\, y) \rhd (Q'\, y)$.

We adopt a total correctness interpretation, whereby a triple $\{H\}\ t\ \{\lambda y.\, H'\}$ guarantees that (under the precondition H) the evaluation of t terminates. This is arbitrary: our read-only permissions would work equally well in a partial correctness setting.

The reasoning rules, by which Hoare triples can be established, are divided in two groups: structural (or non-syntax-directed) rules (Figure 4) and syntax-directed rules (Figure 5).

$$\textsc{val}\quad \{[\,]\}\ v\ \{\lambda y.\ [y = v]\}$$

$$\textsc{if}\quad \frac{n \neq 0 \Rightarrow \{H\}\ t_1\ \{Q\} \qquad n = 0 \Rightarrow \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{if}\ n\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\ \{Q\}}$$

$$\textsc{framed sequencing rule (let)}\quad \frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\ x \star H'\}\ t_2\ \{Q\}}{\{H \star H'\}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}$$

$$\textsc{app}\quad \frac{v_1 = \mu f.\lambda x.t \qquad \{H\}\ ([v_1/f]\,[v_2/x]\,t)\ \{Q\}}{\{H\}\ (v_1\ v_2)\ \{Q\}}$$

$$\textsc{ref}\quad \{[\,]\}\ (\mathsf{ref}\, v)\ \{\lambda y.\ \exists l.\ [y = l] \star l \hookrightarrow v\}$$

$$\textsc{new read axiom (get)}\quad \{\mathsf{RO}(l \hookrightarrow v)\}\ (\mathsf{get}\, l)\ \{\lambda y.\ [y = v]\}$$

$$\textsc{set}\quad \{l \hookrightarrow v'\}\ (\mathsf{set}\, l\, v)\ \{\lambda y.\ l \hookrightarrow v\}$$

Fig. 5. Reasoning rules (syntax-directed)

Among the structural rules, the only nonstandard rule is the READ-ONLY FRAME RULE, which has been explained earlier (§2.2). The rule of CONSEQUENCE allows exploiting entailment to strengthen the precondition and weaken the postcondition. The rules DISCARD-PRE and DISCARD-POST allow discarding part of the pre- or postcondition. (The rule of consequence cannot be used for this purpose: for instance, $x \hookrightarrow v$ does not entail $[\,]$.) The permission GC controls what permissions can be discarded. Here, we let GC stand for $\exists H.\, H$: this means that any permission can be discarded[7]. The last three rules in Figure 4 are elimination rules for pure assertions, disjunction, and existential quantification. Note that the standard symmetric rule of disjunction, shown below, follows directly from the rules EXTRACT-OR and CONSEQUENCE.

$$\textsc{disjunction}\quad \frac{\{H_1\}\ t\ \{Q_1\} \qquad \{H_2\}\ t\ \{Q_2\}}{\{H_1 \curlyvee H_2\}\ t\ \{Q_1 \curlyvee Q_2\}}$$

The syntax-directed reasoning rules appear in Figure 5. They are standard, except for the FRAMED SEQUENCING RULE and the NEW READ AXIOM, which have been explained earlier (§2.3, §2.4).

The implications in the premises of EXTRACT-PROP and IF and the universal quantifiers in the premises of EXTRACT-EXISTS and FRAMED SEQUENCING RULE are part of the metalanguage (which, in our formalization, is Coq). Thus, in reality, we work with assertions of the form $\forall \Gamma.\ \{H\}\ t\ \{Q\}$, where Γ represents a metalevel hypothesis.

[7] $\exists H.\, H$ is equivalent to true. If the programming language had explicit deallocation instead of garbage collection, one might wish to define GC as $\exists H.\, \mathsf{RO}(H)$, which means that read-only permissions can be implicitly discarded, but read-write permissions cannot. This restriction would be necessary in order to enforce "complete collection", that is, to ensure that every reference cell is eventually deallocated.

3.4 Treatment of variables and functions

Our treatment of variables, as well as the manner in which we reason about functions, may seem somewhat mysterious or unusual. We briefly explain them here. This material is entirely independent of the issue of read-only permissions, so this section may safely be skipped by a reader who wishes to focus on read-only permissions.

In the paper presentation, we identify program variables with the variables of the metalanguage. For example, in the FRAMED SEQUENCING RULE, the name x that occurs in the conclusion $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$ denotes a program variable, while the name x that is universally quantified in the second premise denotes a variable of the metalanguage.

In our Coq formalization, we clearly distinguish between program variables and metavariables. On the one hand, program variables are explicitly represented as identifiers (which may be implemented as integers or as strings). They are explicitly embedded in the syntax of values. The type of values, Val, is inductively defined (this is a "deep embedding"). On the other hand, metavariables are not "represented as" anything. They *are* just Coq variables of type Val, that is, they stand for an unknown value. To bridge the gap between program variables and metavariables, a substitution is necessary. For example, in Coq, the FRAMED SEQUENCING RULE is formalized as follows:

$$\begin{aligned} \forall x t_1 t_2 H H' Q Q'. \quad & \{H\}\ t_1\ \{Q'\} \\ & \wedge\ (\forall X.\ \{Q'\ X \star H'\}\ ([X/x]\, t_2)\ \{Q\}) \\ & \Rightarrow \{H \star H'\}\ (\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)\ \{Q\} \end{aligned}$$

Note how, in the second premise, a metavariable X (of type Val) is substituted for the program variable x in the term t_2. The metavariable X denotes the runtime value of the program variable x.

