Separation Logic for Sequential Programs (Functional Pearl)

ARTHUR CHARGUÉRAUD, Inria & Université de Strasbourg, ICube, France

This paper presents a simple mechanized formalization of Separation Logic for sequential programs. This formalization is aimed for teaching the ideas of Separation Logic, including its soundness proof and its recent enhancements. The formalization serves as support for a course that follows the style of the successful *Software Foundations* series, with all the statement and proofs formalized in Coq. This course only assumes basic knowledge of λ-calculus, semantics and logics, and therefore should be accessible to a broad audience.

1 INTRODUCTION

Separation Logic brought a major breakthrough in the area of program verification [O'Hearn 2019] (Gödel Prize citation). Since its introduction, Separation Logic has made its way into a number of practical tools that are used on a daily basis for verifying programs ranging from operating systems kernels [Tuch et al. 2007; Xu et al. 2016] and file systems [Chen et al. 2015] to data structures [Pottier 2017] and graph algorithms [Guéneau et al. 2019]. These programs are written in various programming languages, ranging from assembly [Chlipala 2013; Ni and Shao 2006] and C [Appel et al. 2014] to OCaml [Charguéraud 2011] and Rust [Jung et al. 2017].

The key ideas of Separation Logic were devised by John Reynolds, inspired in part by older work by Burstall [1972]. Reynolds presented his ideas in lectures given in the fall of 1999. The proposed rules turned out to be unsound, but O'Hearn and Ishtiaq [2001] noticed a strong relationship with the logic of *bunched implications* [O'Hearn and Pym 1999], leading to ideas on how to set up a sound program logic. Soon afterwards, the seminal publications on Separation Logic appeared at the CSL workshop [O'Hearn et al. 2001] and at the LICS conference [Reynolds 2002].

Today, when I teach students about Separation Logic, many of them find it hard to believe that Separation Logic has not been around for ever, or at least for as long as program verification exists. Perhaps the best way to truly value Reynold's contribution is to realize that, following the introduction of the first program logics in the late sixties [Floyd 1967; Hoare 1969], people have tried for 30 years to verify programs *without* Separation Logic.

Given its great interest, Separation Logic should presumably be tought to most, if not all, students in the field of programming languages. While Concurrent Separation Logic is a notoriously hard topic [Jung et al. 2017; O'Hearn 2019], Separation Logic for sequential programs is accessible to master students with basic knowledge in logic and semantics. There exists a number of courses that cover Separation Logic for sequential programs, from Reynold's course notes [2006] to modern courses presenting Separation Logic in the context of mechanized proofs, e.g., [Appel 2014; Birkedal and Bizjak 2018; Chlipala 2018a]. However, as we argue in details in the related work section, existing presentations of Separation Logic either axiomatize the logic and omit important aspects of the soundness proof, or present a soundness proof that involves a number of technical obstacles that get in the way of the students' understanding.

This paper presents a formalization of Separation Logic for sequential programs that, we believe, is simple enough that master students can follow through every detail of its soundness proof, interactively in the Coq proof assistant. Actually, the main contribution of the work described in this paper is a course, entitled *Foundations of Separation Logic* and written in the style of the *Software Foundations* series [Pierce and many contributors 2016]. The present paper consists of a

Author's address: Arthur Charguéraud, Inria & Université de Strasbourg, ICube, France, arthur.chargueraud@inria.fr.

2018. 2475-1421/2018/3-ART1 $15.00
https://doi.org/

summary of the material from that course, with formal definitions formatted using LaTeX. We refer to the supplementary material [Anonymous 2020] for the proofs omitted from this paper, as well as numerous additional examples, exercises, comments, and discussion of alternative definitions.

The formalization of Separation Logic that we present targets a λ-calculus with imperative features. We consider this language for two reasons. First, it has proved well-suited for teaching, as it abstracts away from the details of industrial programming languages. Second, targeting a ML-style language with immutable variables and mutable heap-allocated memory cells leads to the simplest formulation of the reasoning rules, avoiding a number of complications associated with mutable variables. In technical terms, our Coq formalization relies on a standard *deep embedding* of an imperative λ-calculus. This embedding features: an inductive definition of the abstract syntax tree, a recursive definition of the capture-avoiding substitution function, and an inductive definition of the operational semantics. We consider a big-step semantics to simplify the soundness proof.

Our formalization targets the simplest variant of Separation Logic. The heap predicates are defined as higher-order logic predicates, that is, as plain Coq definitions. The reasoning rules of the logic are stated as lemmas that are proved correct with respect to the big-step evaluation rules. In technically terms, our formalization consists of a *shallow embedding* of Separation Logic in Coq. This approach naturally yields a very expressive higher-order Separation Logic. It is employed by the majority of practical verification tools that embed Separation Logic in a proof assistant. It thus makes sense to teach that approach to students that will use or develop those verification tools.

By presenting our course notes on Separation Logic, whose contents are summarized in the present paper, we aim to make the following contributions.

(1) We present a mechanized soundness proof for Separation Logic for sequential programs that we believe to be more accessible to students than previously-available proofs.
(2) We present the first course on Separation Logic written in the style of the *Software Foundations* series. This presentation style, in which every definition and every lemma is presented via its mechanized statement, has proved very effective for teaching material related to logic and programming languages. Note that the course is focused on the foundations of Separation Logic and the description of its features. It does not (yet) include a large collection of examples illustrating how to specify and verify data structures and algorithms.
(3) We present, within a same paper, five important enhancements to Separation Logic that were previously scattered in the literature from the past decade.
   - The ramified frame rule is a concise and practical rule that combines the frame rule and the rule of consequence into a single rule.
   - The inductive reasoning rule for loops allows applying the frame rule over the remaining iterations of a loop; it typically saves the need to introduce list segments in invariants.
   - The generalization of the magic wand operator to postconditions; this operator is useful for languages featuring terms with return values, in particular to state the ramified frame rule.
   - The set up of Separation Logic with customizable control over which heap predicates are *linear* and which ones are *affine*, i.e., over which heap predicates may be freely discarded.
   - The presentation of Separation Logic in weakest-precondition style, both for the statement of the reasoning rules and for the statement of function specifications.

This paper is organized as follows. We first give an overview of the key features of Separation Logic (§2). We then present the operators of the logic (§3), the syntax and semantics of the language (§4), the definition of triples and the statement of the reasoning rules (§5). Next, we describe additional techniques: inductive reasoning for loops (§6), the magic wand operator (§7), the generalization to a partially-affine logic (§8), and the presentation in weakest-precondition

style (§9). Finally, we give references for all the ideas presented, discuss related work (§10) and conclude (§11).

The supplementary material [Anonymous 2020] accompanying the present submission contains: (1) the full contents of the course, both in Coq and HTML format, (2) a file that contains just the core definitions and soundness proof (SLFMinimal.v), and (3) an appendix to the present paper. This appendix includes statistics on our soundness proof for Separation Logic (§A), the presentation of the proofs of representative reasoning rules (§B), an example Separation Logic proof (§C), an illustration of the benefits of the frame rule in the proof of recursive functions (§D), a solution to the *cps-append* verification challenge proposed by Reynolds (§E), a discussion of alternative structural rules (§F), a description of the treatment of assertions (§H), of arrays and records (§G), and n-ary functions (§I), and the description of an algorithm for simplifying entailment relations (§J).

2 OVERVIEW OF THE FEATURES OF SEPARATION LOGIC

This section gives an overview of the features that are specific to Separation Logic: the *separating conjunction* and the *frame rule*, which enable *local reasoning* and *small-footprint specifications*, the treatment of aliasing, the specification of recursive pointer-based data structures such as mutable linked lists, and the ability to ensure *complete deallocation* of all allocated data.

2.1 The frame rule

In Hoare logic, the behavior of a command t is specified through a *triple*, written $\{H\}\ t\ \{Q\}$, where the *precondition* H describes the input state and the *postcondition* Q describes the output state. Whereas in Hoare Logic H and Q describe the whole memory state, in Separation Logic they describe only a fragment of the memory state, a fragment that includes all the resources involved in the execution of the command t.

The *frame rule* asserts that if a command t safely executes in a given piece of state, then it also executes safely in a larger piece of state. More precisely, if t executes in a state described by H and produces a final state described by Q, then this program can also be executed in a state that extends H with a *disjoint* piece of state described by H'. The corresponding final state consists of Q extended with H', capturing the fact that the additional piece of state is unmodified by the execution of t. The frame rule enables *local reasoning*, defined as follows [O'Hearn et al. 2001].

> *To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.*

The frame rule is stated using the *separating conjunction*, written $\star$, which is a binary operator over *heap predicates*. In Separation Logic, pieces of states are traditionnally called *heaps*, and predicate over heaps are called *heap predicates*. Given two heap predicates H and H', the heap predicate $H \star H'$ describes a heap made of two disjoint parts, one that satisfies H and one that satisfies H'. The statement of the frame rule, shown below, asserts that any triple remains valid when extending both its precondition and its postcondition with an arbitrary predicate H'.

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \star H'\}}\ \textsc{frame-for-commands} \qquad \text{where } t \text{ is a command.}$$

In this paper, we do not consider a language of commands but a language based on the λ-calculus, with programs described as terms that evaluate to values. (The language is formalized in §4.1.) In that setting, a specification triple takes the form $\{H\}\ t\ \{\lambda x.\, H'\}$, where H describes the input state, x denotes the value produced by the term t, and H' describes the output state, with x bound in H'.

For such triples, the frame rule is stated either as:

$$\frac{\{H\}\ t\ \{\lambda x.\ H''\}}{\{H \star H'\}\ t\ \{\lambda x.\ H'' \star H'\}}\ \textsc{frame} \qquad \text{where } t \text{ is a term producing a value, and } x \notin \mathrm{fv}(H')$$

or, more concisely, as:

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \mathbin{\dot\star} H'\}}\ \textsc{frame} \qquad \text{where } Q \mathbin{\dot\star} H \ \equiv\ \lambda v.\, (Q\, v \star H).$$

2.2 Separation Logic specifications

What makes Separation Logic works smoothly in practice is that specifications are expressed using a small number of operators for defining heap predicates, such that these operators interact well with the separating conjunctions. The most important operators are summarized below—they appear in examples throughout the rest of this section, and are formally defined further on (§3.2).

- $p \hookrightarrow v$, to be read "p points to v", describes a single memory cell, allocated at address p, with contents v.
- [ ] describes an empty state.
- $[P]$ also describes an empty state, and moreover asserts that the proposition P is true.
- $H_1 \star H_2$ describes a heap made of two disjoint parts, one described by H_1 and another described by H_2.
- $\exists x.\, H$ and $\forall x.\, H$ are used to quantify variables in Separation Logic assertions.

We call these operators the *core heap predicate operators*, because all the other Separation Logic operators can be defined in terms of these core operators.

The heap predicate operators appear in the statement of preconditions and postconditions. For example, consider the specification of the function `incr`, which increments the contents of a reference cell. It is specified using a triple of the form $\{H\}\ (\texttt{incr}\ p)\ \{Q\}$.

Example 2.1 (Specification of the increment function).

$$\forall\, p\, n. \qquad \{p \hookrightarrow n\}\ (\texttt{incr}\ p)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}$$

The precondition describes the existence of a memory cell with an integer contents, through the predicate $p \hookrightarrow n$. The postcondition describes the final heap in the form $p \hookrightarrow (n+1)$, reflecting the increment of the contents. The "$\lambda\_.$ " symbol at the head of the postcondition indicates that the value returned by $\texttt{incr}\ p$, namely the unit value, needs not be assigned a name.

Throughout the rest of the paper, the outermost universal quantifications (e.g., "$\forall\, p\, n.$") are left implicit, following standard practice.

2.3 Implications of the frame rule

The precondition in the specification of $\texttt{incr}\ p$ describes only the reference cell involved in the function call, and nothing else. Consider now the execution of $\texttt{incr}\ p$ in a heap that consists of two distinct memory cells, the first one being described as $p \hookrightarrow n$, and the other being described as $q \hookrightarrow m$. In Separation Logic, the conjunction of these two heap predicates are described by the heap predicate $(p \hookrightarrow n) \star (q \hookrightarrow m)$. There, the separating conjunction (a.k.a. the star) captures the property that the two cells are distinct. The corresponding postcondition of $\texttt{incr}\ p$ describes the updated cell $p \hookrightarrow (n+1)$ as well as the other cell $q \hookrightarrow m$, whose contents is not affected by the call to the increment function. The corresponding Separation Logic triple is therefore stated as follows.

Example 2.2 (Application of the frame rule on the specification of the increment function).

$$\{(p \hookrightarrow n) \star (q \hookrightarrow m)\}\ (\texttt{incr}\, p)\ \{\lambda\_.\ (p \hookrightarrow n+1) \star (q \hookrightarrow m)\}$$

The above triple is derivable from the one stated in Example 2.1 by applying the frame rule to add the heap predicate $q \hookrightarrow m$ both to the precondition and to the postcondition. More generally, any heap predicate H can be added to the original, minimalist specification of $\texttt{incr}\, p$. Thus we have:

$$\{(p \hookrightarrow n) \star H\}\ (\texttt{incr}\, p)\ \{\lambda\_.\ (p \hookrightarrow n+1) \star H\}.$$

2.4 Treatment of potentially-aliased arguments

We next discuss the case of potentially-aliased reference cells. In the previous example, we have considered two reference cells p and q assumed to be distinct from each other. Consider now a function that expects as arguments two reference cells, at addresses p and q, and increments both. Potentially, the two arguments might correspond to the same reference cell. The function thus admits two specifications. The first one describes the case of two *distinct* arguments, using separating conjunction to assert the difference. The second one describes the case of two *aliased* arguments, that is, the case $p = q$, for which the precondition describes only one reference cell.

Example 2.3 (Potentially aliased arguments). The function:

```
let incr_two p q = (incr p; incr q)
```

admits the following two specifications.

$$\begin{array}{c}\{(p \hookrightarrow n) \star (q \hookrightarrow m)\}\ (\texttt{incr\_two p q})\ \{\lambda\_.\ (p \hookrightarrow n+1) \star (q \hookrightarrow m+1)\}\\ \{p \hookrightarrow n\}\ (\texttt{incr\_two p p})\ \{\lambda\_.\ (p \hookrightarrow n+2)\}\end{array}$$

2.5 Small-footprint specifications

A Separation Logic triple captures all the interactions that a term may have with the memory state. Any piece of state that is not described explicitly in the precondition is guaranteed to remain untouched. Separation Logic therefore encourages *small footprint* specifications, i.e., specifications that mention nothing but what is strictly needed. The small-footprint specifications for the primitive operations `ref`, `get` and `set` are stated and explained next.

Example 2.4 (Specification of primitive operations on references).

$$\begin{array}{lll}\{[\,]\} & (\mathrm{ref}\, v) & \{\lambda r.\ \exists p.\ [r = p] \star (p \hookrightarrow v)\}\\ \{p \hookrightarrow v\} & (\mathrm{get}\, p) & \{\lambda r.\ [r = v] \star (p \hookrightarrow v)\}\\ \{p \hookrightarrow v\} & (\mathrm{set}\, p\, v') & \{\lambda\_.\ (p \hookrightarrow v')\}\end{array}$$

The operation $\texttt{ref}\, v$ can execute in the empty state, described by $[\,]$. It returns a value, named r, that corresponds to a pointer p, such that the final heap is described by $p \hookrightarrow v$. In the postcondition, the variable p is quantified existentially, and the pure predicate $[r = p]$ denotes an equality between the value r and the address p, viewed as an element from the grammar of values (formalized in §4.1). The operation $\texttt{get}\, p$ requires in its precondition the existence of a cell described by $p \hookrightarrow v$. Its postcondition asserts that the result value, named r, is equal to the value v, and that the final heap remains described by $p \hookrightarrow v$. The operation $\texttt{set}\, p\, v'$ also requires a heap described by $p \hookrightarrow v$. Its postcondition asserts that the updated heap is described by $p \hookrightarrow v'$. The result value, unit, is ignored.

The possibility to state a small-footprint specification for the allocation operation captures an essential property: the reference cell allocated by `ref` is implicitly asserted to be distinct from any pre-existing reference cell. This property can be formally derived by applying the frame rule to the specification triple for `ref`. For example, the triple stated below asserts that if a cell described by

$q \hookrightarrow v'$ exists before the allocation operation $\texttt{ref}\, v$, then the new cell described by $p \hookrightarrow v$ is distinct from that pre-existing cell. This freshness property is captured by the separating conjunction $(p \hookrightarrow v) \star (q \hookrightarrow v')$.

Example 2.5 (Application of the frame rule to the specification of allocation).

$$\{q \hookrightarrow v'\}\ (\mathsf{ref}\, v)\ \{\lambda r.\ \exists p.\ [r = p] \star (p \hookrightarrow v) \star (q \hookrightarrow v')\}$$

The strength of the separating conjunction is even more impressive when involved in the description of recursive data structures such as mutable lists, which we present next.

2.6 Representation of mutable lists

A mutable linked list consists of a chain of cells. Each cell contains two fields: the head field stores a value, which corresponds to an item from the list; the tail field stores either a pointer onto the next cell in the list, or the null pointer to indicate the end of the list.

Definition 2.6 (Representation of a list cell). A list cell allocated at address p, storing the value v and the pointer q, is represented by two singleton heap predicates, in the form:

$$(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)$$

where "$p.k$" is a notation for the address $p + k$, and "head ≡ 0" and "tail ≡ 1" denote the offsets.

A mutable linked list is described by a heap predicate of the form $\mathsf{MList}\, L\, p$, where p denotes the address of the head cell and L denotes the logical list of the elements stored in the mutable list. The predicate MList is called a *representation predicate*, because it relates the pair made of a pointer p and of the heap-allocated data structure that originates at p together with the logical representation of this data structure, namely the list L.

The predicate MList is defined recursively on the structure of the list L. If L is the empty list, then p must be null. Otherwise, L is of the form $x :: L'$. In this case, the head field of p stores the item x, and the tail field of p stores a pointer q such that $\mathsf{MList}\, L'\, q$ describes the tail of the list. The case disjunction is expressed using Coq's pattern matching construct.

Definition 2.7 (Representation of a mutable list).

$$\begin{array}{rcl} \mathsf{MList}\, L\, p & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \\ & & \mid x :: L' \Rightarrow \exists q.\ (p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{MList}\, L'\, q) \end{array}$$

Example 2.8 (Application of the predicate Mlist to a list of length 3). To see how Mlist unfolds on a concrete example, consider the example of a mutable list storing the values 8, 5, and 6.

$$\begin{array}{rcl} \mathsf{MList}\, (8 :: 5 :: 6 :: \mathsf{nil})\, p & \equiv & \exists p_1.\ (p.\mathsf{head} \hookrightarrow 8) \star (p.\mathsf{tail} \hookrightarrow p_1) \\ & & \star\ \exists p_2.\ (p_1.\mathsf{head} \hookrightarrow 5) \star (p_1.\mathsf{tail} \hookrightarrow p_2) \\ & & \star\ \exists p_3.\ (p_2.\mathsf{head} \hookrightarrow 6) \star (p_2.\mathsf{tail} \hookrightarrow p_3) \\ & & \star\ [p_3 = \mathsf{null}] \end{array}$$

Observe how the definition of Mlist, by iterating the separating conjunction operator, ensures that all the list cells are distinct from each other. In particular, Mlist precludes the possibility of cycles in the linked list, and precludes inadvertent sharing of list cells with other mutable lists.

Definition 2.7 characterizes Mlist by case analysis on whether the list L is empty. Another, equivalent definition instead characterizes Mlist by case analysis on whether the pointer p is null. This alternative definition is very useful because most list-manipulating programs involve code that tests whether the list pointer at hand is null.

Definition 2.9 (Alternative definition for Mlist).

$$\begin{array}{lll} \mathsf{Mlist}\, L\, p & \equiv & \mathsf{If}\ (p = \mathsf{null}) \\ & & \quad \mathsf{then}\ [L = \mathsf{nil}] \\ & & \quad \mathsf{else}\ \exists x L' q.\ [L = x :: L'] \star (p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q) \end{array}$$

Note that the above definition is not recognized as structurally-recursive by Coq. Nevertheless, its statement may be formulated as an equality, and proved correct with respect to Definition 2.7.

2.7 Operations on mutable lists

Consider a function that concatenates two mutable lists *in-place*. This function expects two pointers p_1 and p_2 that denote the addresses of two mutable lists described by the logical lists L_1 and L_2, respectively. The first list is assumed to be nonempty. The concatenation operation updates the last cell of the first list so that it points to p_2, the head cell of the second list. After this operation, the mutable list at address p_1 is described by the concatenation $L_1 +\!\!+ L_2$.

Example 2.10 (Specification of in-place append for mutable lists).

$$p_1 \neq \mathsf{null} \ \Rightarrow\ \{(\mathsf{Mlist}\, L_1\, p_1) \star (\mathsf{Mlist}\, L_2\, p_2)\}\ (\mathtt{mappend}\, p_1\, p_2)\ \{\lambda\_.\ \mathsf{Mlist}\, (L_1 +\!\!+ L_2)\, p_1\}$$

Observe how the specification above reflects the fact that the cells of the second list are absorbed by the first list during the operation. These cells are no longer independently available, hence the absence of the representation predicate $\mathsf{Mlist}\, L_2\, p_2$ from the postcondition.

Remark 2.11 (Alternative placement of pure preconditions). The hypothesis $p_1 \neq \mathsf{null}$ from the specification of the append function may be equivalently placed inside the precondition:

$$\{[p_1 \neq \mathsf{null}] \star (\mathsf{Mlist}\, L_1\, p_1) \star (\mathsf{Mlist}\, L_2\, p_2)\}\ (\mathtt{mappend}\, p_1\, p_2)\ \{\lambda\_.\ \mathsf{Mlist}\, (L_1 +\!\!+ L_2)\, p_1\}.$$

Yet, in general, leaving pure hypotheses as premises outside of triples tends to improve readability.

As second example, consider a function that takes as argument a pointer p to a mutable list, and allocates an entirely independent copy of that list, made of fresh cells. This function is specified as shown below. The precondition describes the input list as $\mathsf{Mlist}\, L\, p$, and the postcondition describes the output heap as $\mathsf{Mlist}\, L\, p \star \mathsf{Mlist}\, L\, p'$, where p' denotes the address of the new list.

Example 2.12 (Specification of a copy function for mutable lists).

$$\{\mathsf{Mlist}\, L\, p\}\ (\mathtt{mcopy}\, p)\ \{\lambda r.\ \exists p'.\ [r = p'] \star (\mathsf{Mlist}\, L\, p) \star (\mathsf{Mlist}\, L\, p')\}$$

The separating conjunction from the postcondition asserts that the original list and its copy do not share any cell: they are entirely disjoint from each other. An implementation and a proof for the function `mcopy` is given in the appendix. The key steps of that proof are summarized next.

Proof. The specification of `mcopy` is proved by induction on the length of the list L. When the list is nonempty, $\mathsf{Mlist}\, L\, p$ unfolds as $(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q)$. The induction hypothesis allows to assume the specification to hold for the recursive call of `mcopy` on the tail of the list, with the precondition $\mathsf{Mlist}\, L'\, q$. Over the scope of that call, the frame rule is used to put aside the head cell, described by $(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)$. Let q' denote the result of the recursive call, and let p' denote the address of a freshly-allocated list cell storing the value x and the tail pointer q'. The final heap is described by:

$$(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q) \star (p'.\mathsf{head} \hookrightarrow x) \star (p'.\mathsf{tail} \hookrightarrow q') \star (\mathsf{Mlist}\, L'\, q')$$

which may be folded to $(\mathsf{Mlist}\, L\, p) \star (\mathsf{Mlist}\, L\, p')$, matching the claimed postcondition. □

In the above proof, the frame rule enables reasoning about a recursive call *independently* of all the cells that have already been traversed by the outer recursive calls to `mcopy`. Without the frame rule, one would have to describe the full list at an arbitrary point during the recursion. Doing so requires describing the *list segment* made of cells ranging from the head of the initial list up to the pointer on which the current recursive call is made. Stating an invariant involving list segments is doable, yet involves more complex definitions and assertions. More generally, for a program manipulating tree-shaped data structures, the frame rule saves the need to describe a tree with a subtree carved out of it, thereby saving a significant amount of proof effort.

2.8 Reasoning about deallocation

Consider a programming language with explicit deallocation. For such a language, proofs in Separation Logic guarantee two essential properties: (1) a piece of data is never accessed after its deallocation, and (2) every allocated piece of data is eventually deallocated.

The operation $\texttt{free}\, p$ deallocates the reference cell at address p. This deallocation operation is specified through the following triple, whose precondition describes the cell to be freed by the predicate $p \hookrightarrow v$, and whose postcondition is empty, reflecting the loss of that cell.

Definition 2.13 (Specification of the free operation).

$$\{p \hookrightarrow v\}\ (\texttt{free}\, p)\ \ \{\lambda\_.\ [\,]\}$$

There is no way to get back the predicate $p \hookrightarrow v$ once it is lost. Because $p \hookrightarrow v$ is required in the precondition of all operations involving the reference p, Separation Logic ensures that no operations on p can be performed after its deallocation.

The next examples show how to specify the deallocation of a list cell and of a full list.

Example 2.14 (Deallocation of a list cell). The function `mfree_cell` deallocates a list cell.

$$\{(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)\}\ (\texttt{mfree\_cell}\, p)\ \{\lambda\_.\ [\,]\}.$$

Example 2.15 (Deallocation of a mutable list). The function `mfree_list` deallocates a list by recursively deallocating each of its cells. Its implementation is shown below (using ML syntax, even though the language considered features null pointers and explicit deallocation).

```
let rec mfree_list p =
  if p != null then (let q = p.tail in mfree_cell p; mfree_list q)
```

The specification of `mfree_list` admits the precondition $\mathsf{Mlist}\, L\, p$, describing the mutable list to be freed, and admits an empty postcondition, reflecting the loss of that list.

$$\{\mathsf{Mlist}\, L\, p\}\ (\texttt{mfree\_list}\, p)\ \{\lambda\_.\ [\,]\}$$

Remark 2.16 (Languages with implicit garbage collection). For languages equipped with a garbage-collector, Separation Logic can be adapted to allow freely discarding heap predicates (see §8).

3 HEAP PREDICATES AND ENTAILMENT

3.1 Representation of heaps

Let loc denote the type of locations, i.e., of memory addresses. This type may be realized using, e.g., natural numbers. Let val denote the type of values. The grammar of values depends on the programming language. Its formalization is postponed to §4.

A heap (i.e., a piece of memory state) may be represented as a finite map from locations to values. The finiteness property is required to ensure that fresh locations always exist. Let $\mathsf{fmap}\ \alpha\ \beta$ denote the type of finite maps from a type α to an (inhabited) type β.

Definition 3.1 (Representation of heaps). The type heap is defined as "fmap loc val".

Thereafter, let h denote a heap, that is, a piece of state. Let $h_1 \perp h_2$ assert that two heaps have disjoint domains, i.e., that no location belongs both to the domain of h_1 and to that of h_2. Let $h_1 \uplus h_2$ denote the union of two disjoint heaps. (The union operation may return arbitrary results when applied to non-disjoint arguments, i.e., arguments with overlapping domains.)

3.2 Heap predicates

A heap predicate, written H, is a predicate that asserts properties of a heap.

Definition 3.2 (Heap predicates). A heap predicate is a predicate of type heap → Prop.

The *core heap predicate operators*, informally introduced in §2.2, are realized as predicates over heaps, as shown below and explained next.

Definition 3.3 (Core heap predicates).

Heap predicate operator	Notation	Definition
empty predicate	$[\,]$	$\lambda h.\ h = \varnothing$
pure fact	$[P]$	$\lambda h.\ h = \varnothing \wedge P$
singleton	$p \mapsto v$	$\lambda h.\ h = (p \to v) \wedge p \neq \mathsf{null}$
separating conjunction	$H_1 \star H_2$	$\lambda h.\ \exists h_1 h_2.\ h_1 \perp h_2 \wedge h = h_1 \uplus h_2 \wedge H_1\, h_1 \wedge H_2\, h_2$
existential quantifier	$\exists x.\, H$	$\lambda h.\ \exists x.\ H\, h$
universal quantifier	$\forall x.\, H$	$\lambda h.\ \forall x.\ H\, h$

The definitions for the core heap predicates all take the form $\lambda h.\ P$, where P denotes a proposition. The empty predicate, written $[\,]$, characterizes a heap equal to the empty heap, written $\varnothing$. The pure predicate, written $[P]$, also characterizes an empty heap, and moreover asserts that the proposition P is true. The singleton heap predicate, written $p \mapsto v$, characterizes a heap described by a singleton map, written $p \to v$, which binds p to v. This predicate embeds the property $p \neq \mathsf{null}$, capturing the invariant that no data may be allocated at the null location. The separating conjunction, written $H_1 \star H_2$, characterizes a heap h that decomposes as the disjoint union of two heaps h_1 and h_2, with h_1 satisfying H_1 and h_2 satisfying H_2. The existential and universal quantifiers of Separation Logic allow quantifying entities at the level of heap predicates (heap → Prop), in contrast to the standard Coq quantifiers that operate at the level of propositions (Prop). Note that the quantifiers $\exists x.\, H$ and $\forall x.\, H$ may quantify values of any type, without restriction. In particular, they allow quantifying over heap predicates or proof terms.

Remark 3.4 (Encodings between the empty and the pure heap predicate). In Coq, the pure heap predicate $[P]$ can be encoded as "$\exists(p : P).\ [\,]$", that is, by quantifying over the existence of a proof term p for the proposition P. Note that the empty heap predicate $[\,]$ is equivalent to [True].

Remark 3.5 (Other operators). Traditional presentations of Separation Logic include four additional operators, $\bot$, $\top$, $\curlyvee$, and $\curlywedge$. These four operators may be encoded in terms of the ones from Definition 3.3, with the help of Coq's conditional construct. The table below presents the relevant encodings, in addition to providing direct definitions of these operators as predicates over heaps.

Heap predicate operator	Notation	Definition	Encoding
bottom	$\bot$	$\lambda h.$ False	[False]
top	$\top$	$\lambda h.$ True	$\exists(H : \text{heap} \to \text{Prop}).\, H$
disjunction	$H_1 \curlyvee H_2$	$\lambda h.\ (H_1\, h \vee H_2\, h)$	$\exists(b : \text{bool}).$ If b then H_1 else H_2
non-separating conjunction	$H_1 \curlywedge H_2$	$\lambda h.\ (H_1\, h \wedge H_2\, h)$	$\forall(b : \text{bool}).$ If b then H_1 else H_2

Definition 3.6 (Representation predicate for lists defined with disjunction). The representation predicate for lists introduced in Definition 2.8 can be reformulated using the disjunction operator instead of relying on pattern matching. The corresponding definition is as follows.

$$\begin{aligned}\mathsf{Mlist}\, L\, p \equiv \quad & \Big([p = \mathsf{null}] \star [L = \mathsf{nil}]\Big) \\ & \curlyvee\Big([p \neq \mathsf{null}] \star \exists x L' q.\ [L = x :: L'] \star (p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q)\Big)\end{aligned}$$

3.3 Entailment

The entailment relation, written $H_1 \vdash H_2$, asserts that any heap satisfying H_1 also satisfies H_2.

Definition 3.7 (Entailment relation).

$$H_1 \vdash H_2 \quad \equiv \quad \forall h.\ H_1\, h \Rightarrow H_2\, h$$

Entailment is used to state reasoning rules and to state properties of the heap predicates operators. The entailment relation defines an order relation on the set of heap predicates.

Lemma 3.8 (Entailment defines an order on the set of heap predicates).

HIMPL-REFL
$$\frac{}{H \vdash H}$$

HIMPL-TRANS
$$\frac{H_1 \vdash H_2 \qquad H_2 \vdash H_3}{H_1 \vdash H_3}$$

HIMPL-ANTISYM
$$\frac{H_1 \vdash H_2 \qquad H_2 \vdash H_1}{H_1 = H_2}$$

The antisymmetry property concludes on an equality between two heap predicates. To establish such an equality, it is necessary to exploit the principle of *predicate extensionality*. This principle asserts that if two predicates P and P', when applied to any argument x, yield logically equivalent propositions, then these two predicates can be considered equal in the logic. In proof assistants such as HOL or Isabelle/HOL, extensionality is built-in. In Coq, it needs to be either axiomatized or derived from two more fundamental extensionality axioms: extensionality for functions and extensionality for propositions. These standard axioms are formally stated as follows.