As one descends into the syntax of a term, metavariables are substituted for program variables, as explained above. Thus, one can never reach a leaf that is an occurrence of a program variable x! If a leaf labeled x originally existed in the term, it must be replaced with a value X when the binder for x is entered.

This explains a surprising feature of our reasoning rules, which is that they do not allow reasoning about terms that have free program variables. The rule IF, for instance, allows reasoning about a term of the form $(\mathsf{if}\ n\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)$, where n is a literal integer value. It does not allow reasoning about $(\mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)$, where x is a program variable. As argued above, this is not a problem. Similarly, APP expects the value v_1 to be a λ-abstraction; it does not allow v_1 to be a program variable. NEW READ AXIOM and SET expect the first argument of get and set to be a literal memory location; it cannot be a program variable.

Another possibly mysterious aspect of our presentation is the treatment of functions. In apparence, our only means of reasoning about functions is the rule APP, which states that, in order to reason about a call to a literal function (a λ-abstraction), one should substitute the actual arguments for the formal parameters in the function's body, and reason about the term thus obtained. At

first, this may not seem modular: in Hoare logic, one expects to first assign a specification to each function, then check that each function body satisfies its specification, under the assumption that each function call satisfies its specification. In a total correctness setting, one must also establish termination.

It turns out that, by virtue of the power of the metalanguage, this style of reasoning is in fact possible, based on the rules that we have given. We cannot explain everything here, but present one rule for reasoning about the definition and uses of a (possibly recursive) function. This rule can be derived from the rules that we have given. It is as follows:

$$\begin{array}{l}\left(\begin{array}{ll}\forall S.\forall F. & \left(\left(\begin{array}{ll}\forall X H' Q'. & \{H'\}\ ([F/f]\,[X/x]\,t_1)\ \{Q'\} \\ & \Rightarrow \{H'\}\ (F\,X)\ \{Q'\}\end{array}\right) \Rightarrow S\,F\right) \\ & \quad \wedge\ (S\,F \Rightarrow \{H\}\ ([F/f]\,t_2)\ \{Q\})\end{array}\right) \\ \Rightarrow\ \{H\}\ (\mathsf{let}\ f = \mu f.\lambda x.t_1\ \mathsf{in}\ t_2)\ \{Q\}\end{array}$$

This rule provides a way of establishing a Hoare triple about a term of the form $\mathsf{let}\ f = \mu f.\lambda x.t_1\ \mathsf{in}\ t_2$. In order to establish such a triple, it suffices to:

1. Pick a specification S, whose type is $\mathsf{Val} \to \mathsf{Prop}$. This is typically a Hoare triple, which represents a specification of the function $\mu f.\lambda x.t_1$.
2. Establish the desired triple about the term t_2, under the assumption that the function satisfies the specification S. (This is the third line above.) In doing so, one does not have access to the code of the function: it is represented by a metavariable F about which nothing is known but the hypothesis $S\,F$.
3. Prove that the function satisfies the specification S. (These are the first two lines above.) The function is still represented by a metavariable F. This time, one has access to the hypothesis that invoking F is equivalent to executing the function body t_1, under a substitution of actual arguments for formal parameters. If the function $\mu f.\lambda x.t_1$ is recursive, then this proof must involve induction (which is carried out in the metalanguage).

4 Model

In this section, we establish the soundness of our extension of Separation Logic. We first provide a concrete model of permissions, then give an interpretation of triples with respect to which each of the reasoning rules is proved sound.

4.1 A model of permissions

In traditional Separation Logic, a permission is interpreted as a predicate over heaps, also known as heap fragments. There, a "heap" coincides with what we have called a "memory", that is, a finite map of memory locations to values. Then, famously, a separating conjunction $H_1 \star H_2$ is satisfied by a heap h if and only if h is the disjoint union of two subheaps that respectively satisfy H_1 and H_2.

Two memories m_1 and m_2 have disjoint domains:

$$m_1 \perp m_2 \quad \equiv \quad \mathsf{dom}\, m_1 \cap \mathsf{dom}\, m_2 = \varnothing$$

Two memories r_1 and r_2 agree on the intersection of their domains:

$$\mathsf{agree}\ r_1\ r_2 \quad \equiv \quad \forall l v_1 v_2.\ (l, v_1) \in r_1 \wedge (l, v_2) \in r_2 \Rightarrow v_1 = v_2$$

Two heaps h_1 and h_2 are compatible:

$$\mathsf{compatible}\ h_1\ h_2 \quad \equiv \quad (\mathsf{agree}\ h_1.\mathsf{r}\ h_2.\mathsf{r}) \wedge (h_1.\mathsf{f} \perp h_2.\mathsf{f} \perp (h_1.\mathsf{r} \cup h_2.\mathsf{r}))$$

The composition of two compatible heaps h_1 and h_2:

$$h_1 + h_2 \quad \equiv \quad ((h_1.\mathsf{f} \uplus h_2.\mathsf{f}), (h_1.\mathsf{r} \cup h_2.\mathsf{r}))$$