PREDICATE-EXTENSIONALITY:	$\forall A.$	$\forall (P\, P' : A \to \mathsf{Prop}).$	$(P\, x \Leftrightarrow P'\, x)$	$\Rightarrow$	$(P = P')$
FUNCTIONAL-EXTENSIONALITY:	$\forall A\, B.$	$\forall (f\, f' : A \to B).$	$(f\, x = f'\, x)$	$\Rightarrow$	$(f = f')$
PROPOSITIONAL-EXTENSIONALITY:		$\forall (P\, P' : \mathsf{Prop}).$	$(P \Leftrightarrow P')$	$\Rightarrow$	$(P = P')$

In summary, one may take FUNCTIONAL-EXTENSIONALITY and PROPOSITIONAL-EXTENSIONALITY as axioms in Coq, then from these two derive PREDICATE-EXTENSIONALITY and subsequently HIMPL-ANTISYM. The antisymmetry property plays a critical role for stating the key properties of Separation Logic operators in the form of equalities, as detailed next.

There are 6 fundamental properties of the separating conjunction operator. The first three capture the fact that $(\star, [\,])$ forms a commutative monoid: the star is associative, commutative, and admits the empty heap predicate as neutral element. The next two describe how quantifiers may be extruded from arguments of the star operator. The last one describes a monotonicity property, and is explained afterwards.

Lemma 3.9 (Fundamental properties of the star).

STAR-ASSOC:	$(H_1 \star H_2) \star H_3 = H_1 \star (H_2 \star H_3)$		
STAR-COMM:	$H_1 \star H_2 = H_2 \star H_1$		
STAR-NEUTRAL-R:	$H \star [\,] = H$		
STAR-EXISTS:	$(\exists x.\, H_1) \star H_2 = \exists x.\, (H_1 \star H_2)$		(if $x \notin H_2$)
STAR-FORALL:	$(\forall x.\, H_1) \star H_2 \vdash \forall x.\, (H_1 \star H_2)$		(if $x \notin H_2$)

STAR-MONOTONE-R:
$$\frac{H_1 \vdash H_1'}{H_1 \star H_2 \vdash H_1' \star H_2}$$

$$\textsc{pure-l}\ \frac{P \Rightarrow (H \vdash H')}{([P] \star H) \vdash H'} \qquad \textsc{exists-l}\ \frac{\forall x.\ (H \vdash H')}{(\exists x.\, H) \vdash H'} \qquad \textsc{forall-l}\ \frac{([a/x]\, H) \vdash H'}{(\forall x.\, H) \vdash H'} \qquad \textsc{exists-monotone}\ \frac{\forall x.\ (H \vdash H')}{(\exists x.\, H) \vdash (\exists x.\, H')}$$

$$\textsc{pure-r}\ \frac{(H \vdash H') \qquad P}{H \vdash (H' \star [P])} \qquad \textsc{exists-r}\ \frac{H \vdash ([a/x]\, H')}{H \vdash (\exists x.\, H')} \qquad \textsc{forall-r}\ \frac{\forall x.\ (H \vdash H')}{H \vdash (\forall x.\, H')} \qquad \textsc{forall-monotone}\ \frac{\forall x.\ (H \vdash H')}{(\forall x.\, H) \vdash (\forall x.\, H')}$$

Fig. 1. Useful properties for pure facts and quantifiers, with respect to entailment.

The monotonicity rule can be read from bottom to top: when facing a proof obligation of the form $H_1 \star H_2 \vdash H_1' \star H_2$, one may cancel out H_2 on both sides, leaving the proof obligation $H_1 \vdash H_1'$.

Remark 3.10 (Symmetric version of the monotonicity rule). The monotonicity rule is sometimes also presented in its symmetric variant, stated below. It is provably equivalent to STAR-MONOTONE-R.

$$\frac{H_1 \vdash H_1' \qquad H_2 \vdash H_2'}{H_1 \star H_2 \vdash H_1' \star H_2'}\ \textsc{star-monotone}$$

The useful properties associated with pure facts and quantifiers appear in Figure 1. The application of a number of reasoning rules for entailment can be automated by means of a tactic. (One such tactic is described in the appendix.) Other properties may also be derived, such as $([P_1] \star [P_2]) = [P_1 \wedge P_2]$. Yet, when a simplification tactic is available, one does not need to state such properties explicitly.

The entailment relation may be employed to express how a specific piece of information can be extracted from a given heap predicate. For example, from $p \hookrightarrow v$, one can extract the information $p \neq \mathsf{null}$. Likewise, from a heap predicate of the form $p \hookrightarrow v_1 \star p \hookrightarrow v_2$, where the same location p is described twice, one can derive a contradiction, because the separating conjunction asserts disjointness. These two results are formalized as follows.

Lemma 3.11 (Properties of the singleton heap predicate).

$$\begin{array}{ll} \textsc{single-not-null}: & (p \hookrightarrow v) \ \vdash\ (p \hookrightarrow v) \star [p \neq \mathsf{null}] \\ \textsc{single-conflict}: & (p \hookrightarrow v_1) \star (p \hookrightarrow v_2) \ \vdash\ [\mathsf{False}] \end{array}$$

3.4 Generalization to postconditions

In the imperative λ-calculus considered in this paper and formalized further on (§4), a term evaluates to a value. A postcondition thus describes both an output value and an output state.

Definition 3.12 (Type of postconditions). A postcondition has type: $\mathsf{val} \to \mathsf{heap} \to \mathsf{Prop}$.

Thereafter, we let Q range over postconditions. To obtain concise statements of the reasoning rules of Separation Logic for an imperative λ-calculus, it is convenient to extend separating conjunction and entailment to operate on postconditions. To that end, we generalize the predicates $H \star H'$ and $H \vdash H'$ by introducing the predicates $Q \star H'$ and $Q \vdash Q'$, written with a dot to suggest *pointwise extension*. These two predicates are formalized next.

Definition 3.13 (Separating conjunction between a postcondition and a heap predicate).

$$Q \star H \ \equiv\ \lambda v.\ (Q\, v \star H)$$

This operator appears for example in the statement of the frame rule (recall §2.1).

The entailment relation for postconditions is a pointwise extension of the entailment relation for heap predicates: Q entails Q' if and only if, for any value v, the heap predicate $Q\, v$ entails $Q'\, v$.

Definition 3.14 (Entailment between postconditions).

$$Q \models Q' \quad \equiv \quad \forall v.\, (Q\, v \vdash Q'\, v)$$

This entailment defines an order on postconditions. It appears for example in the statement of the consequence rule, which allows strengthening the precondition and weakening the postcondition.

Example 3.15 (Rule of consequence).

$$\frac{H \vdash H' \qquad \{H'\}\ t\ \{Q'\} \qquad Q' \models Q}{\{H\}\ t\ \{Q\}}\ \textsc{consequence}$$

4 LANGUAGE SYNTAX AND SEMANTICS

The definition of triples depends on the details of the programming language. Thus, let us first describe the syntax and the semantics of terms.

4.1 Syntax

We consider an imperative call-by-value λ-calculus. The syntactic categories are primitive functions π, values v, and terms t. The grammar of values is intended to denote *closed* values, that is, values without occurrences of free variables. This design choice leads to a simple substitution function, which may be defined as the identity over all values.

The primitive operations fall in two categories. First, they include the state-manipulating operations for allocating, reading, writing, and deallocating references. Second, they include Boolean and arithmetic operations. For brevity, we include only the addition and division operations.

The values include the unit value tt, boolean literals b, integer literals n, memory locations p, primitive operations π, and recursive functions $\hat{\mu} f.\lambda x.t$. The latter construct is written with a hat symbol to denote the fact this value is closed.

The terms include variables, values, function invocation, sequence, let-bindings, conditionals, and function definitions. The latter construct is written $\mu f.\lambda x.t$, this time without a hat symbol.

Definition 4.1 (Syntax of the language).

$$\begin{array}{lcl} \pi & := & \mathsf{ref} \mid \mathsf{get} \mid \mathsf{set} \mid \mathsf{free} \mid (+) \mid (\div) \\ v & := & tt \mid b \mid n \mid p \mid \pi \mid \hat{\mu} f.\lambda x.t \\ t & := & v \mid x \mid (t\, t) \mid \mathsf{let}\, x = t\, \mathsf{in}\, t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mu f.\lambda x.t \end{array}$$

A non-recursive function $\lambda x.\, t$ may be viewed as a recursive function $\mu f.\lambda x.t$ with a dummy name f. Likewise, a sequence $(t_1\,;\, t_2)$ may be viewed as a let-binding of the form $\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$ for a dummy name x. The Coq formalization actually includes these two constructs explicitly in the grammar to avoid unnecessary complications associated with the elimination of dummy variables.

Although our syntax technically allows for arbitrary terms, for simplicity we assume terms to be written in "administrative normal form" (*A-normal form*), that is, "$\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$" is the sole sequencing construct: no sequencing is implicit in any other construct. For instance, the conditional construct "$\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$" must be encoded as "$\mathsf{let}\, x = t\, \mathsf{in}\, \mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$". This presentation is intended to simplify the statement of the evaluation rules and reasoning rules. Note that many practical program verification tools perform code normalization as a preliminary step.

4.2 Semantics

Thereafter, we use the meta-variable s to denote a variable of type heap that corresponds to a full memory state at a given point in the execution, in contrast to the meta-variable h, which denotes a heap that that may correspond to only a piece of the memory state.

$$\textsc{eval-val}\ \frac{}{v/s \Downarrow v/s} \qquad \textsc{eval-fix}\ \frac{}{(\mu f.\lambda x.t)/s \Downarrow (\hat{\mu} f.\lambda x.t)/s}$$

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

$$\textsc{eval-let}\ \frac{t_1/s \Downarrow v_1/s' \qquad ([v_1/x]\,t_2)/s' \Downarrow v/s''}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)/s \Downarrow v/s''} \qquad \textsc{eval-if}\ \frac{\mathsf{If}\ b\ \mathsf{then}\ (t_1/s \Downarrow v'/s')\ \mathsf{else}\ (t_2/s \Downarrow v'/s')}{(\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow v'/s'}$$

$$\textsc{eval-ref}\ \frac{p \notin \mathrm{dom}\, s}{(\mathsf{ref}\, v)/s \Downarrow p/(s[p := v])} \qquad \textsc{eval-free}\ \frac{p \in \mathrm{dom}\, s}{(\mathsf{free}\, p)/s \Downarrow tt/(s \setminus p)} \qquad \textsc{eval-get}\ \frac{p \in \mathrm{dom}\, s}{(\mathsf{get}\, p)/s \Downarrow (s[p])/s}$$

$$\textsc{eval-set}\ \frac{p \in \mathrm{dom}\, s}{(\mathsf{set}\, p\, v)/s \Downarrow tt/(s[p := v])} \qquad \textsc{eval-add}\ \frac{}{((+)\, n_1\, n_2)/s \Downarrow (n_1 + n_2)/s} \qquad \textsc{eval-div}\ \frac{n_2 \neq 0}{((\div)\, n_1\, n_2)/s \Downarrow (n_1 \div n_2)/s}$$

Fig. 2. Evaluation rules, in big-step style

The semantics of the language is described by the big-step judgment $t/s \Downarrow v/s'$, which asserts that the term t, starting from the state s, evaluates to the value v and the final state s'.

Definition 4.2 (Semantics of the language). The evaluation rules appear in Figure 2.

The rules are standard. A value evaluates to itself. Likewise, a function evaluates to itself. The evaluation rules for function calls and let-bindings involve the standard (capture-avoiding) substitution operation: $[v/x]\,t$ denotes the substitution of x by v throughout the term t. The evaluation rule for conditionals is stated concisely using Coq's conditional construct. The primitive operations on reference cells are described using operations on finite maps: $\mathrm{dom}\, s$ denotes the domain of the state s, the operation $s[p]$ returns the value associated with p, the operation $s \setminus p$ removes the binding on p, and the operation $s[p := v]$ sets or updates a binding from p to v.

5 TRIPLES AND REASONING RULES

5.1 Separation Logic triples

Separation Logic is a refinement of Hoare logic. Interestingly, Separation Logic triples can be defined *in terms of* Hoare triples.

A Hoare triple, written $^{\text{HOARE}}\{H\}\, t\, \{Q\}$, asserts that in any state s satisfying the precondition H, the evaluation of the term t terminates and produces output value v and output state s', as described by the evaluation judgment $t/s \Downarrow v/s'$. Moreover, the output value and output state satisfy the postcondition Q, in the sense that $Q\, v\, s'$ holds. Note that this definition captures termination—it defines a *total correctness* triple.

Definition 5.1 (Total correctness Hoare triple).

$$^{\text{HOARE}}\{H\}\, t\, \{Q\} \quad \equiv \quad \forall s.\ H\, s \Rightarrow \exists v.\ \exists s'.\ (t/s \Downarrow v/s') \wedge (Q\, v\, s')$$

Whereas a Hoare triple describes the evaluation of a term with respect to the whole memory state, a Separation Logic triple describes the the evaluation of a term with respect to only a fragment of the memory state. To relate the two concepts, it suffices to quantify over "the rest of the state", that is, the part of the state that the evaluation of the term is not concerned with.

A Separation Logic triple, written $\{H\}\ t\ \{Q\}$, asserts that, for any heap predicate H' describing the "rest of the state", the Hoare triple $^{\text{HOARE}}\{H \star H'\}\, t\, \{Q \star H'\}$ holds. This formulation effectively

bakes in the frame rule, by asserting from the very beginning that specifications are intended to preserve any resource that is not mentioned in the precondition.

Definition 5.2 (Total correctness Separation Logic triple).

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall H'.\ {}^{\text{HOARE}}\{H \star H'\}\, t\, \{Q \star H'\}$$

To fully grasp the meaning of a Separation Logic triple, it helps to contemplate an alternative definition expressed directly with respect to the evaluation judgment. This alternative definition, shown below, reads as follows: if the input states decomposes as a part h_1 that satisfies the precondition H and a disjoint part h_2 that describes the rest of the state, then the term t terminates on a value v, producing a heap made of a part h_1' and, disjointly, the part h_2 which was unmodified; moreover, the value v and the heap h_1' together satisfy the postcondition Q.

Definition 5.3 (Alternative definition of total correctness Separation Logic triples).

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall h_1.\, \forall h_2.\, \left\{ \begin{array}{l} H\, h_1 \\ h_1 \perp h_2 \end{array} \right. \quad \Rightarrow \quad \exists v.\, \exists h_1'.\, \left\{ \begin{array}{l} h_1' \perp h_2 \\ t/(h_1 \uplus h_2) \Downarrow v/(h_1' \uplus h_2) \\ Q\, v\, h_1' \end{array} \right.$$

The reasoning rules of Separation Logic fall in three categories. First, the structural rules: they do not depend on the details of the language. Second, the reasoning rules for terms: there is one such rule for each term construct of the language. Third, the specification of the primitive operations: there is one such rule for each primitive operation. All these rules are presented next.

5.2 Structural rules

The structural rules of Separation Logic include the consequence rule and the frame rule, which were already discussed, and two rules for extracting pure facts and existential quantifiers out of preconditions. (The role of these rules is illustrated in the example proof presented in the appendix.)

Lemma 5.4 (Structural rules of Separation Logic). The following reasoning rules can be stated as lemmas and proved correct with respect to the interpretation of triples given by Definition 5.2.

$$\textsc{consequence}\ \frac{H \vdash H' \quad \{H'\}\ t\ \{Q'\} \quad Q' \vdash Q}{\{H\}\ t\ \{Q\}} \qquad \textsc{frame}\ \frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \star H'\}} \qquad \textsc{prop}\ \frac{P \Rightarrow \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}} \qquad \textsc{exists}\ \frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\, H\}\ t\ \{Q\}}$$

The frame rule may be exploited in practice as a *forward* reasoning rule: given a triple $\{H\}\ t\ \{Q\}$, one may derive another triple by extending both the precondition and the postcondition with a heap predicate H'. This rule is, however, almost unusable as a *backward* reasoning rule: indeed, it is extremely rare for a proof obligation to be exactly of the form $\{H \star H'\}\ t\ \{Q \star H'\}$. In order to exploit the frame rule in backward reasoning, one usually needs to first invoke the consequence rule. The effect of a combined application of the consequence rule followed with the frame rule is captured by the combined *consequence-frame* rule, stated below.

Lemma 5.5 (Combined consequence-frame rule).

$$\frac{H \vdash H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 \vdash Q}{\{H\}\ t\ \{Q\}}\ \textsc{consequence-frame}$$

This combined rule applies to a proof obligation of the form $\{H\}\ t\ \{Q\}$, with no constraints on the precondition nor the postcondition. To prove this triple from an existing triple $\{H_1\}\ t\ \{Q_1\}$, it suffices to show that the precondition H decomposes as $H_1 \star H_2$, and to show that the postcondition Q can be recovered from $Q_1 \star H_2$. In practice, the "framed" heap predicate H_2 is computed as the difference between H and H_1.

5.3 Rules for terms

The program logic includes one rule for each term construct. The corresponding rules are stated below and explained next.

Lemma 5.6 (Reasoning rules for terms in Separation Logic). The following rules can be stated as lemmas and proved correct with respect to the interpretation of triples given in Definition 5.2.

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

$$\frac{\{H\}\ t_1\ \{\lambda v.\,H'\} \qquad \{H'\}\ t_2\ \{Q\}}{\{H\}\ (t_1\,;\,t_2)\ \{Q\}}\ \textsc{seq} \qquad \frac{\{H\}\ t_1\ \{Q'\} \qquad \forall v.\ \{Q'\,v\}\ ([v/x]\,t_2)\ \{Q\}}{\{H\}\ (\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}\ \textsc{let}$$

$$\frac{b = \mathsf{true} \Rightarrow \{H\}\ t_1\ \{Q\} \qquad b = \mathsf{false} \Rightarrow \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{if}\,b\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2)\ \{Q\}}\ \textsc{if}$$

The rules VAL and FIX apply to terms that correspond to closed values. A value evaluates to itself, without modifying the state. If the heap at hand is described in the precondition by the heap predicate H, then this heap, together with the value v, should satisfy the postcondition. This implication is captured by the premise $H \vdash Q\,v$. Note that the rules VAL and FIX can also be formulated using triples featuring an empty precondition.

Lemma 5.7 (Small-footprint reasoning rules for values).

$$\frac{}{\{\,[\,]\,\}\ v\ \{\lambda r.\,[r = v]\}}\ \textsc{val}' \qquad \frac{}{\{\,[\,]\,\}\ (\mu f.\lambda x.t)\ \{\lambda r.\,[r = (\hat{\mu} f.\lambda x.t)]\}}\ \textsc{fix}'$$

The APP rule merely reformulates the β-reduction rule. It asserts that reasoning about the application of a function to a particular argument amounts to reasoning about the body of this function in which the name of the argument gets substituted with the value of the argument involved in the application. This rule is typically exploited to begin the proof of the specification triple for a function. Once established, such a specification triple may be invoked for reasoning about calls to that function.

The SEQ rule asserts that a sequence "$t_1\,;\,t_2$" admits precondition H and postcondition Q provided that t_1 admits the precondition H and a postcondition describing a heap satisfying H', and that t_2 admits the precondition H' and the postcondition Q. (The result value v produced by t_1 is ignored.)

The LET rule enables reasoning about a let-binding of the form "$\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$". It reads as follows. Assume that, in the current heap described by H, the evaluation of t_1 produces a postcondition Q'. Assume also that, for any value v that the evaluation of t_1 might produce, the evaluation of $[v/x]\,t_2$ in a heap described by $Q'\,v$ produces the postcondition Q. Then, under the precondition H, the term "$\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$" produces the postcondition Q.

The IF rule enables reasoning about a conditional. Its statement features two premises: one for the case where the condition is the value true, and one for the case where it is the value false.

5.4 Specification of primitive operations

The third and last category of reasoning rules corresponds to the specification of the primitive operations of the language. The operations on references have already been discussed (§2.5 and §2.8). The arithmetic operations admit specifications that involve only empty heaps.

Lemma 5.8 (Specification for primitive operations).

$$
\begin{array}{lll}
\textsc{ref}: & \{[\,]\}\ \ (\texttt{ref}\,v) & \{\lambda r.\ \exists p.\ [r = p] \star (p \hookrightarrow v)\} \\
\textsc{get}: & \{p \hookrightarrow v\}\ \ (\texttt{get}\,p) & \{\lambda r.\ [r = v] \star (p \hookrightarrow v)\} \\
\textsc{set}: & \{p \hookrightarrow v\}\ \ (\texttt{set}\,p\,v') & \{\lambda \_.\ (p \hookrightarrow v')\} \\
\textsc{free}: & \{p \hookrightarrow v\}\ \ (\texttt{free}\,p) & \{\lambda \_.\ [\,]\} \\
\textsc{add}: & \{[\,]\}\ ((+)\,n_1\,n_2) & \{\lambda r.\ [r = n_1 + n_2]\} \\
\textsc{div}: \quad n_2 \neq 0 \ \Rightarrow & \{[\,]\}\ ((\div)\,n_1\,n_2) & \{\lambda r.\ [r = n_1 \div n_2]\}
\end{array}
$$

This completes the presentation of the reasoning rules of Separation Logic. Technically, these 18 reasoning rules suffice to verify imperative programs, although additional infrastructure helps obtain more concise proof scripts. The Coq formalization of the material from Sections §3, §4, and §5 amount to 564 non-blank lines of Coq script. It includes 23 definitions, 58 lemmas, 24 lines of tactic definitions, and 116 lines of proofs. We encourage the reader to check out the corresponding file, which is called SLFMinimal.v in the supplementary material [Anonymous 2020].

6 INDUCTIVE REASONING FOR LOOPS

Pointer-manipulating programs are typically written using loops. Although loops can be simulated using recursive functions, it simplifies the proofs to include direct reasoning rules for them. Let us assume in this section the presence of a while-loop construct, written "while t_1 do t_2".

A loop "while t_1 do t_2" is equivalent to its one-step unfolding: if t_1 evaluates to true, then t_2 is executed and the loop proceeds; otherwise the loop terminates on the unit value. The rules EVAL-WHILE and WHILE shown below capture this one-step unfolding principle.

Lemma 6.1 (Evaluation rule and reasoning rules for while loops).

$$
\textsc{eval-while}\ \frac{(\text{if } t_1 \text{ then } (t_2\,;\ \text{while } t_1 \text{ do } t_2) \text{ else } tt)/s \Downarrow v/s'}{(\text{while } t_1 \text{ do } t_2)/s \Downarrow v/s'}
\qquad
\textsc{while}\ \frac{\{H\}\ (\text{if } t_1 \text{ then } (t_2\,;\ \text{while } t_1 \text{ do } t_2) \text{ else } tt)\ \{Q\}}{\{H\}\ (\text{while } t_1 \text{ do } t_2)\ \{Q\}}
$$

One may establish a triple about the behavior of a while loop by conducting a proof by induction over a decreasing measure or well-founded relation, exploiting the induction hypothesis to reason about the "remaining iterations". Note that this approach is essentially equivalent to encoding the loop as a tail-recursive function, yet without the boilerplate associated with an encoding.

Example 6.2 (Length of a list using a while loop). Consider the following code fragment, which sets the contents of `s` to the length of the mutable list at location `p`.

```
let r = ref p and s = ref 0 in
while !r != null do (incr s; r := !r.tail) done
```

The triple $\{\text{Mlist}\,L\,p \star r \hookrightarrow p \star s \hookrightarrow 0\}$ (`while ... done`) $\{\lambda \_.\ \text{Mlist}\,L\,p \star r \hookrightarrow \text{null} \star s \hookrightarrow |L|\}$ specifies the behavior of the loop. Its proof is conducted by induction on the statement: $\forall L n p.\ \{\text{Mlist}\,L\,p \star r \hookrightarrow p \star s \hookrightarrow n\}$ (`while ... done`) $\{\lambda \_.\ \text{Mlist}\,L\,p \star r \hookrightarrow \text{null} \star s \hookrightarrow n + |L|\}$. Applying the WHILE rule reveals the conditional on whether `!r` is null. In the case where it is not null, `s` is incremented, `r` is set to the tail of the current list, and the loop starts over. To reason about this "recursive invocation" of the while-loop, it suffices to apply the frame rule to put aside the head cell described by a predicate of the form $(p.\text{head} \hookrightarrow x) \star (p.\text{tail} \hookrightarrow q)$, and to apply the induction hypothesis to the tail of the list described by $\text{Mlist}\,L'\,q$, where $L = x :: L'$.

The above example shows that, by carrying a proof by induction, it is possible to apply the frame rule over the remaining iterations of a loop. Doing so would not be possible with a reasoning rule

that imposes a loop invariant to be valid both at the entry point and exit point of the loop body. Indeed, such a loop invariant would necessarily involve the description of a list segment.

For-loops may be handled in a similar fashion.

7 THE MAGIC WAND OPERATOR

7.1 Definition and properties of the magic wand

The magic wand is an additional heap predicate operator, written $H_1 \mathrel{-\!\!\star} H_2$, and read "$H_1$ wand H_2". Although it is technically possible to carry out all Separation Logic proofs without the magic wand, this operator helps to state several reasoning rules and specifications more concisely.

Intuitively, $H_1 \mathrel{-\!\!\star} H_2$ defines a heap predicate such that, if starred with H_1, it produces H_2. In other words, the magic wand satisfies the *cancellation rule* $H_1 \star (H_1 \mathrel{-\!\!\star} H_2) \vdash H_2$. The magic wand operator can be formally defined in at least four different ways.

Definition 7.1 (Magic wand). The magic wand operator is equivalently characterized by:

(1) $H_1 \mathrel{-\!\!\star} H_2 \;\equiv\; \lambda h.\ \big(\forall h'.\ h \perp h' \wedge H_1\, h' \;\Rightarrow\; H_2\,(h \uplus h')\big)$

(2) $H_1 \mathrel{-\!\!\star} H_2 \;\equiv\; \exists H_0.\ H_0 \star \big[(H_1 \star H_0) \vdash H_2\big]$

(3) $H_0 \vdash (H_1 \mathrel{-\!\!\star} H_2) \;\Leftrightarrow\; (H_1 \star H_0) \vdash H_2$

(4) $H_1 \mathrel{-\!\!\star} H_2$ satisfies the following introduction and elimination rules.

$$\frac{(H_1 \star H_0) \vdash H_2}{H_0 \vdash (H_1 \mathrel{-\!\!\star} H_2)}\ \textsc{wand-intro} \qquad \frac{}{H_1 \star (H_1 \mathrel{-\!\!\star} H_2) \vdash H_2}\ \textsc{wand-cancel}$$

The first characterization asserts that $H_1 \mathrel{-\!\!\star} H_2$ holds of a heap h if and only if, for any disjoint heap h' satisfying H_1, the union of the two heaps $h \uplus h'$ satisfies H_2.

The second characterization describes a heap satisfying a predicate H_0 that, when starred with H_1 entails H_2. This characterization shows that the magic wand can be encoded using previously-introduced concepts from higher-order Separation Logic.

The third characterization consists of an equivalence that provides both an introduction rule and an elimination rule. The left-to-right direction is equivalent to the cancellation rule WAND-CANCEL stated in definition (4). The right-to-left direction corresponds exactly to the introduction rule from definition (4), namely WAND-INTRO, which reads as follows: to show that a heap described by H_0 satisfies the magic wand $H_1 \star H_2$, it suffices to prove that H_1 starred with H_0 entails H_2.

In practice, to work with the magic wand, the following properties are useful.

Lemma 7.2 (Useful properties of the magic wand).

$$\textsc{wand-monotone}\ \frac{H_1' \vdash H_1 \qquad H_2 \vdash H_2'}{(H_1 \mathrel{-\!\!\star} H_2) \vdash (H_1' \mathrel{-\!\!\star} H_2')} \qquad \textsc{wand-self}\ \frac{}{[\,] \vdash (H \mathrel{-\!\!\star} H)} \qquad \textsc{wand-pure-l}\ \frac{P}{([P] \mathrel{-\!\!\star} H) = H}$$

$$\textsc{wand-curry}\ \frac{}{((H_1 \star H_2) \mathrel{-\!\!\star} H_3) = (H_1 \mathrel{-\!\!\star} (H_2 \mathrel{-\!\!\star} H_3))} \qquad \textsc{wand-star}\ \frac{}{((H_1 \mathrel{-\!\!\star} H_2) \star H_3) \vdash (H_1 \mathrel{-\!\!\star} (H_2 \star H_3))}$$

Lemma 7.3 (Partial concellation of a magic wand). If the left-hand side of a magic wand involves the separating conjunction of several heap predicates, it is possible to cancel out just one of them with an occurrence of the same heap predicate occurring outside of the magic wand. For example, the entailment $H_2 \star \big((H_1 \star H_2 \star H_3) \mathrel{-\!\!\star} H_4\big) \vdash \big((H_1 \star H_3) \mathrel{-\!\!\star} H_4\big)$ is obtained by cancelling H_2.

7.2 Magic wand for postconditions

Just as useful as the magic wand is its generalization to postconditions, which is involved for example in the statement of the ramified frame rule (§7.3). This operator, written $Q_1 \mathrel{-\!\!\star} Q_2$, takes as argument two postconditions Q_1 and Q_2 and produces a heap predicate.

Definition 7.4 (Magic wand for postconditions). The operator $(\mathrel{-\!\!\star})$ is equivalently defined by:

(1) $\quad Q_1 \mathrel{-\!\!\star} Q_2 \;\equiv\; \forall v.\ \big((Q_1\,v) \mathrel{-\!\!\star} (Q_2\,v)\big)$

(2) $\quad Q_1 \mathrel{-\!\!\star} Q_2 \;\equiv\; \lambda h.\ \big(\forall v h'.\ h \perp h' \wedge Q_1\,v\,h' \Rightarrow Q_2\,v\,(h \uplus h')\big)$

(3) $\quad Q_1 \mathrel{-\!\!\star} Q_2 \;\equiv\; \exists H_0.\ H_0 \star \big[(Q_1 \mathbin{\dot\star} H_0) \vdash Q_2\big]$

(4) $\quad H_0 \vdash (Q_1 \mathrel{-\!\!\star} Q_2) \;\Leftrightarrow\; (Q_1 \mathbin{\dot\star} H_0) \vdash Q_2$

(5) $\quad Q_1 \mathrel{-\!\!\star} Q_2$ satisfies the following introduction and elimination rules.

$$\frac{(Q_1 \mathbin{\dot\star} H_0) \vdash Q_2}{H_0 \vdash (Q_1 \mathrel{-\!\!\star} Q_2)}\ \textsc{qwand-intro} \qquad \frac{}{Q_1 \mathbin{\dot\star} (Q_1 \mathrel{-\!\!\star} Q_2) \vdash Q_2}\ \textsc{qwand-cancel}$$

Lemma 7.5 (Useful properties of the magic wand for postconditions).

$$\textsc{qwand-monotone}\ \frac{Q_1' \vdash Q_1 \qquad Q_2 \vdash Q_2'}{(Q_1 \mathrel{-\!\!\star} Q_2) \vdash (Q_1' \mathrel{-\!\!\star} Q_2')} \qquad \textsc{qwand-self}\ \frac{}{[\,] \vdash (Q \mathrel{-\!\!\star} Q)}$$

$$\textsc{qwand-star}\ \frac{}{((Q_1 \mathrel{-\!\!\star} Q_2) \mathbin{\dot\star} H) \vdash (Q_1 \mathrel{-\!\!\star} (Q_2 \star H))} \qquad \textsc{qwand-specialize}\ \frac{}{\big(Q_1 \mathrel{-\!\!\star} Q_2\big) \vdash \big((Q_1\,v) \mathrel{-\!\!\star} (Q_2\,v)\big)}$$

7.3 Ramified frame rule

One key practical application of the magic wand operator appears in the statement of the *ramified frame rule*. This rule reformulates the consequence-frame rule in a manner that is both more concise and better-suited for automated processing. Recall the rule CONSEQUENCE-FRAME, which is reproduced below. To exploit it, one must provide a predicate H_2 describing the "framed" part. Providing the heap predicate H_2 by hand in proofs involves a prohibitive amount of work; it is strongly desirable that H_2 may be inferred automatically.