Fig. 6. Compatibility and composition of heaps

$$\begin{aligned}
[P] &\equiv \lambda h.\ (h.\mathsf{f} = \varnothing) \wedge (h.\mathsf{r} = \varnothing) \wedge P \\
l \hookrightarrow v &\equiv \lambda h.\ (h.\mathsf{f} = (l \mapsto v)) \wedge (h.\mathsf{r} = \varnothing) \\
H_1 \star H_2 &\equiv \lambda h.\ \exists h_1 h_2.\ \mathsf{compatible}\ h_1\ h_2 \wedge (h = h_1 + h_2) \wedge H_1\, h_1 \wedge H_2\, h_2 \\
H_1 \curlyvee H_2 &\equiv \lambda h.\ H_1\, h \vee H_2\, h \\
\exists x.\, H &\equiv \lambda h.\ \exists x.\ H\, h \\
\mathsf{RO}(H) &\equiv \lambda h.\ (h.\mathsf{f} = \varnothing) \wedge \exists h'.\ (h.\mathsf{r} = h'.\mathsf{f} \uplus h'.\mathsf{r}) \wedge H\, h' \\
\mathsf{normal}(H) &\equiv \forall h.\ H\, h \Rightarrow h.\mathsf{r} = \varnothing \\
H_1 \rhd H_2 &\equiv \forall h.\ H_1\, h \Rightarrow H_2\, h
\end{aligned}$$

Fig. 7. Interpretation of permissions

In the following, we need a slightly more complex model, where a "heap" is a richer object than a "memory". Whereas a memory maps locations to values, a heap must additionally map memory locations to access rights: that is, it must keep track of which memory locations are considered accessible for reading and writing and which memory locations are accessible only for reading. A permission remains interpreted as a predicate over heaps.

We let a "heap" be a pair (f, r) of two memories f and r whose domains are disjoint[8]. (We let f and r range over memories.) The memory f represents the memory cells that are fully accessible, that is, accessible for reading and writing. The memory r represents the memory cells that are accessible only for reading. We note that there exist other (isomorphic) concrete representations of heaps: for instance, we could have defined a heap as a map of memory locations to pairs of a value and an access right (either "read-write" or "read-only").

We let h range over heaps. We write $h.\mathsf{f}$ for the first component of the pair h (that is, the read-write memory) and $h.\mathsf{r}$ for its second component (the read-only memory).

In traditional Separation Logic, two heaps are compatible (that is, can be composed) if and only if they have disjoint domains, and their composition is just

[8] Technically, in Coq, a heap is defined as a pair (f, r) accompanied with a proof that f and r have disjoint domains. Thus, whenever in the paper we assemble a heap (f, r), we have an implicit obligation to prove $\mathsf{dom}\, f \cap \mathsf{dom}\, r = \varnothing$.

their union. Here, because heaps contain information about access rights, we need slightly more complex notions of compatibility and composition of heaps. These notions are defined in Figure 6. We first introduce a few notations. We write $m_1 \perp m_2$ when the memories m_1 and m_2 have disjoint domains. By extension, we write $m_1 \perp m_2 \perp m_3$ when the memories m_1, m_2 and m_3 have pairwise disjoint domains. We write $\mathsf{agree}\ r_1\ r_2$ when the memories r_1 and r_2 agree where their domains overlap.

These notations allow us to succinctly state when two heaps h_1 and h_2 are compatible. First, the read-only components of h_1 and h_2 must agree where their domains overlap. Second, the read-write component of h_1, the read-write component of h_2, and the combined read-only components of h_1 and h_2 must have pairwise disjoint domains. When two heaps h_1 and h_2 are compatible, they can be composed. Composition is performed component-wise, that is, by taking the (disjoint) union of the read-write components and the (compatible) union of the read-only components. (The hypothesis that h_1 and h_2 are compatible is used to meet the proof obligation that $h_1 + h_2$ is a well-formed heap whose read-write and read-only components have disjoint domains.) Composition, where defined, is associative and commutative.

The interpretation of permissions appears in Figure 7. The interpretation of the standard permission forms is essentially standard: superficial adaptations are required to deal with the fact that a heap is a pair of a read-write memory and a read-only memory. The permission $[P]$ is satisfied by a heap whose read-write and read-only components are both empty, provided the proposition P holds. The permission $l \hookrightarrow v$ is satisfied by a heap whose read-write component is a singleton memory $(l \mapsto v)$ and whose read-only component is empty. The permission $H_1 \star H_2$ is satisfied by a heap h if and only if h is the composition of two (compatible) subheaps h_1 and h_2 which respectively satisfy H_1 and H_2. The interpretation of disjunction and existential quantification is standard.

A key aspect is the interpretation of $\mathsf{RO}(H)$. A human-readable, yet relatively accurate rendering of the formal meaning of $\mathsf{RO}(H)$ is as follows: "we do not have write access to any memory locations, but if we did have read-write (instead of read-only) access to certain locations, then H would hold". Technically, this is expressed as follows. The permission $\mathsf{RO}(H)$ is satisfied by a heap h if (1) the read-write component of h is empty and (2) the read-only component of h is of the form $h'.\mathsf{f} \uplus h'.\mathsf{r}$, where h' satisfies H. Thus, "RO" is a modality: it changes the "world" (the heap) with respect to which H is interpreted. In the outside world h, everything must be marked as read-only, whereas in the inside world h', some locations may be marked as read-write.

As explained earlier (§2.2), the meaning of $\mathsf{normal}\, H$ is defined as follows: a permission H is normal if and only if every heap h that satisfies it has an empty read-only component.

Entailment is defined in a standard way: $H_1 \rhd H_2$ holds if every heap that satisfies H_1 also satisfies H_2.

Lemma 1. *The above definitions validate the laws listed in Figures 1 and 2.*

4.2 A model of triples

We now wish to assign an interpretation to a Hoare triple of the form $\{H\}\ t\ \{Q\}$. Then, each of the reasoning rules of the logic can be proved sound, independently, by checking that it is a valid lemma. Before giving this interpretation, a couple of auxiliary definitions are needed.