The predicate H_2 can be computed as the difference between H and H_1. Automatically computing this difference is relatively straightforward in simple cases, however this task becomes quite challenging when H and H_1 involve numerous quantifiers. Indeed, it is not obvious to determine which quantifiers from H should be cancelled against those from H_1, and which quantifiers should be carried over to H_2.

The benefit of the ramified frame rule is that it eliminates the problem altogether. The key idea is to observe that the premise $Q_1 \mathbin{\dot\star} H_2 \vdash Q$ from the CONSEQUENCE-FRAME rule is equivalent to $H_2 \vdash (Q_1 \mathrel{-\!\!\star} Q)$, by the 4th characterization of Definition 7.4. Thus, in the other premise $H \vdash H_1 \star H_2$, the heap predicate H_2 may be replaced with $Q_1 \mathrel{-\!\!\star} Q$. The RAMIFIED-FRAME rule appears below.

Lemma 7.6 (Ramified frame rule). RAMIFIED-FRAME reformulates CONSEQUENCE-FRAME.

$$\textsc{consequence-frame}\ \frac{H \vdash H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \mathbin{\dot\star} H_2 \vdash Q}{\{H\}\ t\ \{Q\}} \qquad \textsc{ramified-frame}\ \frac{\{H_1\}\ t\ \{Q_1\} \qquad H \vdash H_1 \star (Q_1 \mathrel{-\!\!\star} Q)}{\{H\}\ t\ \{Q\}}$$

8 AFFINE SEPARATION LOGIC

8.1 Affine heap predicates

The Separation Logic presented so far is well-suited for a language with explicit deallocation, however it is impractical for a language equipped with a garbage collector. Indeed, it does not provide any rule for discarding the description of pieces of state that are ready for the garbage collector to dispose of. The contents of this section explains how to refine the definition of Separation Logic triples so as to support rules that enable discarding heap predicates from either the precondition or the postcondition.

The formalization presented is general enough to allow fine-tuning of which heap predicates may be freely discarded, and which heap predicates should, on the contrary, remain treated *linearly*. For example, linearity is useful to ensure that every file handle opened eventually gets closed, or to ensure that every lock acquired eventually gets released. To that end, the reasoning rules should not allow discarding the heap predicates that represent these resources.

The predicate affine H asserts that the heap predicate H may be freely discarded. This predicate is defined in terms of a lower-level predicate, written haffine h, that characterizes which heaps may be discarded. This predicate is axiomatized. Two specific instantiations are presented: one that treats all heaps as discardable, leading to a *fully-affine* logic, and one that treats none of them as discardable, leading to a *fully-linear* logic, equivalent to the logic developed so far.

Definition 8.1 (Axiomatization of affine heaps). The predicate haffine h must satisfy two rules:

$$\frac{}{\mathsf{haffine}\ \varnothing}\ \textsc{staffine-empty} \qquad \frac{\mathsf{haffine}\ h_1 \qquad \mathsf{haffine}\ h_2 \qquad h_1 \perp h_2}{\mathsf{haffine}\ (h_1 \uplus h_2)}\ \textsc{staffine-union}$$

The predicate affine H characterizes heap predicates that hold only of affine states.

Definition 8.2 (Definition of affine heap predicates).

$$\mathsf{affine}\ H \quad \equiv \quad \forall h.\ H\,h \Rightarrow \mathsf{haffine}\ h$$

The rules presented next establish that the predicate affine distributes well over all operators: heap predicates obtained by composing affine heap predicates are themselves affine.

Lemma 8.3 (Sufficient conditions for affinity of a heap predicate).

$$\frac{}{\mathsf{affine}\ [\,]}\ \textsc{affine-empty} \qquad \frac{}{\mathsf{affine}\ [P]}\ \textsc{affine-pure} \qquad \frac{\mathsf{affine}\ H_1 \qquad \mathsf{affine}\ H_2}{\mathsf{affine}\ (H_1 \star H_2)}\ \textsc{affine-star}$$

$$\frac{\forall x.\ \mathsf{affine}\ H}{\mathsf{affine}\ (\exists x.\ H)}\ \textsc{affine-exists} \qquad \frac{\forall x.\ \mathsf{affine}\ H \qquad \text{the type of } x \text{ is inhabited}}{\mathsf{affine}\ (\forall x.\ H)}\ \textsc{affine-forall} \qquad \frac{P \Rightarrow \mathsf{affine}\ H}{\mathsf{affine}\ ([P] \star H)}\ \textsc{affine-star-pure}$$

To state the reasoning rules that enable discarding affine heap predicates, it is helpful to introduce the *affine top* heap predicate, which is written $\mathbb{T}$. Whereas the top heap predicate (written $\top$ and defined as "$\lambda h.\ \mathsf{true}$") holds of *any heap*, the affine top predicate holds only of *any affine heap*.

Definition 8.4 (Affine top). The predicate $\mathbb{T}$ can be equivalently defined in two ways.

$$(1) \quad \mathbb{T} \ \equiv\ \lambda h.\ \mathsf{haffine}\ h \qquad\qquad (2) \quad \mathbb{T} \ \equiv\ \exists H.\ [\mathsf{affine}\ H] \star H$$

There are three important properties of $\mathbb{T}$. The first one asserts that any affine heap predicate H entails $\mathbb{T}$. The second one asserts that the predicate $\mathbb{T}$ is itself affine. The third one asserts that several copies of $\mathbb{T}$ are equivalent to a single $\mathbb{T}$.

Lemma 8.5 (Properties of affine top).

$$\frac{\text{affine}\, H}{H \vdash \mathbb{T}}\ \textsc{atop-r} \qquad \frac{}{\text{affine}\, \mathbb{T}}\ \textsc{affine-atop} \qquad \frac{}{(\mathbb{T} \star \mathbb{T}) = \mathbb{T}}\ \textsc{star-atop-atop}$$

All the aforementioned definitions and lemmas hold for any predicate haffine satisfying the axiomatization from Definition 8.1. Two extreme instantiations of haffine are particularly interesting.

Example 8.6 (Fully-affine Separation Logic). The definition "haffine $h \equiv$ True" satisfies the requirements of Definition 8.1, and leads to a Separation Logic where all heap predicates may be freely discarded. In that setting, $(\text{affine}\, H) \Leftrightarrow \text{True}$, and $\mathbb{T} = \top = (\lambda h.\, \text{true}) = (\exists H.\, H)$.

Example 8.7 (Fully-linear Separation Logic). The definition "haffine $h \equiv (h = \varnothing)$" satisfies the requirements of Definition 8.1, and leads to a Separation Logic where no heap predicate may be freely discarded. In that setting, $(\text{affine}\, H) \Leftrightarrow (H \vdash [\,])$, and $\mathbb{T} = [\,] = (\lambda h.\, h = \varnothing)$.

8.2 Affine triples

To accommodate reasoning rules that enable freely discarding affine heap predicates, it suffices to refine the definition of a Separation Logic triple (Definition 5.2) by integrating the affine top predicate $\mathbb{T}$ into the postcondition of the underlying Hoare triple, as formalized next.

Definition 8.8 (Refined definition of triples for Separation Logic).

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall H'.\ {}^{\text{HOARE}}\{H \star H'\}\ t\ \{Q \star H' \star \mathbb{T}\}$$

Note that, with the fully-linear instantiation described in Example 8.7, the predicate $\mathbb{T}$ is equivalent to the empty heap predicate, therefore Definition 8.8 is strictly more general than Definition 5.2.

Lemma 8.9 (Reasoning rules for refined Separation Logic triples). All the previously-mentioned reasoning rules, in particular the structural rules (Lemma 5.4) and the reasoning rules for terms (Lemma 5.6), remain correct with respect to the refined definition of triples (Definition 8.8).

The *discard rules*, which enable discarding affine heap predicates, may be stated in a number of ways. The variants that are most useful in practice are shown below. The rule DISCARD-PRE allows discarding a predicate H' from the precondition, provided that H' is affine. The rule ATOP-POST allows extending the postcondition with $\mathbb{T}$, allowing a subsequent proof step to yield an entailment relation of the form $Q_1 \vdash (Q \star \mathbb{T})$, allowing to discard unwanted pieces from Q_1. The rule RAMIFIED-FRAME-ATOP extends the ramified frame rule so that its entailment integrates the predicate $\mathbb{T}$, allowing to discard unwanted pieces from either H or Q_1. These three rules have equivalent expressive power with respect to discarding heap predicates.

Lemma 8.10 (Discard rules).

$$\textsc{discard-pre}\ \frac{\{H\}\ t\ \{Q\} \qquad \text{affine}\, H'}{\{H \star H'\}\ t\ \{Q\}} \qquad \textsc{atop-post}\ \frac{\{H\}\ t\ \{Q \star \mathbb{T}\}}{\{H\}\ t\ \{Q\}} \qquad \textsc{ramified-frame-atop}\ \frac{\{H_1\}\ t\ \{Q_1\} \qquad H \vdash H_1 \star (Q_1 \mathrel{-\!\!\star} (Q \star \mathbb{T}))}{\{H\}\ t\ \{Q\}}$$

9 WEAKEST-PRECONDITION STYLE

9.1 Semantic weakest precondition

The notion of weakest precondition has been used pervasively in the development of practical tools based on Hoare logic. Recent work has shown that this notion also helps streamlining the set up of practical tools based on Separation Logic.

The *semantic weakest precondition* of a term t with respect to a postcondition Q denotes a heap predicate, written $\text{wp}\, t\, Q$, which corresponds to the *weakest* precondition H satisfying the triple

$\{H\}\ t\ \{Q\}$. The notion of "weakest" is to be understood with respect to the entailment relation, which induces an order relation on the set of heap predicates (recall Lemma 3.8). The definition of the predicate wp can be formalized in at least five different ways. The corresponding definitions are shown below and commented next.

Definition 9.1 (Semantic weakest precondition). The predicate wp is equivalently characterized by:

(1) $\quad \mathrm{wp}\, t\, Q \ \equiv\ \min_{(\vdash)} \left\{ H \;\middle|\; \{H\}\ t\ \{Q\} \right\}$

(2) $\quad \Big(\{\mathrm{wp}\, t\, Q\}\ t\ \{Q\}\Big) \ \wedge\ \Big(\forall H.\ \{H\}\ t\ \{Q\} \ \Rightarrow\ H \vdash \mathrm{wp}\, t\, Q\Big)$

(3) $\quad \mathrm{wp}\, t\, Q \ \equiv\ \lambda h.\ \Big(\{\lambda h'.\, h' = h\}\ t\ \{Q\}\Big)$

(4) $\quad \mathrm{wp}\, t\, Q \ \equiv\ \exists\!\!\exists H.\ H \star \Big[\{H\}\ t\ \{Q\}\Big]$

(5) $\quad H \vdash \mathrm{wp}\, t\, Q \ \Leftrightarrow\ \{H\}\ t\ \{Q\}$

The first and second characterization asserts that $\mathrm{wp}\, t\, Q$ is *the weakest precondition*: it is a valid precondition for a triple for the term t with the postcondition Q. Moreover, any other valid precondition H for a triple involving t and Q entails $\mathrm{wp}\, t\, Q$.

The third characterization defines $\mathrm{wp}\, t\, Q$ as a predicate over a heap h, asserting that $\mathrm{wp}\, t\, Q$ holds of the heap h if and only if the evaluation of the term starting from a heap *equal to* h produces the postcondition Q.

The fourth characterization asserts that $\mathrm{wp}\, t\, Q$ is entailed by any heap predicate H satisfying the triple $\{H\}\ t\ \{Q\}$. This characterization shows that the notion of weakest precondition can be expressed as a derived notion in terms of the core heap predicate operators.

The fifth characterization asserts that any triple of the form $\{H\}\ t\ \{Q\}$ may be equivalently reformulated by replacing this triple with $H \vdash \mathrm{wp}\, t\, Q$.

9.2 WP-style structural rules

The structural reasoning rule can be reformulated in weakest-precondition style, as follows.

Lemma 9.2 (Structural rules in weakest precondition style).

$$\frac{Q \vdash Q'}{\mathrm{wp}\, t\, Q \ \vdash\ \mathrm{wp}\, t\, Q'}\ \textsc{wp-consequence} \qquad \frac{}{(\mathrm{wp}\, t\, Q) \star H \ \vdash\ \mathrm{wp}\, t\, (Q \star H)}\ \textsc{wp-frame}$$

$$\frac{\mathrm{affine}\, H}{(\mathrm{wp}\, t\, Q) \star H \ \vdash\ (\mathrm{wp}\, t\, Q)}\ \textsc{wp-discard-pre} \qquad \frac{}{\mathrm{wp}\, t\, (Q \star \top) \ \vdash\ \mathrm{wp}\, t\, Q}\ \textsc{wp-atop-post}$$

The rule WP-CONSEQUENCE captures a monotonicity property. The rule WP-FRAME reads as follows: *if I own a heap in which the execution of t produces the postcondition Q, and, separately, I own a heap satisfying H, then, altogether, I own a heap in which the execution of t produces both Q and H.* These four structural rules may be combined into a single rule, called WP-RAMIFIED-FRAME-ATOP, which subsumes all the other structural rules of Separation Logic.

Lemma 9.3 (Ramified frame rule in weakest precondition style).

$$\frac{}{(\mathrm{wp}\, t\, Q) \ \star\ (Q \mathrel{-\!\!\star}\ (Q' \star \top)) \ \vdash\ (\mathrm{wp}\, t\, Q')}\ \textsc{wp-ramified-frame-atop}$$

9.3 WP-style rules for terms

The weakest-precondition style reformulation of the reasoning rules for terms yields rules that are similar to the corresponding Hoare logic rules. For example, the rule for sequence is as follows.

$$\frac{}{\mathrm{wp}\, t_1\, (\lambda v.\ \mathrm{wp}\, t_2\, Q) \ \vdash\ \mathrm{wp}\, (t_1\,;\, t_2)\, Q}\ \textsc{wp-seq}$$

This rule can be read as follows: *if I own a heap in which the execution of t_1 produces a heap in which the execution of t_2 produces the postcondition Q, then I own a heap in which the execution of the sequence "t_1 ; t_2" produces Q*. The other reasoning rules for terms appear below.

Lemma 9.4 (Reasoning rules for terms in weakest precondition style).

$$\textsc{wp-val}\ \frac{}{Q\,v \ \vdash\ \mathsf{wp}\,v\,Q} \qquad \textsc{wp-fix}\ \frac{}{Q\,(\hat{\mu}f.\lambda x.t) \ \vdash\ \mathsf{wp}\,(\mu f.\lambda x.t)\,Q} \qquad \textsc{wp-app}\ \frac{v_1 = \hat{\mu}f.\lambda x.t}{\mathsf{wp}\,([v_2/x]\,[v_1/f]\,t)\,Q \ \vdash\ \mathsf{wp}\,(v_1\,v_2)\,Q}$$

$$\frac{}{\mathsf{wp}\,t_1\,(\lambda v.\ \mathsf{wp}\,([v/x]\,t_2)\,Q) \ \vdash\ \mathsf{wp}\,(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\,Q}\ \textsc{wp-let}$$

$$\frac{}{\mathsf{If}\ b\ \mathsf{then}\ (\mathsf{wp}\,t_1\,Q)\ \mathsf{else}\ (\mathsf{wp}\,t_2\,Q) \ \vdash\ \mathsf{wp}\,(\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\,Q}\ \textsc{wp-if}$$

9.4 WP-style function specifications

Function specifications were so far expressed using triples of the form $\{H\}\ (f\,v)\ \{Q\}$. These specifications may be equivalently expressed using assertions of the form $H \vdash \mathsf{wp}\,(f\,v)\,Q$.

The primitive operations are specified using wp as shown below. For example, the allocation operation $\mathsf{ref}\,v$ produces a postcondition Q, provided that the result of extending the current precondition with $p \hookrightarrow v$ yields $Q\,p$. In the formal statement of the specification WP-REF, observe how the address p is quantified universally in the left-hand side of the entailment.