First, if h is a heap, let $\lfloor h \rfloor$ be the memory $h.\mathsf{f} \uplus h.\mathsf{r}$. That is, $\lfloor h \rfloor$ is the memory obtained by forgetting the distinction between read-write and read-only locations in the heap h. This definition serves as a link between memories, which exist at runtime (they appear in the operational semantics: see Figure 3), and heaps, which additionally contain access right information. This information does not exist at runtime: it is "ghost" data.

Second, if H is a permission, let $\mathsf{on\text{-}some\text{-}rw\text{-}frag}(H)$ stand for the permission (that is, the predicate over heaps) that holds of a heap h if and only if h is the composition of two heaps h_1 and h_2, where h_1 has an empty read-only component and satisfies H. In mathematical notation:

$$\begin{array}{l}\mathsf{on\text{-}some\text{-}rw\text{-}frag}(H) \ \equiv \\ \quad \lambda h.\, \exists h_1 h_2.\, \mathsf{compatible}\ h_1\ h_2\ \wedge\ h = h_1 + h_2\ \wedge\ h_1.\mathsf{r} = \varnothing\ \wedge\ H\, h_1\end{array}$$

Intuitively, $\mathsf{on\text{-}some\text{-}rw\text{-}frag}(H)\, h$ asserts that H holds of a fragment of h whose read-only component is empty. This definition is used in the following to precisely express the meaning of postconditions, which in reality do not apply to the whole final heap, but to some fragment of the final heap whose read-only component is empty, or equivalently, to "some read-write fragment" of the final heap.

The interpretation of triples is now defined as follows:

Definition 1 (Interpretation of triples). *A semantic Hoare triple* $\{H\}\ t\ \{Q\}$ *is a short-hand for the following statement:*

$$\forall h_1 h_2.\, \begin{cases} \mathit{compatible}\ h_1\ h_2 \\ H\, h_1 \end{cases} \Rightarrow\ \exists v h_1'.\, \begin{cases} \mathit{compatible}\ h_1'\ h_2 \\ t/\lfloor h_1 + h_2 \rfloor \Downarrow v/\lfloor h_1' + h_2 \rfloor \\ h_1'.r = h_1.r \\ \mathit{on\text{-}some\text{-}rw\text{-}frag}(Q\, v)\, h_1' \end{cases}$$

In order to understand this definition, it is useful to first read the special case where the heap h_2 is empty:

$$\forall h_1.\ H\, h_1 \Rightarrow\ \exists v h_1'.\, \begin{cases} t/\lfloor h_1 \rfloor \Downarrow v/\lfloor h_1' \rfloor \\ h_1'.\mathsf{r} = h_1.\mathsf{r} \\ \mathsf{on\text{-}some\text{-}rw\text{-}frag}(Q\, v)\, h_1' \end{cases}$$

This may be read as follows. Let h_1 be an arbitrary initial heap that satisfies the permission H. Then, the term t, placed in the memory $\lfloor h_1 \rfloor$, runs safely and terminates, returning a value v in a final memory that can be described as $\lfloor h_1' \rfloor$, for some heap h_1'. The heap h_1' obeys the constraint $h_1'.\mathsf{r} = h_1.\mathsf{r}$, which means

that the set of all read-only locations is unchanged[9] and that the content of these locations is unchanged as well. Last, the permission $Q\ v$ is satisfied by some read-write fragment of the heap h_1'[10].

In the general case, where h_2 is an arbitrary heap, the definition of $\{H\}\ t\ \{Q\}$ states that the execution of the term t cannot affect a subheap h_2 which t "does not know about". Thus, running t in an initial heap $\lfloor h_1 + h_2 \rfloor$ must yield a final heap of the form $\lfloor h_1' + h_2 \rfloor$. This requirement has the effect of "building the frame rule into the interpretation of triples". It is standard [14, Definition 11]. We note that the memories $\lfloor h_1 \rfloor$ and $\lfloor h_2 \rfloor$ are not necessarily disjoint: indeed, the read-only components of the heaps h_1 and h_2 may have overlapping domains.

Theorem 1 (Soundness). *The above definition of triples validates all of the reasoning rules of Figures 4 and 5.*

Proof. We refer the reader to our Coq formalization [12]. □

5 Related work

5.1 The C/C++ "const" modifier

The C and C++ languages provide a type qualifier, `const`, which, when applied to the type of a pointer, makes this pointer usable only for reading. For example, a pointer of type `const int*` offers read access to an integer memory cell, but does not allow mutating this cell. A pointer of type `int*` can be implicitly converted to a pointer of type `const int*`.

The `const` qualifier arguably suffers from at least two important defects. First, as aliasing is not restricted, the above conversion rule immediately implies that a single pointer can perfectly well be stored at the same time in two distinct variables of types `const int*` and `int*`. Thus, although `const` prevents writing (through this pointer), it does not guarantee that the memory cell cannot be written (through an alias). Second, `const` does not take effect "in depth". If `t` has type `const tree*`, for instance, then `t->left` has type `tree*`, as opposed to `const tree*`. Thus, the `const` qualifier, applied to the type `tree*`, does not forbid modifications to the tree.

For these reasons, when a function expects a `const` pointer to a mutable data structure, one cannot be certain that this data structure is unaffected by a call to this function. In contrast, our read-only permissions do offer such a guarantee.

[9] That is, no read-write locations become read-only, or vice-versa. This reflects the fact that "RO" permissions appear and disappear following a lexical scope discipline. As a consequence, any newly-allocated memory cells must be marked read-write in h_1'.