Lemma 9.5 (Specification of primitive operations in weakest-precondition style).

$$\begin{array}{lll}
\textsc{wp-ref}: & \forall\,Q\,v. & \big(\forall p.\ (p \hookrightarrow v) \mathrel{-\!\!\star} (Q\,p)\big) \vdash \mathsf{wp}\,(\mathsf{ref}\,v)\,Q \\
\textsc{wp-get}: & \forall\,Q\,p. & (p \hookrightarrow v) \star \big((p \hookrightarrow v) \mathrel{-\!\!\star} (Q\,v)\big) \vdash \mathsf{wp}\,(\mathsf{get}\,p)\,Q \\
\textsc{wp-set}: & \forall\,Q\,p\,v\,v'. & (p \hookrightarrow v) \star \big(\forall r.\ (p \hookrightarrow v') \mathrel{-\!\!\star} (Q\,r)\big) \vdash \mathsf{wp}\,(\mathsf{set}\,p\,v')\,Q \\
\textsc{wp-free}: & \forall\,Q\,p\,v. & (p \hookrightarrow v) \star \big(\forall r.\ (Q\,r)\big) \vdash \mathsf{wp}\,(\mathsf{free}\,p)\,Q
\end{array}$$

Remark: WP-SET and WP-FREE can also be stated by specializing the variable r to the unit value $\vec{tt}$.

There exists a general pattern for translating from conventional triples to weakest-precondition style specifications. The following lemma covers the case of a specification involving a single auxiliary variable named x. It may easily be generalized to a larger number of auxiliary variables.

Lemma 9.6 (Specifications in weakest-precondition style). Let v denote a value that may depend on a variable x, and let H' denote a heap predicate that may depend on the variables x and r.

$$\big(\{H\}\ t\ \{\lambda r.\ \exists x.\ [r = v] \star H'\}\big) \ \Leftrightarrow\ \big(\forall Q.\ H \star (\forall x.\ H' \mathrel{-\!\!\star} (Q\,v)) \ \vdash\ \mathsf{wp}\,t\,Q\big)$$

Stating specifications in weakest-precondition style is not at all mandatory for working with reasoning rules in weakest-precondition style. Indeed, one may continue stating specifications using conventional triples, which one might find more intuitive to read, and exploit the following rule for reasoning about function applications.

Lemma 9.7 (Variant of the ramified frame rule for proof obligations in weakest-precondition style).

$$\frac{\{H_1\}\ t\ \{Q_1\} \qquad H \ \vdash\ H_1 \star (Q_1 \mathrel{-\!\!\star} Q_2)}{H \ \vdash\ \mathsf{wp}\,t\,Q}\ \textsc{ramified-frame-for-wp}$$

10 RELATED WORK

For a broad survey of Separation Logic, we refer to O'Hearn's CACM paper [2019], in particular its appendix, which surveys tools based on Separation Logic. In this section, we give references for the ideas presented in this paper, discuss practical verification tools that leverage Separation Logic in interactive proofs assistants, and compare with other teaching material for Separation Logic.

Original presentation of Separation Logic. Traditional presentations of Separation Logic target command-based languages, which involve mutable variables in addition to heap-allocated data. In that setting, the statement of the frame rule involves a side-condition to assert that the mutable variables occurring in the framed heap predicate are not modified by the command. Up to minor differences in presentation, many fundamental concepts described in the present paper already appeared in the first descriptions of Separation Logic [O'Hearn et al. 2001; Reynolds 2002], including:

- the grammar of heap predicate operators, except the pure heap predicate [P], and with the limitation that quantifiers $\exists x.\, H$ and $\forall x.\, H$ range only over integer values;
- the rule of consequence and the frame rules;
- a variant of the rule EXISTS (this variant is named EXISTS2 in the appendix);
- the fundamental properties of the star operator described in Lemma 3.9;
- the small footprint specifications for primitive state-manipulating operations,
- the definition of Mlist, stated by pattern matching over the list structure like in Definition 2.7;
- the characterization of the magic wand operator via characterizations (1), (3) and (4) from Definition 7.1, but not characterization (2), which involves quantification over heap predicates;
- the example of a copy function for binary trees, generalizing the copy function for lists described in the appendix;
- the encoding of records and arrays using pointer arithmetics (described in the appendix).

Mechanized presentations. Gordon [1989] proposed the first mechanization of Hoare logic in higher-order logic, more precisely in the HOL proof assistant. Gordon's pioneering work was followed by numerous formalizations of Hoare logic, targeting various programming languages. Mechanizations of Separation Logic appeared later. They include: work by Marti et al. [2006] in Coq; work by Tuch et al. [2007] in Isabelle/HOL, as part of the Sel4 verification project; work by Appel and Blazy [2007] in Coq, at the origins of the VST tool [Appel 2011; Appel et al. 2014]; work by Myreen and Gordon [2007] in HOL4, eventually leading to the CakeML compiler [Kumar et al. 2014]; work by Chlipala [2011, 2013] in Coq, as part of the Bedrock framework.

Fewer presentations of Separation Logic target languages based on an imperative λ-calculus, where terms produce values (a minor complication) and where local variables are immutable (a major simplification). Birkedal et al. [2006] present a Separation Logic for Idealized Algol extended with heaps. Their theory, developed on paper, formalizes reasoning rules in the form of a dependent type theory: the triple $\{H\}\ t\ \{Q\}$ is expressed by the fact that the term t admits the type "$\{H\} \cdot \{Q\}$". This work was subsequently generalized into Hoare Type Theory (HTT) [Nanevski et al. 2006, 2008] to allow the heap to contain not just integers but structured values, including first-class functions—that is, supporting higher-order stores. Hoare Type Theory was then axiomatically embedded in Coq, resulting in the Ynot tool [Chlipala et al. 2009].

Ynot introduces the bracket notation $[P]$ for pure predicates, and defines entailment ($H \vdash H'$) as in the present paper. However, it does not assume predicate extensionality, hence properties of separating conjunction are expressed using morphisms. In Ynot, programs are shallowly embedded in Coq: they are expressed using Coq primitive constructs and axiomatized monadic constructs for effects. In Ynot, a Coq term t admits the Coq type "ST H Q" to express the specification $\{H\}\ t\ \{Q\}$. The frame rule takes the form of an identity coercion, which updates the type of a term from

ST $H\,Q$ to ST $(H \star H')\,(\lambda v.\ Q\,v \star H')$. The dependent-type presentation of specifications makes it essentially impossible to use auxiliary variables that range over both H and Q. To circumvent the problem, the precondition is implicitly repeated in the postcondition, allowing to relate the two.

The CFML tool [Charguéraud 2011] targets the verification of OCaml programs. CFML does not state reasoning rules directly in Coq; instead, a program is verified by means of its *characteristic formula*, which corresponds to a form of strongest postcondition. These characteristic formulae are generated as Coq axioms by an external tool that parses input programs in OCaml syntax. CFML includes a formalization of heap predicates similar to that of Ynot, and adds the operators $Q \star H$ and $Q \vdash Q'$ for postconditions (§3.4). It includes the treatment of while loops with the possibility to frame over the remaining iterations (§6). This possibility was independently discovered at the same time by Tuerk [2010]. CFML initially hard-wired fully-affine triples, featuring unrestricted discard rules (§8.2). It was subsequently refined to integrate the customizable predicate haffine (§8) [Guéneau et al. 2019].

The CakeML verified compiler [Kumar et al. 2014] targets SML programs and produces machine code. It is implemented in HOL and includes a Separation Logic used in particular for the formal verification of pieces of the runtime system. CakeML's machinery computes, within HOL, characteristic formulae. It follows CFML definitions, extending them with support for catchable exceptions [Guéneau et al. 2017].

The Iris framework [Jung et al. 2017, 2018; Krebbers et al. 2017], implemented in Coq, supports higher-order concurrent Separation Logic. Iris features a grammar of heap predicates that includes two modalities: the *later modality*, which is used in particular to state reasoning principles akin to coinduction, and the *persistent modality*, which is used in particular to express duplicable assertions. The core operators are realized using *step-indexed* definitions: a heap predicate depends not only on a heap but also on a natural number, which is used in particular for interpreting the *later modality*. Iris is currently restricted to a fully-affine logic, with an affine entailment. It exploits weakest-precondition style reasoning rules (§9) and function specifications are stated as in Lemma 9.6, although using syntactic sugar to make specifications resemble conventional triples.

Other contributions to Separation Logic. We next discuss the origins of three technical ingredients.

The interpretation of a Separation Logic triple that bakes in the frame rule by quantifying over a heap predicate describing the rest of the state like in Definition 5.2 seem to have first appeared in the *Views* paper by Dinsdale-Young et al. [2013].

The ramified frame rule stated in Lemma 7.6 was introduced by Hobor and Villard [2013]. It is used in particular in VST [Appel et al. 2014; Qinxiang Cao and Appel 2018] and Iris [Jung et al. 2017]. More broadly, these two tools have advertised the benefits of the magic wand operator and of the ramified frame rules.

The introduction of the magic wand between postconditions, written $Q_1 \mathrel{-\!\!\star} Q_2$ (as opposed to the use of an explicit quantification $\forall v.\ Q_1\,v \mathrel{-\!\!\star} Q_2\,v$) and the five equivalent characterizations of this operator (Definition 7.4), appear to be a (minor) contribution of the present paper.

Tutorials on Separation Logic. There exists a number of course notes on Separation Logic. Many of them follow the presentation from Reynolds' article [2002] and course notes [2006]. These course notes consider languages with mutable variables, whose treatment adds complexity to the reasoning rules. The Separation Logic is presented as a first-order logic on its own, without attempt to relate it in a way or another to the higher-order logic of a proof assistant. The soundness of the logic is generally only skimmed over, with a few lines explaining how to justify the frame rule.

A few, more recent courses present Separation Logic in relation with its application in mechanized proofs. Appel's book *Program Logics For Certified Compilers* [2014] presents a formalization of a Separation Logic targeting the C semantics from CompCert [Leroy 2009]. Again, the presence of

mutable variables, in addition to other specificities of the C memory model, makes the presentation unnecessarily complex for a first exposure to Separation Logic and to its soundness proof.

In the scope of Separation Logic for mechanized proofs, we are aware of two other tutorials that target a λ-calculus based language, with immutable variables and return values for terms.

The Iris tutorial by Birkedal and Bizjak [2018] presents the core ideas of Iris' concurrent Separation Logic [Krebbers et al. 2017]. Chapters 3 and 4 introduce heap predicates and Separation Logic for sequential programs. Unlike in Iris' Coq formalization, which leverages a shallow embedding of Separation Logic, the tutorial presents the heap predicate in deep embedding style, via a set of typing rules for heap predicates. The realization of these predicates is not explained, and the tutorial does not discuss how the reasoning rules are proved sound with respect to the small-step semantics of the language. The logic presented targets partial correctness, not total correctness, and it is necessarily affine.

Chlipala's course notes [Chlipala 2018a] include a 5-page chapter on Separation Logic. This chapter focuses on the core of the logic—it does not cover any of the enhancements listed in the introduction. Interestingly, the chapter is accompanied with a corresponding Coq formalization, which includes a soundness proof [Chlipala 2018b]. The programming language is described in *mixed-embedding* style: the syntax includes a constructor Bind, which represents bindings using Coq functions, in higher-order abstract syntax style. The rest of the syntax consists of operations for allocation and deallocation, for reading and writing integer values into the heap, plus the constructors Return, Loop, and Fail. These constructs are dependently-typed: a term that produces a value of type α admits the type $\mathsf{cmd}\,\alpha$. Altogether, this design allows for minimalistic formal definitions of the source language, yet, we believe, at the cost of an increased cost of entry for the reader unfamiliar with the techniques involved. The core heap predicates are formalized like in Ynot [Chlipala et al. 2009], i.e., essentially like in Definition 3.3. Triples are defined in deep embedding style, via an inductive definition whose constructors correspond to the reasoning rules. This deep embedding presentation requires "not-entirely-obvious" inversion lemmas, which are not needed in our approach. The soundness proof establishes a partial correctness result expressed via preservation and progress lemmas. This approach is well suited for reasoning about an operating system kernel that should never terminate, or reasoning about concurrent code. However, for reasoning about sequential executions of functions that do terminate, our total correctness proof carried out with respect to a big-step semantics yields a stronger result, via a simpler proof.

11 CONCLUSION

Let us conclude by citing the last paragraph from Reynolds' seminal paper [2002].

> *It can be discouraging to go back and read the "future direction" sections of old papers, and to realize how few of the future directions have been successfully pursued. It will be fortunate if half of the ideas and suggestions in this section bear fruit. In the meantime, however, the field is young, the game's afoot, and the possibilities are tantalizing.*

Well, not just half of the ideas from this paper were relevant to the field of program verification—*all of them* turned out to be *extremely relevant*. After two decades of fruitful research in Separation Logic, its possibilities still appear tantalizing. We hope that our course material will help more students get introduced to the powerful ideas of Separation Logic.

ACKNOWLEDGMENTS

I wish to thank François Pottier and Armaël Guéneau, with whom I have worked on extensions of Separation Logic. I am also grateful to Jacques-Henri Jourdan, who showed me several tricks

for working with the magic wand operator and with weakest preconditions. I also with to thank ... who contributed to the historical notes.

REFERENCES

Anonymous. 2020. Supplementary material, submitted with the paper.

Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software (ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17.

Andrew W Appel. 2014. *Program logics for certified compilers*. Cambridge University Press. With Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy.

Andrew W Appel and Sandrine Blazy. 2007. Separation logic for small-step Cminor. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 5–21.

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press, USA.

Lars Birkedal and Aleš Bizjak. 2018. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. https://iris-project.org/tutorial-material.html

Lars Birkedal, Noah Torp-smith, and Hongseok Yang. 2006. Semantics of separation-logic typing and higher-order frame rules for algol-like languages. In *Logical Methods in Computer Science*.

R. M. Burstall. 1972. Some Techniques for Proving Correctness of Programs which Alter Data Structures. In *Machine Intelligence 7*, B. Meltzer and D. Mitchie (Eds.). Edinburgh University Press, Edinburgh, Scotland., 23–50.

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. 418–430.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

Adam Chlipala. 2011. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. *SIGPLAN Not.* 46, 6 (June 2011), 234–245. https://doi.org/10.1145/1993316.1993526

Adam Chlipala. 2013. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 391–402.

Adam Chlipala. 2018a. Formal reasoning about programs. http://adam.chlipala.net/frap/frap_book.pdf Course notes.

Adam Chlipala. 2018b. Formal reasoning about programs, Coq material for Chapter 14. https://github.com/achlipala/frap/blob/master/SeparationLogic.v

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2009. Effective Interactive Proofs for Higher-Order Imperative Programs. In *ACM International Conference on Functional Programming (ICFP)*.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. *SIGPLAN Not.* 48, 1 (Jan. 2013), 287–300. https://doi.org/10.1145/2480359.2429104

R. W. Floyd. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics)*, Vol. 19. 19–32.

Michael J. C. Gordon. 1989. *Mechanizing Programming Logics in Higher Order Logic*. Springer-Verlag, Berlin, Heidelberg, 387–439.

Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. 2019. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *Interactive Theorem Proving (ITP) (Leibniz International Proceedings in Informatics)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:20.

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *European Symposium on Programming (ESOP)*.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. 12, 10 (1969), 576–580. http://doi.acm.org/10.1145/363235.363259

Aquinas Hobor and Jules Villard. 2013. The Ramifications of Sharing in Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2429069.2429131

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. *SIGPLAN Not.* 36, 3 (Jan. 2001), 14–26. https://doi.org/10.1145/373243.375719

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. Springer-Verlag, Berlin, Heidelberg, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. 2006. Formal Verification of the Heap Manager of an Operating System Using Separation Logic. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering (ICFEM'06)*. Springer-Verlag, Berlin, Heidelberg, 400–419. https://doi.org/10.1007/11901433_22

Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare Logic for Realistically Modelled Machine Code. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*. Springer-Verlag, Berlin, Heidelberg, 568–582.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and Separation in Hoare Type Theory. *SIGPLAN Not.* 41, 9 (Sept. 2006), 62–73. https://doi.org/10.1145/1160074.1159812

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2008. Hoare Type Theory, Polymorphism and Separation1. *J. Funct. Program.* 18, 5–6 (Sept. 2008), 865–911. https://doi.org/10.1017/S0956796808006953

Zhaozhong Ni and Zhong Shao. 2006. Certified Assembly Programming with Embedded Code Pointers. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. Association for Computing Machinery, New York, NY, USA, 320–333. https://doi.org/10.1145/1111037.1111066

O'Hearn, Reynolds, and Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL: 15th Workshop on Computer Science Logic*. LNCS, Springer-Verlag.