[10] The operator on-some-rw-frag is used here for two reasons. First, in the presence of the rules DISCARD-PRE and DISCARD-POST, which discard arbitrary permissions, one cannot expect the postcondition $Q\ v$ to be satisfied by the whole final heap h_1'. Instead, one should expect $Q\ v$ to be satisfied by a fragment of h_1'. Second, $Q\ v$ is typically a normal permission, which can be satisfied only by a heap whose read-only component is empty. So, one may expect that $Q\ v$ is satisfied by a "read-write" fragment of h_1'.

5.2 The "read-only frame" connective

Jensen *et al.* [23] present a Separation Logic for "low-level" code. It is "high-level" in the sense that, even though machine code does not have a built-in notion of function, the logic offers structured reasoning rules, including first- and higher-order frame rules. The logic is stratified: assertions and specifications form two distinct levels. At the specification level, one finds a "frame" connective $\otimes$ and a "read-only frame" connective $\oslash$.

When applied to the specification of a first-order function, the "frame" connective has analogous effect to the "macro-expansion" scheme that was discussed earlier (§1.2): framing such a specification S with an assertion R amounts to applying $\_ \star R$ to the pre- and postcondition. Thus, if R is (say) $h \rightsquigarrow \mathsf{HashTable}\, M$, then the specification $S \otimes R$ states that h must remain a valid hash table that represents the dictionary M, but does not require that the table be unchanged: it could, for instance, be resized.

The "read-only frame" connective $\oslash$ is stronger: the specification $S \oslash R$ requires not just that the assertion R be preserved, but that the concrete heap fragment that satisfies R be left in its initial state. It is defined on top of the "frame" connective by bounded quantification. A typical use is to indicate that the code of a function (which, in this machine model, is stored in memory) must be accessible for reading, and is not modified when the function is called. Jensen *et al.*'s "read-only frame" connective allows "read-only" memory to be temporarily modified, as long as its initial state is restored upon exit. Therefore, it is quite different from our read-only permissions, which at the same time impose a restriction and offer a guarantee: they prevent the current function from modifying the "read-only" memory, and they guarantee that a callee cannot modify it either. Furthermore, our read-only permissions can be duplicated and discarded, whereas Jensen *et al.*'s "read-only frame" connective exists at the specification level: they have no read-only assertions.

5.3 Thoughts about lexical scope

The read-only frame rule takes advantage of lexical scope: it applies to a code block with well-defined entry and exit points, and governs how permissions are transformed when control enters and exits this block. Upon entry, a read-write permission is transformed to a read-only permission; upon exit, no read-only permissions are allowed to go through, and the original read-write permission reappears. The soundness of this rule relies on the fact that read-only permissions cannot escape through side channels[11].

There are several type systems and program logics in the literature which rely on lexical scope in a similar manner, sometimes for the same purpose (that

[11] For instance, we do not have concurrency, so a read-only permission cannot be transmitted to another thread via a synchronization operation. Furthermore, unlike Mezzo [2], we do not allow a closure to capture a duplicable permission, so a read-only permission cannot escape by becoming hidden in a closure.

is, to temporarily allow shared read-only access to a mutable data structure), sometimes for other purposes.

Wadler's "let!" construct [32, §4], for instance, is explicitly designed to allow temporary shared read-only access to a "linear value", that is, in our terminology, a uniquely-owned, mutable data structure. The "let!' rule changes the type of a variable x from T outside the block to $!T$ inside the block, which means that x temporarily becomes shareable and accessible only for reading. In order to ensure that no component of x is accessed for reading after the block is exited, Wadler requires a stronger property, namely that no component of x is accessible through the result value. Furthermore, in order to enforce this property, he imposes an even more conservative condition, namely that the result type U be "safe for T". In comparison, things are simpler for us. Separation Logic distinguishes values and permissions: thus, we do not care if a value (the address of x, or of a component of x) escapes, as long as no read permission escapes. Furthermore, in our setting, it is easy to enforce the latter condition. Technically, the side condition "$\mathsf{normal}\, H'$" in the READ-ONLY FRAME RULE plays this role. At a high level, this side condition implies that read-only permissions appear only in preconditions, never in postconditions. In Wadler's system, in contrast, the "!" modality describes both inputs and outputs, and describes both permanently-immutable and temporarily-read-only data, so things are less clear-cut.

In Vault [17], the "focus" mechanism temporarily yields a unique read-write permission for an object that inhabits a region (therefore, can be aliased, so normally would be accessible only for reading). Meanwhile, the permission to access this region is removed. This is essentially the dual of the problem that we are addressing! In the case of "focus", it is comparatively easy to ensure that the temporary read-write permission does not escape: as this is a unique permission, it suffices to require it upon exit. For the same reason, it is possible to relax the lexical scope restriction by using an explicit linear implication [17, §6]. Boyland and Retert's explanation of "borrowing" [8] is also in terms of "focus".

Gordon *et al.* [18] describe a variant of C# where four kinds of references are distinguished, namely: ordinary writable references; readable references, which come with no write permission and no guarantee; immutable references, which come with a guarantee that nobody has (or will ever have) write permission; and isolated references, which form a unique entry point into a cluster of objects. Quite strikingly, the system does not require any accounting. Lexical scope is exploited in several interesting ways. In particular, the "isolation recovery" rule states that, if, upon entry into a block, only isolated and immutable references are available, and if, upon exit of that block, a single writable reference is available, then this reference can safely be viewed as isolated. (The soundness of this rule relies on the fact that there are no mutable global variables.) This rule may seem superficially analogous to the read-only frame rule, in that a unique permission is lost and recovered. It is unclear to us whether there is a deeper connection. The system has a typing rule for structured parallelism and allows a mutable data structure to be temporarily shared (for reading) by several threads.

Gordon *et al.*'s work is part of a line of research on "reference immutability", where type qualifiers are used to control object mutation. We refer to the reader to Gordon *et al.*'s paper [18] and to Potanin *et al.*'s survey [27]. Coblenz *et al.*'s recent study of language support for immutability [13] is also of interest.

Rust [31] has lexically-scoped "borrows", including "immutable borrows", during which multiple temporary read-only pointers into a uniquely-owned mutable data structure can be created. The borrowing discipline does not require any counting, but involves "lifetimes", a form of region variables. Lifetimes can often be hidden in the surface syntax, but must sometimes be exposed to the programmer. In contrast, our read-only permissions require neither counting nor region variables. Reed [29] offers a tentative formal description of Rust's borrowing discipline.

5.4 Fractional permissions

Fractional permissions were introduced by Boyland [6] with the specific purpose of enabling temporary shared read-only access to a data structure. They have been integrated into several variants of Concurrent Separation Logic [5,19,21] and generalized in several ways, e.g., by replacing fractions with more abstract "shares" [16]. They are available in several program verification tools, including VeriFast [22], Chalice [24,20], and the Verified Software Toolchain [1].

In its simplest incarnation, a fractional permission takes the form $l \overset{\alpha}{\hookrightarrow} v$, where α is a rational number in the range $(0, 1]$. If α is 1, then this permission grants unique read-write access to the memory location l; if α is less than 1, then it grants shared read access. The following conversion rule allows splitting and joining fractional permissions:

$$(l \overset{\alpha+\beta}{\hookrightarrow} v) \;=\; (l \overset{\alpha}{\hookrightarrow} v) \star (l \overset{\beta}{\hookrightarrow} v) \qquad \text{when } \alpha, \beta, (\alpha + \beta) \in (0, 1]$$

Thanks to this rule, one can transition from a regime where a single thread has read-write access to a regime where several threads have read-only access, and back. Fractional permissions are not duplicable, and must not be carelessly discarded: indeed, in order to move back from read-only regime to read-write regime, one must prove that "no share has been lost" and that the fraction 1 has been recovered. This requires "accounting", that is, arithmetic reasoning, as well as returning fractional permissions in postconditions. In contrast, our proposal is less expressive in some ways (for instance, it does not support unstructured concurrency; see §6.2) but does not require accounting: our read-only permissions can be freely duplicated and discarded.

Fractional permissions also allow expressing a form of irrevocable (as opposed to temporary) read-only permissions. Define $(l \overset{\mathsf{ro}}{\hookrightarrow} v)$ as follows:

$$(l \overset{\mathsf{ro}}{\hookrightarrow} v) = \exists \alpha \in (0, 1).\ (l \overset{\alpha}{\hookrightarrow} v)$$

From this definition, it follows that $(l \overset{\mathsf{ro}}{\hookrightarrow} v)$ is duplicable. It also follows that $(l \overset{1}{\hookrightarrow} v)$ can be converted to $(l \overset{\mathsf{ro}}{\hookrightarrow} v)$, but not the other way around: the transition from read-write to read-only mode, in this case, is permanent.

In the systems of fractional permissions mentioned above, the fraction α is built into the points-to assertion $l \overset{\alpha}{\hookrightarrow} v$. Boyland [7] studies a more general idea, "scaling", where any permission H can be scaled by a fraction: that is, $H ::= \alpha.H$ is part of the syntax of permissions. Scaling seems a desirable feature, as it allows expressing read-only access to an abstract data structure, as in, say, $\frac{1}{2}(h \rightsquigarrow \mathsf{HashTable}\, M)$, which is impossible when scaling is built into points-to assertions. However, scaling exhibits a problematic interaction with disjunction and existential quantification. Boyland shows that "reasoning with a fractional unrestricted existential is unsound" [7, §5.4]. In short, it seems difficult to find a model that validates both of the laws $(\alpha.H) \star (\beta.H) = (\alpha + \beta).H$ and $\alpha.(\exists x.\, H) = \exists x.\, (\alpha.H)$. Indeed, under these laws, $\frac{1}{2}.(l_1 \hookrightarrow v) \star \frac{1}{2}.(l_2 \hookrightarrow v)$ entails $\exists l.\, (1.(l \hookrightarrow v))$, which does not make intuitive sense. Boyland escapes this problem by restricting the existential quantifier so that it is precise: the construct $\exists x.\, H$ is replaced with $\exists y.\, (x \hookrightarrow y \star H)$. In contrast, our read-only permissions do not require any such restriction, yet do support "scaling", in the sense that "RO" can be applied to an arbitrary permission: $\mathsf{RO}(H)$ is well-formed and has well-defined meaning for every H.

Chalice offers "abstract read permissions" [20], an elaborate layer of syntactic sugar above fractional permissions. An abstract read permission, which could be written $l \overset{\mathsf{rd}}{\hookrightarrow} v$, is translated to a fractional permission $l \overset{\epsilon}{\hookrightarrow} v$, where the variable ϵ stands for an unknown fraction. The variable ϵ is suitably quantified: for instance, if this abstract read permission appears in a method specification, then ϵ is universally quantified in front of this specification [20, §4.1]. The system is powerful enough to automatically introduce and instantiate quantifiers and automatically split and join fractional permissions where needed. Unfortunately, because abstract read permissions are just fractional permissions, they are not duplicable, and they must not be carelessly discarded: they must be returned (or transferred to some other thread) so that the fraction 1 can eventually be recovered. Also, it is not known to us whether abstract read permissions can be explained to the programmer in a direct manner, without reference to fractional permissions.