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968

Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *The Bulletin of Symbolic Logic* 5, 2 (1999), 215–244. http://www.jstor.org/stable/421090

Benjamin C. Pierce and many contributors. 2016. Software Foundations. https://softwarefoundations.cis.upenn.edu/

François Pottier. 2017. Verifying a hash table and its iterators in higher-order separation logic. In *ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)*. 3–16. https://doi.org/10.1145/3018610.3018624

Aquinas Hobor Qinxiang Cao, Shengyi Wang and Andrew W. Appel. 2018. Proof pearl: Magic wand as frame. Unpublished.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. 55–74.

John C Reynolds. 2006. A short course on separation logic. http://cs.ioc.ee/yik/schools/win2006/reynolds/estslides.pdf

Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, Bytes, and Separation Logic. *SIGPLAN Not.* 42, 1 (Jan. 2007), 97–108. https://doi.org/10.1145/1190215.1190234

Thomas Tuerk. 2010. Local Reasoning about While-Loops. In *VSTTE LNCS*.

Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A practical verification framework for preemptive OS kernels. In *International Conference on Computer Aided Verification*. Springer, 59–79.

A STATISTICS ON THE MINIMAL SOUNDNESS PROOF FOR SEPARATION LOGIC

	Number of definitions	Number of lemmas	Lines of Tactic defs.	Lines of Proofs
Syntax of the language	8			
Substitution and big-step semantics	2			
Tactic for heap equality and disjointness			5	
Extensionality axioms	2			
Core heap predicates	7			
Entailment	2	4	2	4
Properties of separating conjunction		9		21
Properties of other operators		11		14
Tactic for entailments		2	17	3
Lemmas for heap-manipulating primitives		4		13
Hoare triples and associated rules	1	14		41
Separation Logic and associated rules	1	15		21
Total	23	59	24	117

Fig. 3. Statistics for the Coq formalization

B EXAMPLE PROOFS OF REASONING RULES

B.1 Proof of a structural rule

To illustrate the kind of reasoning involved in the proof of structural rules, consider the proof of the combined consequence-frame rule.

Lemma B.1 (Combined consequence-frame rule).

$$\frac{H \vdash H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \mathrel{-\!\!\star} H_2 \vdash Q}{\{H\}\ t\ \{Q\}}\ \textsc{consequence-frame}$$

Proof. The consequence rule is straightforward to establish for Hoare triples, based on the definition of entailment (Definitions 3.7 and 3.14) and of Hoare triples (Definition 5.1), in which the precondition appears as hypothesis and the postcondition as conclusion of a logical implication.

Let us prove the consequence-frame rule with respect to the interpretation of Separation Logic triples given in Definition 5.2.

The conclusion $\{H\}\ t\ \{Q\}$ is equivalent to $\forall H'.\ ^{\text{HOARE}}\{H \star H'\}\, t\, \{Q \star H'\}$. Consider a particular heap predicate H'. Invoking the premise $\{H_1\}\ t\ \{Q_1\}$ on the predicate $H_2 \star H'$ yields $^{\text{HOARE}}\{H \star (H_2 \star H')\}\, t\, \{Q \mathrel{-\!\!\star} (H_2 \star H')\}$. By the consequence rule of Hoare logic, to derive $^{\text{HOARE}}\{H \star H'\}\, t\, \{Q \mathrel{-\!\!\star} H'\}$, it suffices to establish the entailments $H \star H' \vdash H \star (H_2 \star H')$ and $Q \mathrel{-\!\!\star} (H_2 \star H') \vdash Q \mathrel{-\!\!\star} H'$.

To prove the first entailment, first exploit rule star-assoc to rewrite it as $H \star H' \vdash (H \star H_2) \star H'$, then observe that the resulting entailment follows from the premise $H \vdash H_1 \star H_2$ by rule star-monotone-r. To prove the second entailment, let us begin by revealing the definition of the entailment for postconditions (recall Definition 3.14). The second entailment is equivalent to $(Q\, v) \star (H_2 \star H') \vdash (Q\, v) \star H'$. It follows from the premise $Q_1 \mathrel{-\!\!\star} H_2 \vdash Q$, which implies $(Q_1\, v) \star H_2 \vdash (Q\, v)$, by exploiting the rules star-assoc and star-monotone-r just like for the first entailment. □

B.2 Proof of a reasoning rule for terms

To illustrate the kind of reasoning involved in the proof of reasoning rules for terms, consider the case of sequences. The proof is two-step: first establish a Hoare logic reasoning rule for sequences, then derive its Separation Logic counterpart.

Lemma B.2 (Reasoning rule for sequences in Hoare Logic).

$$\frac{{}^{\text{HOARE}}\{H\}\, t_1\, \{\lambda v.\, H'\} \qquad {}^{\text{HOARE}}\{H'\}\, t_2\, \{Q\}}{{}^{\text{HOARE}}\{H\}\, (t_1\,;\, t_2)\, \{Q\}}\ \textsc{hoare-seq}$$

Proof. The evaluation rule for sequence is a simplified version of the evaluation rule for let-bindings, stated as shown below.

$$\frac{t_1/s \Downarrow v_1/s' \qquad t_2/s' \Downarrow v/s''}{(t_1\,;\, t_2)/s \Downarrow v/s''}\ \textsc{eval-seq}$$

Recall from Definition 5.1 the interpretation of Hoare triples. Consider a state s satisfying the precondition H, that is, such that $H\, s$ holds. The goal is to find v and s' such that $(t_1\,;\, t_2)/s \Downarrow v/s'$ and $Q\, v\, s'$ hold.

By the first premise ${}^{\text{HOARE}}\{H\}\, t_1\, \{\lambda v.\, H'\}$ applied that state s, there exists v_1 and s_1' such that $t_1/s \Downarrow v_1/s_1'$ and $(\lambda v.\, H')\, v_1\, s_1'$ hold. The latter simplifies to $H'\, s_1'$.

By the second premise $\{H'\}\ t_2\ \{Q\}$ applied to the state s_1', which satisfies the precondition H', there exists v and s' such that $t_2/s \Downarrow v/s'$ and $Q\, v\, s'$ hold. The latter corresponds to one half of the conclusion.

Applying the evaluation rule for sequence EVAL-SEQ to the judgments $t_1/s \Downarrow v_1/s_1'$ and $t_2/s \Downarrow v/s'$ yields $(t_1\,;\, t_2)/s \Downarrow v/s'$, which corresponds to the second half of the conclusion. □

Lemma B.3 (Reasoning rule for sequences in Separation Logic).

$$\frac{\{H\}\ t_1\ \{\lambda v.\, H'\} \qquad \{H'\}\ t_2\ \{Q\}}{\{H\}\ (t_1\,;\, t_2)\ \{Q\}}\ \textsc{seq}$$

Proof. Recall from Definition 5.2 the interpretation of Separation Logic triples: $\{H\}\ (t_1\,;\, t_2)\ \{Q\}$ is equivalent to $\forall H''.\ {}^{\text{HOARE}}\{H \star H''\}\, (t_1\,;\, t_2)\, \{Q \star H''\}$. Consider a particular heap predicate H''.

By the first premise $\{H\}\ t_1\ \{\lambda v.\, H'\}$ applied to that H'', one derives ${}^{\text{HOARE}}\{H \star H''\}\, t_1\, \{(\lambda v.\, H') \star H''\}$. In that judgment, the postcondition $(\lambda v.\, H') \star H''$ simplifies to $\lambda v.\, (H' \star H'')$.

By the second premise $\{H'\}\ t_2\ \{Q\}$ applied to the same H'', one derives ${}^{\text{HOARE}}\{H' \star H''\}\, t_2\, \{Q \star H''\}$.

Applying the Hoare logic reasoning rule for sequences (HOARE-SEQ from Lemma B.2) to the two judgments ${}^{\text{HOARE}}\{H \star H''\}\, t_1\, \{\lambda v.\, (H' \star H'')\}$ and ${}^{\text{HOARE}}\{H' \star H''\}\, t_2\, \{Q \star H''\}$ yields ${}^{\text{HOARE}}\{H \star H''\}\, (t_1\,;\, t_2)\, \{Q \star H''\}$, as required. □

C VERIFICATION OF THE INCREMENT FUNCTION

To illustrate the use of the reasoning rules, let us present the proof of the increment function. Consider the following implementation.

$$\begin{array}{lll} \mathsf{incr} & \equiv & \hat{\mu}\_.\lambda p.\ \ \mathsf{let}\ n = \mathsf{get}\, p\ \mathsf{in} \\ & & \quad\ \mathsf{let}\ m = (+)\, n\, 1\ \mathsf{in} \\ & & \quad\ \mathsf{set}\, p\, m \end{array}$$

Example C.1 (Verification of the increment function). The following statement holds.

$$\forall p\, n.\ \ \{p \hookrightarrow n\}\ (\mathtt{incr}\, p)\ \{\_.\ \ p \hookrightarrow (n+1)\}$$

Proof. Consider particular values of p and n.

- Applying rule APP leaves:

$$\{p \hookrightarrow n\}\ (\mathsf{let}\ n = \mathsf{get}\ p\ \mathsf{in}\ \mathsf{let}\ m = (+)\ n\ 1\ \mathsf{in}\ \mathsf{set}\ p\ m)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

- Applying rule LET with Q' instantiated as $\lambda r.\ [r = n] \star (p \hookrightarrow n)$ leaves two subgoals. The first one is: $\{p \hookrightarrow n\}\ (\mathsf{get}\ p)\ \{\lambda r.\ [r = n] \star (p \hookrightarrow n)\}$. It is an instance of the rule GET. The second subgoal is:

$$\forall v.\quad \{[v = n] \star p \hookrightarrow n\}\ (\mathsf{let}\ m = (+)\ v\ 1\ \mathsf{in}\ \mathsf{set}\ p\ m)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

- Introducing v, applying the rule PROP to extract $[v = n]$ from the precondition, then substituting n for v turns the proof obligation into:

$$\{p \hookrightarrow n\}\ (\mathsf{let}\ m = (+)\ n\ 1\ \mathsf{in}\ \mathsf{set}\ p\ m)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

- Applying again the rule LET, this time with Q' instantiated as $\lambda r.\ [r = n+1] \star (p \hookrightarrow n)$. leaves two subgoals. The first one is: $\{p \hookrightarrow n\}\ ((+)\ n\ 1)\ \{\lambda r.\ [r = n+1] \star (p \hookrightarrow n)\}$. Applying the FRAME rule with H' instantiated as $p \hookrightarrow n$ yields an instance of the rule ADD. The second subgoal is:

$$\forall v.\quad \{[v = n+1] \star p \hookrightarrow n\}\ (\mathsf{set}\ p\ v)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

- Introducing and eliminating v simplifies the proof obligation into an instance of rule SET:

$$\{p \hookrightarrow n\}\ (\mathsf{set}\ p\ (n+1))\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

□

D BENEFITS OF THE FRAME RULE IN THE PROOF OF A RECURSIVE FUNCTION

The frame rule allows for simpler proofs than what could be achieved without it. To substantiate this claim, let us present the proof of the copy function in Separation Logic, then discuss how it would be more complicated without the frame rule.

```
let rec mcopy p =
  if p == null
    then null
    else { head = p.head; tail = mcopy p.tail }
```

PROOF. The proof of the specification stated in Example 2.12 is carried out by induction on the length of the list L. The induction hypothesis allows in particular to assume the specification to hold for the recursive call.

At the entry of the body of the function, the state corresponds to the precondition, that is, to $\mathsf{Mlist}\ L\ p$. To reason by case analysis on whether p is null, we exploit Definition 2.9.

- Case $p = \mathsf{null}$. In this case, $L = \mathsf{nil}$ and the function returns the value null. This value is named p' in the postcondition. The new piece of postcondition to establish is $\mathsf{Mlist}\ L\ p'$. By Definition 2.8, because $L = \mathsf{nil}$, the predicate $\mathsf{Mlist}\ L\ p'$ is equivalent to $p' = \mathsf{null}$. Hence, the postcondition is satisfied.
- Case $p \neq \mathsf{null}$. In this case, L decomposes as $x :: L'$, and the current state is described by the heap predicate $(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\ L'\ q)$. The operations performed by the code are verified as follows.
  - The read operation `p.head` returns the value x, and `p.tail` returns the address q.
  - The recursive call `mcopy` q creates a copy of the list represented by L'. By induction hypothesis applied to L', this recursive call can be assumed to satisfy the triple:

$$\{\mathsf{Mlist}\ L'\ q\}\ (\mathtt{mcopy}\ q)\ \{\lambda r'.\ \exists q'.\ [r' = q'] \star (\mathsf{Mlist}\ L'\ q) \star (\mathsf{Mlist}\ L'\ q')\}.$$

– Applying the frame rule to that triple and to $(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)$ yields the triple:

$\{(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q)\}$
$(\mathtt{mcopy}\, q)$
$\{\lambda r'.\ \exists q'.\ [r' = q'] \star (p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q) \star (\mathsf{Mlist}\, L'\, q')\}$

which enables reasoning about the recursive call in the current state.
– Let q' denote the result of the recursive call, and let p' denote the result of the record allocation operation`{ head = p.head; tail = q' }`. This record allocation produces a heap described by $(q.\mathsf{head} \hookrightarrow x) \star (q.\mathsf{tail} \hookrightarrow q')$.
– Thus, the final state is described by:

$$(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q) \star (p'.\mathsf{head} \hookrightarrow x) \star (p'.\mathsf{tail} \hookrightarrow q') \star (\mathsf{Mlist}\, L'\, q')$$

which may be folded to $(\mathsf{Mlist}\, L\, p) \star (\mathsf{Mlist}\, L\, p')$, matching the claimed postcondition. □

Intuitively, in the above reasoning, the frame rule is invoked at each recursive call on the head cell of the input list. One is therefore able to reason about a recursive call to `mcopy` by assuming that it makes a copy of a sublist, *independently* of all the cells that have already been traversed by the outer recursive calls to `mcopy`.

Without the frame rule, one would have to describe the full list at an arbitrary point during the recursion. Doing so requires describing the *list segment* made of cells ranging from the head of the initial list up to the pointer on which the current recursive call is made. Stating an invariant involving list segments is doable yet involves more complex definitions and assertions.

More generally, for a program manipulating tree-shaped data structures, the frame rule saves the need to describe a tree with a subtree carved out of it. The ability to invoke the frame rule in proofs carried out by induction allows to reason *locally* about the work performed by the recursive call, without having to explicitly describe the whole *context* in which this recursive call is taking place, thereby saving a significant amount of proof effort.

E REASONING ABOUT HIGHER-ORDER FUNCTIONS

Example E.1 (Reynold's CPS-append challenge). Consider the (nontrivial) function shown below. It takes as arguments two mutable lists, and returns an address describing the head of a list whose cells correspond to the concatenation of the two input lists. The function is implemented by means of a recursive function that expects a continuation, named `k` in the code, to be invoked on the output list. Each recursive call is made with a fresh continuation, responsible for updating the tail pointer of the list cell at hand before invoking the current continuation.

```
let cps_append p1 p2 =
  let rec aux p k =
    if p == null
      then k p2
      else aux p.tail p2 (fun r => (p.tail <- r); k p)
    in
 aux p1 (fun r => r)
```

This function admits the following Separation Logic specification, which describes the two disjoint input lists represented by L_1 and L_2, respectively, and the result list represented by $(L_1 +\!\!+ L_2)$.

$$\{(\mathsf{Mlist}\, L_1\, p_1) \star (\mathsf{Mlist}\, L_2\, p_2)\}\ (\mathsf{cps\_append}\, p_1\, p_2)\ \{\lambda r.\ \exists p_3\ [r = p_3] \star \mathsf{Mlist}\, (L_1 +\!\!+ L_2)\, p_3\}$$

The crux of the proof consists of providing the appropriate specification for the internal recusive function aux. This specification reads as follows. Assume the continuation k to admit the postcondition Q when invoked on a list p_3 describing a list of the form $L \mathbin{+\!\!+} L_2$, and with an auxiliary heap described by H. Then, the call $\mathsf{aux}\, p_1\, k$ also admits the postcondition Q (indeed, the ultimate action performed by aux is to invoke its continuation k) under the precondition including $(\mathsf{Mlist}\, L\, p)$, which corresponds to a sublist of $(\mathsf{Mlist}\, L_1\, p_1)$, including $(\mathsf{Mlist}\, L_2\, p_2)$, which is exploited only in the base case where p becomes null, and including H.

$$\begin{array}{lll} \forall p\, k\, L. & & \big(\forall p_3.\ \{\mathsf{Mlist}\ (L \mathbin{+\!\!+} L_2)\, p_3 \ \star\ H\}\ (k\, p_3)\ \{Q\}\big) \\ & \Rightarrow & \{(\mathsf{Mlist}\, L\, p) \ \star\ (\mathsf{Mlist}\, L_2\, p_2) \ \star\ H\}\ (\mathsf{aux}\, p_1\, k)\ \{Q\} \end{array}$$

In this statement, H describes, in an abstract way, the cells from list p_1 that have already passed by. The introduction of such an abstract heap predicate H is a common pattern for CPS-style functions.