In a somewhat related vein, Aldrich *et al.* [25] propose a type system where (among other features) out of a "unique" reference, any number of "local immutable" references can be temporarily "borrowed". The type system internally relies on integer accounting, but this is hidden from the user.

6 Potential applications and extensions

6.1 Where read-only permissions could (or could not) help

The second author has specified and proved an OCaml implementation of hash tables [28], in Separation Logic, using the first author's tool, CFML. Iterating on a hash table is possible either via a higher-order function, `HashTable.iter`, or via "cascades", a form of iterators, built by the function `HashTable.cascade`.

In either case, the specification should ideally be stated in such a way that the consumer has read-only access to the table while iteration is in progress. For this

purpose, the abstract predicate $h \rightsquigarrow \mathsf{HashTable}\, M$, which gives unique read-write access, is not appropriate. A finer-grained predicate, $h \rightsquigarrow \mathsf{HashTableInState}\, M\ s$, is introduced, where s is an abstract name for the current concrete state of the table. Then, a function that does not modify the table, like *population*, requires the permission $h \rightsquigarrow \mathsf{HashTableInState}\, M\ s$ and returns it. (It is polymorphic in s.) In contrast, a function that does modify the table, like *resize*, also requires $h \rightsquigarrow \mathsf{HashTableInState}\, M\ s$, but returns $\exists s'.\, h \rightsquigarrow \mathsf{HashTableInState}\, M\ s'$. In fact, because $h \rightsquigarrow \mathsf{HashTable}\, M$ is an abbreviation for $\exists s.\, h \rightsquigarrow \mathsf{HashTableInState}\, M\ s$, one can simply say that *resize* requires and returns $h \rightsquigarrow \mathsf{HashTable}\, M$.

The specification of `HashTable.iter` (not shown here) involves universal quantification over s and the predicate $h \rightsquigarrow \mathsf{HashTableInState}\, M\ s$. This allows expressing the fact that `iter` does not modify the table and requires the function that it receives as an argument to not modify it either. Read-only permissions, if implemented in CFML, would help simplify this specification. It would be sufficient to state that `iter` requires the permission $\mathsf{RO}(h \rightsquigarrow \mathsf{HashTable}\, M)$ and passes it on to the function that it receives as an argument.

The specification of `HashTable.cascade` (not shown either) also exploits s in order to express the fact that the iterator returned by `cascade` remains valid as long as the hash table is not modified. Unfortunately, because our read-only permissions have lexical scope, they cannot help state a simpler specification. Indeed, the iterator outlives the call to `cascade`: it still needs read access after this call is finished.

6.2 Parallelism and concurrency

We believe that our read-only permissions remain sound when the calculus is extended with "structured parallelism", that is, with a term construct $t ::= (t \parallel t)$ for evaluating two terms in parallel. The parallel composition rule of Concurrent Separation Logic [26] can be used:

$$\textsc{parallel composition}\quad \frac{\{H_1\}\ t_1\ \{Q_1\} \qquad \{H_2\}\ t_2\ \{Q_2\}}{\{H_1 \star H_2\}\ (t_1 \parallel t_2)\ \{Q_1 \star Q_2\}}$$

Because read-only permissions are duplicable, this rule, combined with the rule of consequence, allows read access to be shared between the threads t_1 and t_2. That is, the following rule is derivable:

$$\textsc{parallel composition with shared read}\quad \frac{\{H_1 \star \mathsf{RO}(H')\}\ t_1\ \{Q_1\} \qquad \{H_2 \star \mathsf{RO}(H')\}\ t_2\ \{Q_2\}}{\{H_1 \star H_2 \star \mathsf{RO}(H')\}\ (t_1 \parallel t_2)\ \{Q_1 \star Q_2\}}$$

This rule can be used to share read access between any number of threads. By combining it with the read-only frame rule, one obtains the following rule, which allows a mutable data structure (represented by the permission H') to temporarily be made accessible for reading to several threads and to become

again accessible for reading and writing once these threads are finished:

$$\textsc{parallel composition with temporary shared read}\quad \frac{\{H_1 \star \mathsf{RO}(H')\}\ t_1\ \{Q_1\} \qquad \{H_2 \star \mathsf{RO}(H')\}\ t_2\ \{Q_2\} \qquad \mathsf{normal}\ H'}{\{H_1 \star H_2 \star H'\}\ (t_1 \parallel t_2)\ \{Q_1 \star Q_2 \star H'\}}$$

At present, we do not have a proof that the PARALLEL COMPOSITION rule is sound. Our current proof technique apparently cannot easily accommodate it, primarily because it is based on a big-step operational semantics (Figure 3), which (to the best of our knowledge) cannot be easily extended to support parallel composition $(t_1 \parallel t_2)$.

These remarks lead to several questions. Could Separation Logic with read-only permissions be proved sound, based on a small-step operational semantics? Could the logic and its proof then be extended with support for structured parallelism? There seems to be no reason why they could not, but this requires further research.

Another question arises: could Separation Logic with read-only permissions be extended so as to support unstructured parallelism, that is, shared-memory concurrency with explicit threads and synchronization facilities, such as locks and channels? We do not have an answer. We know that naïvely transmitting a read-only permission from one thread to another would be unsound. So, probably, the logic would have to be made more complex, perhaps by explicitly annotating read-only permissions with "lifetime" information. Whether this can be done while preserving the simplicity of the approach is an open question. After all, the whole approach is worthwhile only as long as it remains significantly simpler than fractional permissions (§5.4), which offer a well-understood solution to this problem.

7 Conclusion

We have extended sequential Separation Logic with a simple form of temporary read-only permissions. We have argued that they can (at least in some situations) express more concise, more accurate, more useful specifications and give rise to simpler proofs. Our proposal involves very few additions to Separation Logic, namely the "RO" modality; the read-only frame rule, which subsumes the frame rule; and a generalized sequencing rule. We have given semantic meaning to read-only permissions and to Hoare triples in terms of heaps that include "ghost" access rights information. We have formalized the logic and its proof of soundness in Coq. We hope to implement read-only permissions in CFML [11] in the future.

References

1. Appel, A.W.: Verified software toolchain. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 6602, pp. 1–17. Springer (2011)
2. Balabonski, T., Pottier, F., Protzenko, J.: The design and formalization of Mezzo, a permission-based programming language. ACM Transactions on Programming Languages and Systems 38(4), 14:1–14:94 (2016)
3. Bengtson, J., Jensen, J.B., Birkedal, L.: Charge! A framework for higher-order separation logic in Coq. In: Interactive Theorem Proving (ITP). pp. 315–331 (2012)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 4111, pp. 115–137. Springer (2005)
5. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: Principles of Programming Languages (POPL). pp. 259–270 (2005)
6. Boyland, J.: Checking interference with fractional permissions. In: Static Analysis Symposium (SAS). Lecture Notes in Computer Science, vol. 2694, pp. 55–72. Springer (2003)
7. Boyland, J.T.: Semantics of fractional permissions with nesting. ACM Transactions on Programming Languages and Systems 32(6), 22:1–22:33 (2010)
8. Boyland, J.T., Retert, W.: Connecting effects and uniqueness with adoption. In: Principles of Programming Languages (POPL). pp. 283–295 (2005)
9. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NASA Formal Methods (NFM). Lecture Notes in Computer Science, vol. 6617, pp. 459–465. Springer (2011)
10. Charguéraud, A.: Characteristic Formulae for Mechanized Program Verification. Ph.D. thesis, Université Paris 7 (2010)
11. Charguéraud, A.: Characteristic formulae for the verification of imperative programs (2013), unpublished. `http://www.charglueraud.org/research/2013/cf/cf.pdf`
12. Charguéraud, A., Pottier, F.: Self-contained archive. `http://gallium.inria.fr/~fpottier/dev/seplogics/` (2017)
13. Coblenz, M.J., Sunshine, J., Aldrich, J., Myers, B.A., Weber, S., Shull, F.: Exploring language support for immutability. In: International Conference on Software Engineering (ICSE). pp. 736–747 (2016)
14. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: Principles of Programming Languages (POPL). pp. 287–300 (2013)
15. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 213–226 (2008)
16. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: Asian Symposium on Programming Languages and Systems (APLAS). Lecture Notes in Computer Science, vol. 5904, pp. 161–177. Springer (2009)
17. Fähndrich, M., DeLine, R.: Adoption and focus: practical linear types for imperative programming. In: Programming Language Design and Implementation (PLDI). pp. 13–24 (2002)

18. Gordon, C.S., Parkinson, M.J., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and reference immutability for safe parallelism. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 21–40 (2012)
19. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. Tech. Rep. MSR-TR-2007-39, Microsoft Research (2007)
20. Heule, S., Leino, K.R.M., Müller, P., Summers, A.J.: Abstract read permissions: Fractional permissions without the fractions. In: Verification, Model Checking and Abstract Interpretation (VMCAI). Lecture Notes in Computer Science, vol. 7737, pp. 315–334. Springer (2013)
21. Hobor, A., Appel, A.W., Zappa Nardelli, F.: Oracle semantics for concurrent separation logic. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 4960, pp. 353–367. Springer (2008)
22. Jacobs, B., Piessens, F.: The VeriFast program verifier. Tech. Rep. CW-520, Department of Computer Science, Katholieke Universiteit Leuven (2008)
23. Jensen, J.B., Benton, N., Kennedy, A.: High-level separation logic for low-level code. In: Principles of Programming Languages (POPL). pp. 301–314 (2013)
24. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 5502, pp. 378–393. Springer (2009)
25. Naden, K., Bocchino, R., Aldrich, J., Bierhoff, K.: A type system for borrowing permissions. In: Principles of Programming Languages (POPL). pp. 557–570 (2012)
26. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1–3), 271–307 (2007)
27. Potanin, A., Östlund, J., Zibin, Y., Ernst, M.D.: Immutability. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification, Lecture Notes in Computer Science, vol. 7850, pp. 233–269. Springer (2013)
28. Pottier, F.: Verifying a hash table and its iterators in higher-order separation logic. In: Certified Programs and Proofs (CPP). pp. 3–16 (2017)
29. Reed, E.: Patina: A formalization of the Rust programming language. Tech. Rep. UW-CSE-15-03-02, University of Washington (2015)
30. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). pp. 55–74 (2002)
31. The Mozilla foundation: The Rust programming language (2014)
32. Wadler, P.: Linear types can change the world! In: Broy, M., Jones, C. (eds.) Programming Concepts and Methods. North Holland (1990)