Remark: a proof for the function cps_append was first conducted in CFML [Charguéraud 2011].

F ALTERNATIVE STRUCTURAL RULES

Lemma 5.4 presents 4 core structural reasoning rules: CONSEQUENCE, FRAME, PROP and EXISTS.

The rule PROP for extracting pure facts may in fact be seen as a particular instance of the rule EXISTS for extracting existential quantifiers. Indeed, as pointed out in Remark 3.4, the heap predicate $[P]$ is equal to $\exists (p : P).\ [\,]$. Besides, the quantification "$\forall (p : P).\ \ldots$" is equivalent to the implication "$P \Rightarrow \ldots$".

The rule EXISTS is very useful in practice, although its statement does not appear in the original papers on Separation Logic. These papers instead formulated a rule featuring an existential quantifier both in the precondition and the postcondition. In a ML-style language, it would corresponds to the rule EXISTS2 shown below. The rules EXISTS and EXISTS2 yield equivalent expressive power, that is, they may be derived from one another (in the presence of the rule CONSEQUENCE, and EXISTS-R and EXISTS-L from Figure 1). Compared with EXISTS2, the statement of EXISTS is more concise and better-suited for practical purpose.

The rule FORALL, stated below, is also useful in practice. This rule is derivable from the CONSEQUENCE, and FORALL-L from Figure 1.

Lemma F.1 (Other structural rules).

$$\frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\ H\}\ t\ \{\lambda v.\ \exists x.\ (Q\, v)\}}\ \textsc{exists2} \qquad\qquad \frac{\{[a/x]\, H\}\ t\ \{Q\}}{\{\forall x.\ H\}\ t\ \{Q\}}\ \textsc{forall}$$

G ARRAYS AND RECORDS

This section briefly summarizes the key ideas involved in the treatment of arrays and records. Details may be found in the accompanying Coq development.

Note that the presentation does not take into account the notion of "allocated blocks", and the usual restriction that the free operation can only be invoked on the head of an allocated block. We are currently working on a refinement of the definitions that would enforce this property.

G.1 Arrays

The programming language is assumed to include two additional primitive operations: alloc n for allocating n consecutive cells, and dealloc $n\, p$ for deallocating n consecutive cell starting from address p. It is possible to refine the Separation Logic to ensure that deallocation operations are only invoked on the head of allocated blocks—details are beyond the scope of the present paper.

The allocated cells are assigned as contents a special *uninitialized value*, written $\bot$. The semantics of read operations may be adapted to prevent read operations in uninitialized cells, by adding a premise of the form $v \neq \bot$ to the specification of get.

The i-th cell of an array allocated at address p corresponds to the cell at address $p + i$.

Definition G.1 (Representation of consecutive cells). The heap predicate $\mathsf{cells}\, L\, p$ describes consecutive cells allocated in the range $[p, p + |L|)$, whose contents are the items from the list L.

$$\begin{array}{rcl} \mathsf{cells}\, L\, p & \equiv & \mathsf{match}\, L\, \mathsf{with} \mid \mathsf{nil} \Rightarrow [\,] \\ & & \qquad\qquad \mid x :: L' \Rightarrow (p \hookrightarrow x) \star (\mathsf{cells}\, L'\, (p+1)) \end{array}$$

Lemma G.2 (Specification of array operations).

$$\begin{array}{rcllll} n \geq 0 & \Rightarrow & \{[\,]\} & (\mathsf{alloc}\, n) & \{\lambda r.\ \exists p.\ [r = p] \star \mathsf{cells}\, (\mathsf{List.make}\, n\, \bot)\, p\} \\ n = |L| & \Rightarrow & \{\mathsf{cells}\, L\, p\} & (\mathsf{dealloc}\, n\, p) & \{\lambda \_.\ [\,]\} \\ 0 \leq i < |L| & \Rightarrow & \{\mathsf{cells}\, L\, p\} & (\mathsf{array\_get}\, i\, p) & \{\lambda r.\ [r = \mathsf{List.nth}\, i\, L] \star \mathsf{cells}\, L\, p\} \\ 0 \leq i < |L| & \Rightarrow & \{\mathsf{cells}\, L\, p\} & (\mathsf{array\_set}\, i\, v\, p) & \{\lambda \_.\ \mathsf{cells}\, (\mathsf{List.update}\, i\, v\, L)\, p\} \end{array}$$

For functions that process an array by making recursive calls to increasingly-smaller segments of the array, the following *range-split* lemma allows splitting the segment at hand and applying the frame rule to the segment that is not involved in the recursive call. One thereby gets for free the fact that cells in that segment are unmodified during the recursive call.

Lemma G.3 (Splitting a range of cells).

$$(\mathsf{cells}\, (L_1 +\!\!+ L_2)\, p) \;=\; (\mathsf{cells}\, L_1\, p) \star (\mathsf{cells}\, L_2\, (p + |L_1|))$$

Another useful result is the following *focus* lemma, which allows isolating the i-th cell out of a range of consecutive cells starting at address p and described by a list L, so as to perform operations on that cell in isolation from the rest of the range. Subsequently, the cell with its updated contents v may be merged back into the range. This logical operation involves cancelling a magic wand.

Lemma G.4 (Focusing on a cell from a range). Assume $0 \leq i < |L|$.

$$\Big(\mathsf{cells}\, L\, p\Big) \;\vdash\; \Big((p+i) \hookrightarrow (\mathsf{List.nth}\, i\, L)\Big) \star \Big(\forall v.\ ((p+i) \hookrightarrow v) \mathrel{-\!\!\star} \mathsf{cells}\, (\mathsf{List.update}\, i\, v\, L)\, p\Big)$$

G.2 Records

Definition G.5 (Representation of record fields). A record field is represented by the heap predicate $p.k \hookrightarrow v$, which stands for $(p + k) \hookrightarrow v$.

Definition G.6 (Representation of records). Consider for example the record `{head = x; tail = q}`, with the offsets $\mathsf{head} \equiv 0$ and $\mathsf{tail} \equiv 1$. This record may be represented in three different ways:

(1) as "$\mathsf{cells}\, (x :: q :: \mathsf{nil})\, p$", just like an array of length 2,
(2) as "$(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)$", with two separated fields,
(3) as "$\mathsf{record}\, (\mathsf{head}, x) :: (\mathsf{tail}, q) :: \mathsf{nil})\, p$", where the heap predicate $\mathsf{record}\, K\, p$ describes a record at location p and whose field names and contents are described by the association list K.

$$\begin{array}{rcl} \mathsf{record}\, K\, p & \equiv & \mathsf{match}\, K\, \mathsf{with} \mid \mathsf{nil} \Rightarrow [\,] \\ & & \qquad\qquad \mid (k, v) :: K' \Rightarrow (p.k \hookrightarrow v) \star (\mathsf{cells}\, K'\, p) \end{array}$$

To each of these three representations of records correspond different specifications for allocation, deallocation, read and write operations. The third representation is the one that scales better to large programs, because: (1) it groups the fields of a same record into a single predicate, thereby reducing the number of conjuncts, (2) it allows describing the ownership of an arbitrary subset of the fields, (3) it allows providing items in any order, and (4) with appropriate syntactic sugar it may be written in the form $p \hookrightarrow \{\mathsf{head} := x; \mathsf{tail} := q\}$, which is easy to read.

H TREATMENT OF ASSERTIONS (DYNAMIC CHECKS)

The language construct "assert t" expresses a Boolean assertion. If the term t evaluates to the value true, the assertion produces unit. Otherwise, the term "assert t" gets stuck—the program halts on an error. The verification of a program should statically ensure that: (1) the body of every assertion evaluates to true, and (2) the program remains correct when assertions are disabled either via a compiler option such as `-noassert` in OCaml, or via the programming pattern "if debug then assert t", where debug denotes a compilation flag. The ASSERT rule, shown below, satisfies these two properties.

Lemma H.1 (Evaluation rules and reasoning rule for assertions).

$$\frac{t/s \Downarrow \mathsf{true}/s'}{(\mathsf{assert}\ t)/s \Downarrow tt/s'}\ \textsc{eval-assert-enabled} \qquad \frac{}{(\mathsf{assert}\ t)/s \Downarrow tt/s}\ \textsc{eval-assert-disabled} \qquad \frac{\{H\}\ t\ \{\lambda r.\ [r = \mathsf{true}] \star H\}}{\{H\}\ (\mathsf{assert}\ t)\ \{\lambda\_.\ H\}}\ \textsc{assert}$$

Remark H.2 (Assert false). The term "assert false" denotes inaccessible branches of the code. A valid triple for this term can only be derived from a false precondition: {[False]} (assert false) {Q}.

Remark H.3 (Assertions involving write operations). Interestingly, the reasoning rule ASSERT is not limited to read-only terms. For example, consider the Union-Find data structure, which involves the operation `find` that performs path compression. The evaluation of an assertion of the form `assert (find x = find y)` may involve write operations. It nevertheless preserves all the invariants of the data structure. These invariants would be captured by the heap predicate H from rule ASSERT.

The treatment of assertions (§H) in Separation Logic seem to have first been formalized in the CFML tool [Charguéraud 2011].

I TREATMENT OF FUNCTIONS OF SEVERAL ARGUMENTS

Functions of several arguments may be represented as curried functions (`fun x y => t`), as tupled functions (`fun (x,y) => t`), or as native n-ary functions (like, e.g., in the C language),

The curried function $\mu f.\lambda x_1.\lambda x_2.\, t$ is represented as $\mu f.\lambda x_1.(\mu\_.\lambda x_2.t)$. An application takes the form $(v_0\ v_1\ v_2)$. The reasoning rule for such an application, APP2, generalizes the rule APP. A version of this rule may be stated for every arity. Alternatively, an arity-generic rule may be devised, although in practice it requires tactic support for synthesizing the lists of variables and arguments.

Lemma I.1 (Reasoning rule for curried functions of arity 2).

$$\frac{v_0 = \hat{\mu} f.\lambda x_1 x_2.t \qquad \{H\}\ ([v_2/x_2]\,[v_1/x_1]\,[v_0/f]\,t)\ \{Q\} \qquad \mathsf{noduplicates}\,(f :: x_1 :: x_2 :: \mathsf{nil})}{\{H\}\ (v_0\ v_1\ v_2)\ \{Q\}}\ \textsc{app2}$$

A language featuring primitive n-ary functions more naturally admits arity-generic reasoning rules. In such a language, $\mu f.\lambda \overline{x}.t$ denotes a function f expecting a list $\overline{x}$ of arguments of the form "$x_1 :: \ldots :: x_n :: \mathsf{nil}$", and the n-ary application term $v_0\,\overline{v}$ denotes the application of a value v_0 to a list of arguments $\overline{v}$ of the form "$v_1 :: \ldots :: v_n :: \mathsf{nil}$".

Lemma I.2 (Reasoning rule for primitive n-ary functions).

$$\frac{v_0 = \hat{\mu} f.\lambda \overline{x}.t \qquad \{H\}\ [(v_0 :: \overline{v})/(f :: \overline{x})]\,t\ \{Q\} \qquad |\overline{v}| = |\overline{x}| > 0 \qquad \mathsf{noduplicates}\,(f :: \overline{x})}{\{H\}\ (v_0\,\overline{v})\ \{Q\}}\ \textsc{apps}$$

Remark: it is possible to set up coercions in Coq such that an application written in curried style $v_0\ v_1 \ldots v_n$ gets interpreted as the n-ary application $v_0\ (v_1 :: \ldots :: v_n :: \mathsf{nil})$.

J A TACTIC FOR SIMPLIFYING ENTAILMENTS

In practical proofs, entailment relations to be established typically involve dozens of tokens, e.g.:

$$\exists v.\, (q \hookrightarrow v) \star [n = 4] \star (p \hookrightarrow n) \star H \;\vdash\; \exists m.\, (p \hookrightarrow m) \star H \star [m > 0] \star \top$$
$$H1 \star H2 \star \Big((H1 \star H3) \mathrel{-\!\!\star} (H4 \mathrel{-\!\!\star} H5)\Big) \star H4 \;\vdash\; \Big((H2 \mathrel{-\!\!\star} H3) \mathrel{-\!\!\star} H5\Big).$$

Proving such entailment relations by manually invoking the appropriate reasoning rules involves an overwhelming amount of work. Thus, for practical program verification, automation is a must-have. Ideally, an automated procedure should not only be able to discharge true entailments, it should also be able to perform all the "obvious" simplifications, leaving what remains of the entailment for a manual processing step by the user. Nearly all practical verification framework come with some amount of tooling for simplifying entailments.

One may also wish that the simplification tactic systematically produces as output an entailment that is logically equivalent to its input. However, there are a number of simplifications that are technically not equivalence-preserving, yet nearly always desirable to perform. It seems to make sense to nevertheless apply these simplifications, taking into account that the user always has the ability to "lock" certain subexpressions for preventing simplifications involving them.

In what follows, we describe (in informal terms) a strategy for simplifying triples in a systematic and predictable manner. The corresponding tactic is named `xsimpl` in the Coq development. (It is entirely implemented in Ltac.) The tactic named `xpull` is a variant of `xsimpl` that is limited to performing simplifications only on the left-hand side.

Definition J.1 (Tactic for simplifying entailment). The tactic first attempts to perform simplifications in the left-hand side, then in the right-hand side, then it attempts to perform simplifications that involve both sides. It may iterate this process several times, as long as progress is made.

(1) On each of the two sides, independently:
- Remove empty heap predicates and normalize expressions with respect to associativity.
- Collapse occurrences of the predicate $\top$, so that there is at most one occurrence at top-level on each side.
- Bring all existential quantifiers and pure facts to the front of each side.

(2) On the left-hand side:
- Pull out pure facts and existential quantifiers out into the Coq context.
- For each magic wand of the form $(H_1 \star \ldots H_n) \mathrel{-\!\!\star} H'$, if one of the H_i also occurs on the left-hand side, cancel H_i out as explained Lemma 7.3.
- For each magic wand of the form $Q \mathrel{-\!\!\star} Q'$, if a predicate of the form $Q\,v$ also occurs on the left-hand side, then specialize this magic wand to $Q\,v \mathrel{-\!\!\star} Q'\,v$, cancel out $Q\,v$, leaving $Q'\,v$.

(3) On the right-hand side:
- If a pure fact $[P]$ occurs on the right-hand side, remove it and generate a subgoal asserting the proposition P.
- It a quantifier $\exists x.\, H$ occurs on the right-hand side, instantiate x with a value v, leaving $[v/x]\,H$. The value v may be provided by the user (via arguments passed to the tactic `xsimpl`), or it may be realized as a Coq unification variable (a.k.a. *evar*).
- Remove expressions of the form $H \mathrel{-\!\!\star} H$ or $Q \mathrel{-\!\!\star} Q$, which are entailed by $[\,]$.

(4) On both sides:
- If a predicate H occurs on the both sides, and H is not $\top$, then remove H from both sides.
- If the entailment is of the form $H_0 \vdash (H_1 \mathrel{-\!\!\star} H_2)$, then replace it with $(H_1 \star H_0) \vdash H_2$. Likewise, replace $H_0 \vdash (Q_1 \mathrel{-\!\!\star} Q_2)$ with $(Q_1 \star H_0) \vdash Q_2$.
- If the entailment is of the form $[\,] \vdash [\,]$, then the entailment is true.
- It the entailment is of the form $H \vdash \top$, replace it with the proof obligation affine H.

- If the entailment is of the form $H \vdash (\forall x.\, H')$, replace it with the proof obligation $\forall x.\, (H \vdash H')$ and continue simplifying the inner entailment.
- If the goal is of the form $Q \vdash Q'$, replace it with the proof obligation $\forall x.\, (Q\, x \vdash Q'\, x)$ and continue simplifying the inner entailment.